Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Proyecto final de Master

Implementación de un servidor de streaming de vídeo adaptativo

Tutor: Jordi Casademont

Alumno: Eliseo Catalán

19 de marzo de 2009

Agradecimientos:

A Jordi Casademont y Josep Paradells por el apoyo y la paciencia.

A los compañeros de laboratorio y del grupo: Jordi Casals, Jordi Vilaseca, Guillermo, Toni, José Luís, Jacobo, Xavi, Miguel, Marisa, Javi, Alessandro, Gabrielle, Petter, Victoria… Por echarme una mano y hacérmelo pasar tan bien durante este tiempo.

Finalmente a los más importantes: A mis padres y a Tamara.

Índice de Contenidos

Índice de Figuras

Índice de Tablas

Glosario

3G	Tercera Generación
3GP	Third Generation Platform
3GPP	Third Generation Partnership Project
ADU	Application Data Unit
APP	Application Specific Functions
AVI	Audio Video Interleave
BBDD	Bases de datos
CPU	Central Processing Unit
CSRC	Contributing Source
DNS	Domain Name Server
DRM	Digital Rights Management
DSS	Darwin Streaming Server
DVB	Digital Video Broadcast
FBS	Free Buffer Space
GPL	General Public License
GSM	Global System for Mobile Communications
GUI	Graphic User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IEEE	Institute of Electrical & Electronics Engineers
ISO	International Organization for Standarization
IP	Internet Protocol
MP3	Moving Picture Experts Group Layer-3 Audio (audio file format/extension)
MP4	MPEG-4 Part 14
MIME	Multipurpose Internet Mail Extensions
MOV	QuickTime Movie
MPEG-4	Motion Picture Experts Group Layer-4 Video
NADU	Next Application Data Unit
NAT	Network Address Translation
QTSS	QuickTime Streaming Server
RFC	Request For Comments
RTP	Real-time Transfer Protocol
RTSP	Real Time Streaming Protocol
RTT	Round Trip Time
SDES	Source Description Items
SDP	Session Description Protocol
SECAM	Séquentiel Couleur Avec Mémoire (Estándar de televisión francesa)
SMIL	Synchronized Multimedia Integration Language
SSID	Service Set Identifier

SSRC	Synchronization Source
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TTL	Time To Live
UAPROF	User Agent Profile
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunication System
UPC	Universitat Politècnica de Catalunya
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VOD	Video On Demand
VLC	VideoLan Client
WAP	Wireless Application Protocol
WCDMA	Wideband Code Division Multiple Access
WEP	Wired Equivalent Privacy
WLAN	Wireless Local Area Network
WMV	Windows Media Video
WPA	Wi-Fi Protected Access
XML	Extensible Markup Language

1. Introducción

A medida que avanza la tecnología es común encontrarse con nuevos servicios y posibilidades. Un caso particular de dicha evolución es la que tiene lugar en los dispositivos móviles, los cuales aumentan sus capacidades y prestaciones de manera constante. Con el fin de aprovechar la evolución de los terminales, las operadoras y empresas distribuidoras de contenidos buscan nuevas formas de comunicación con los usuarios, lo cual implica el uso y desarrollo de nuevos métodos de presentación y transferencia de información.

Un caso particular de evolución en los terminales móviles es el incremento de dispositivos capaces de reproducir vídeo. Este crecimiento supone, además de una característica interesante para el usuario, una nueva vía de acceso a éste. Gracias a las nuevas capacidades, el usuario es capaz de recibir y visualizar vídeos, ya sea almacenándolos con antelación o vía *streaming*. Las funcionalidades de transmisión de vídeo en terminales móviles[1] se diseñaron inicialmente pensando en una distribución a través de las redes telefónicas celulares; dichas redes tienen unas velocidades de transmisión relativamente reducidas lo cual, junto a las características restrictivas de los terminales móviles, desemboca en el desarrollo de un sistema de transmisión de vídeo con pocos requisitos tanto a nivel de reproducción como de transmisión.

Otra evolución particular de los terminales es la inclusión de componentes que permiten la conexión a redes inalámbricas locales IEEE 802.11[2] o *Bluetooth*[3]. Esta característica, además de permitir el acceso a Internet de una forma doméstica, permite una nueva forma de distribución del contenido hacia el cliente. Esta solución resulta mucho más económica para el usuario debido a que únicamente se requiere un dispositivo capaz de enviar los datos a través de dichas redes inalámbricas.

La combinación de estas dos particularidades da lugar a un incremento en el interés por el desarrollo de nuevas soluciones para la distribución de vídeo de manera local. Dentro de esta serie de soluciones, destaca el caso de *streaming* de vídeo hacia el cliente ya que se trata de una tecnología que permite la reproducción de los contenidos a medida que éstos son descargados, en lugar de forzar al usuario a obtener completamente el vídeo antes de su reproducción. Estas técnicas de reproducción, casi en tiempo real, encajan perfectamente con las características de los terminales y son aplicables tanto para redes de telefonía móvil como para redes inalámbricas locales.

Este Proyecto Final de Master se engloba dentro de la cátedra Red.es[4] ya que tiene como objetivo mejorar la distribución de contenidos de vídeo en *streaming*. El software desarrollado para este proyecto se incorporará al proyecto *Infopoints*, que nace con la intención de desarrollar una plataforma de distribución de contenido capaz, entre otras cosas, de ofrecer un servicio de *streaming* de vídeo para terminales móviles.

[1] Principalmente teléfonos.

[2] www.ieee802.org/11

[3] http://standards.ieee.org/getieee802/download/802.15.1-2005.pdf

[4] http://www.etsetb.upc.es/es/cat_red_es/

2. Objetivos

El objetivo principal de este Proyecto Final de Master es estudiar la viabilidad de la implantación de un servidor de código abierto[5] en una serie de puntos de acceso o plataformas, las cuales están diseñadas para trabajar formando una red mallada. Este servidor debe ser capaz de responder a las peticiones de los clientes conectados vía inalámbrica, procesarlas y servir en la medida de lo posible un vídeo acorde a las características del terminal. En todo momento debe trabajarse con protocolos y sistemas de archivos compatibles con los terminales[6] y la reproducción del vídeo debe ser de tipo *streaming*. Por otra parte se debe permitir cierta interacción con el usuario (inicio, detención, pausado...) en tiempo real, a la vez que, en la medida de lo posible, se trate de adaptar la calidad del vídeo a las circunstancias instantáneas de la red.

Inicialmente, los vídeos servidos deben utilizar la detección de capacidades llevada a cabo en otro módulo del proyecto para ofrecer aquél que mejor se adapte a las características de cada terminal. En una segunda fase del proyecto se propone el estudio de transmisión vía multicast y envío de archivos sin la solicitud previa del cliente, sistema conocido como *push*, además de estudiar un sistema de adaptación de tasa en tiempo real.

La adaptación de contenidos a los clientes es un punto crucial cuando se habla de dispositivos móviles, debido principalmente a la gran variedad de modelos disponibles. De este modo, pueden encontrarse desde terminales con capacidades muy reducidas, hasta dispositivos con características similares a las de un ordenador portátil, ya sea en capacidad de proceso, tamaño de pantalla, formatos aceptados, etcétera. No obstante, pese a la diversidad de dispositivos, se debe partir de una base común para todos los clientes, tanto a nivel de archivos como codificaciones y tasas de transmisión.

Las redes inalámbricas locales[7] utilizan un sistema de acceso al medio compartido, de modo que la llegada de un nuevo cliente o la pérdida de cobertura de éste, suele producir una degradación en la velocidad del resto de usuarios. Por este motivo, además del rango de cobertura irregular del que disponen este tipo de redes, se puede producir una variación de la calidad de conexión sin desplazar el terminal. Debido a la frecuencia con la que se producen desplazamientos, llegadas y salidas de clientes, el ancho de banda disponible se encuentra en constante cambio; de ahí nace la necesidad de estudiar la posibilidad de implementar un sistema de adaptación de tasa en tiempo real que permita a los usuarios continuar con la visualización de un vídeo incluso en condiciones adversas, a costa de perder algo de calidad.

En este proyecto, por lo tanto, se va a presentar una alternativa de software abierto para implantar un servidor de *streaming* de vídeo funcional en las plataformas *InfoPoints*. Por otra parte se indica un sistema de interacción con la base de datos del mecanismo de detección de capacidades y un conjunto de ejemplos para proveer contenido adaptado a los terminales. Finalmente se implementan una serie de modificaciones para lograr un sistema de adaptación de tasa en tiempo real basado en los informes de los terminales. Además se realizan una serie de estudios y alternativas a las soluciones propuestas.

La estructura fundamental del proyecto trata de seguir el siguiente esquema:

[5] Código de libre acceso y modificación.

[6] Comentados en apartados posteriores

[7] En adelante WLAN

El primer apartado consta de una introducción general al proyecto explicando la motivación y objetivos perseguidos.

A continuación se presentan de las principales tecnologías relacionadas con el proyecto entre las que se incluyen algunos estándares de *streaming* de vídeo, el marco de trabajo en el que se definen algunos de estos estándares (PSS), el grupo implicado en su desarrollo (3GPP), el contenedor de vídeo utilizado para la transmisión de vídeo (3GP), el concepto de perfiles de usuario, etcétera. También incluye un breve estado del arte de la adaptación de tasa en el marco en el que se encuentra este proyecto y realiza una descripción de los principales protocolos de comunicación implicados como RTP/RTCP, RTSP o SDP. Además, se presenta el flujo de funcionamiento típico de un servidor de *streaming* a terminales móviles.

El siguiente apartado es el dedicado al servidor en sí; en él se lleva a cabo una presentación de los principales servidores existentes con sus características y deficiencias realizándose una justificación sobre la elección de uno de ellos. Una vez justificado se presentan sus principales características incluyendo instalación básica, funcionamiento e interfaces, listas de reproducción y sistemas de *relay* entre otros.

El tercer gran bloque es el dedicado al desarrollo para Darwin Streaming Server y muestra los principales roles, estructuras de datos y módulos utilizados para la programación en DSS. Incluye ejemplos para la programación de un módulo y el flujo de trabajo que sigue éste.

El siguiente punto abarca las principales implementaciones llevadas a cabo en el servidor DSS, el escenario y la problemática a resolver, el estado actual y la estructura del desarrollo incluyendo ejemplos de código y una sección de pruebas del sistema.

Finalmente se incluye un apartado de las conclusiones extraídas de algunas de las pruebas realizadas en las implementaciones del servidor.

3. Tecnología Implicada y estado del arte

3.1. Introducción

En este apartado se pretende comentar brevemente la tecnología básica implicada en el estudio y desarrollo del proyecto. Esta tecnología abarca fundamentalmente protocolos, formatos de archivo y codificación utilizados para la transmisión de vídeo en tiempo real para terminales móviles. Para un mayor detalle en la definición de estándares, protocolos y características es interesante utilizar la bibliografía propuesta.

Inicialmente se presentan los grupos 3GPP y su marco PSS, en el cual se definen los protocolos y estándares utilizados en los diferentes ámbitos relacionados con los sistemas de telefonía móvil de tercera generación. Estos estándares engloban, entre otros, a la tecnología relacionada con la codificación y encapsulado del vídeo utilizada en el proyecto. Se presentan las motivaciones principales del 3GPP como grupo y la utilidad y necesidades resueltas en el marco PSS, incluyendo la justificación de la elección de ciertas tecnologías y algunas de las características fundamentales de éstas. A continuación se comentan los diferentes protocolos necesarios para llevar a cabo la comunicación entre los terminales y el servidor, sus principales características, funcionalidades y limitaciones.

3.2. El grupo 3GPP

El 3GP o Third Generation Partnership Project[8] es un consorcio formado por grupos de diferentes nacionalidades (Norteamérica, Japón, Europa, China y Corea del Sur) fundado con la intención de definir especificaciones técnicas comunes a nivel global durante el desarrollo de la tercera generación de telefonía móvil.[9] Estas especificaciones se basan en los estándares definidos por la International Telecommunication Union[10] (ITU).

Una de las principales problemáticas relativas a los sistemas de telefonía de segunda generación como el GSM[11] utilizado en Europa, es la incompatibilidad del sistema en diferentes regiones. El desarrollo de la tecnología de tercera generación tenía entre sus objetivos la idea de que el sistema fuese global, es decir, compatible en las diferentes áreas geográficas del consorcio. Para llevar a cabo esta idea se formo el grupo 3GPP bajo el que se definen diferentes propuestas de especificaciones técnicas; propuestas que engloban desde el aspecto más físico de la comunicación[12], hasta aquellas relacionadas con los estándares de vídeo y audio soportados en los terminales. Precisamente este último grupo viene definido en el marco PSS o *Packet-switched Streaming Service* que se presenta a continuación.

[8] www.3gpp.org

[9] En adelante se utiliza indistintamente tercera generación o 3G

[10] www.itu.int

[11] Global System for Mobile communications

[12] Como multiplexaciones en tiempo y frecuencia

3.3. El marco PSS

PSS o *Packet-switched Streaming Service* [1] es un marco de trabajo para la definición de aplicaciones destinadas al *streaming* sobre redes de datos de tercera generación. Entre los flujos a transmitir se comprenden señales de vídeo, audio, imágenes estáticas, texto, etcétera. La finalidad del marco PSS es estandarizar los procesos de codificación, almacenamiento, transmisión y reproducción de los diferentes contenidos, basándose en algunos estándares que se comentarán más adelante. Este proyecto se centra fundamentalmente en el *streaming* de vídeo y audio por lo que no se considerarán los pormenores relacionados con la distribución de otro tipo de contenido, si bien generalmente el sistema de transmisión es muy similar.

El sistema de trabajo del marco PSS se redefine periódicamente, al igual que el resto de características, siguiendo el sistema de *releases,* es decir, cada *release* corresponde a una nueva versión bien diferenciada de las especificaciones técnicas. Estas *releases* son comunes a todas las tecnologías relacionadas, por lo que engloba, entre otros, al marco PSS. A medida que las versiones avanzan se añaden funcionalidades considerables respecto a las anteriores, siendo en cualquier caso compatibles con éstas. De este modo un servidor preparado para una versión concreta de las especificaciones puede interactuar con clientes de cualquier otra versión, las funcionalidades disponibles serán aquellas relativas a la mayor versión común en ambos.

Pese a que el grupo se formó con la idea de trabajar centrados en tecnología de tercera generación, la mayoría de *releases* tienen una versión 2G / EDGE [2] a la que se denomina fase 2. En este proyecto se comentan fundamentalmente aquellas versiones relativas a 3G.

A pesar de que los primeros desarrollos disponibles datan de antes del año 2000, es a partir de entonces cuando nace la primera versión funcional con sistemas de tercera generación, es la conocida como Release 99. Esta versión introducía las principales características de los sistemas de tercera generación, como son: Técnicas de multiplexación de acceso basadas en código CDMA, diversidad radio en los terminales, traspasos suaves entre celdas, etcétera. A nivel PSS no se realiza ninguna definición ya que éste se centra en entornos basados en paquetes. Se puede apreciar una evolución en las *releases* en la Figura 1 así como la disponibilidad de terminales compatibles.

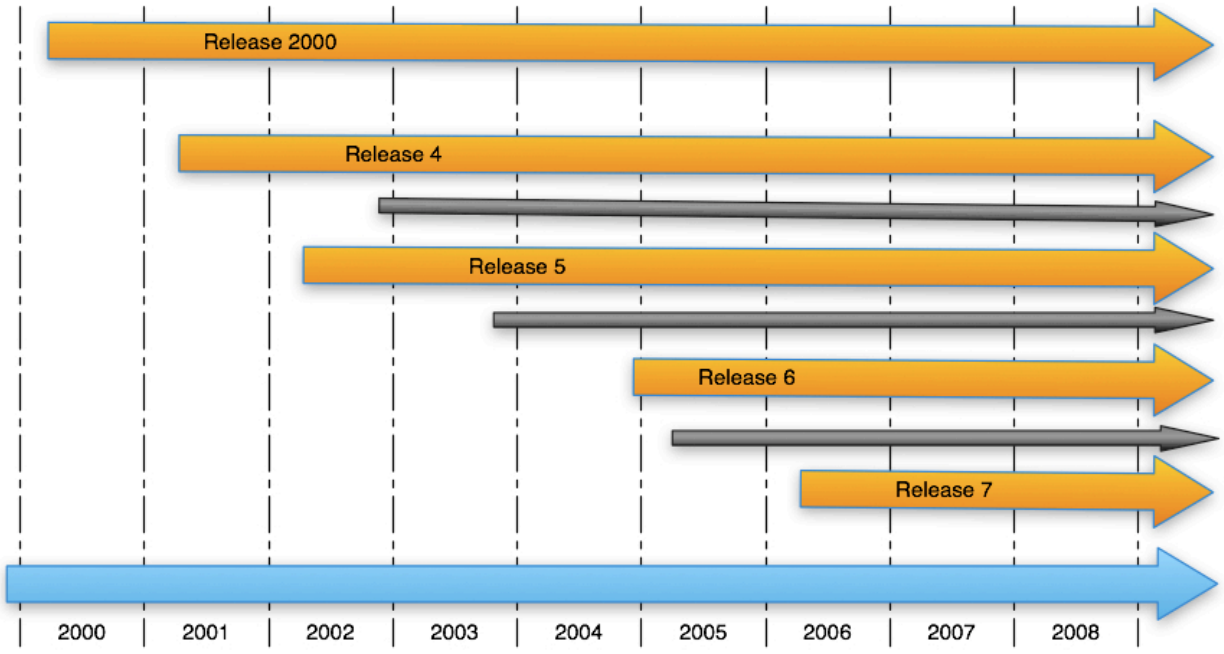

Figura 1: Evolución de las diferentes *releases* de PSS y disponibilidad de terminales.

Una de las principales carencias de esta primera versión era la incompatibilidad con entornos IP, problema que se solventa en la release 4 la cual, además, incluye las primeras definiciones para el marco PSS. En adelante PSS rel-x o release-x, se referirá a las definiciones técnicas del marco PSS contempladas en la versión x del desarrollo. En esta primera release-4 se definen las características básicas del marco PSS que serán comentadas en los apartados posteriores.

En la actualidad, se encuentra desplegada hasta la versión 7, mientras que la octava se encuentra en fase de desarrollo. A continuación se detallan las principales características introducidas en cada una de las versiones presentadas:

Releases	**Principales características**
Release-4	Definición general del marco PSS.
Release-5	Intercambio de capacidades, texto temporizado, MIDI...
Release-6	Adaptación de tasa, QoE, DRM.
Release-7	Arranque rápido, intercambio de contenido.
Release-8	Multimedia Broadcast Multicast Service, interacción del cliente.

Tabla 1: Principales evoluciones de PSS

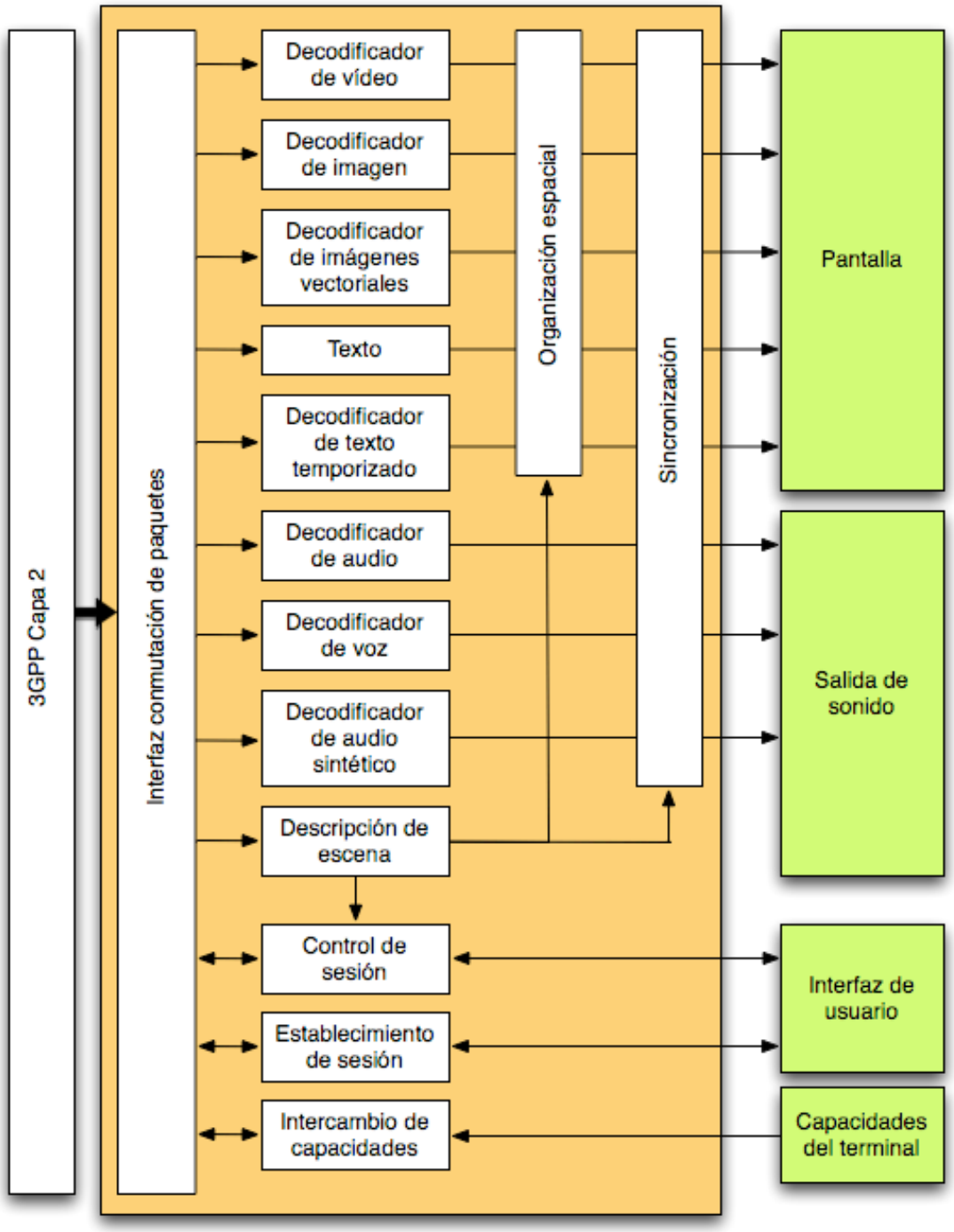

Figura 2: Marco PSS en un dispositivo móvil

En este proyecto se ha trabajado fundamentalmente en las versiones comprendidas entre la cuarta y sexta debido a los terminales y servidores disponibles.

En la Figura 2 se pueden apreciar los diferentes componentes relativos al marco PSS que pueden encontrarse en un cliente de *streaming*.[13] Existen dos bloques fundamentales: El bloque relativo al transporte y el bloque relativo a control.

En el primero se aprecia como la comunicación es unidireccional y comprende la transmisión de la información a reproducir en el terminal. Éste es el encargado de, mediante el uso de información provista por el servidor, recomponer el contenido (vídeo, audio, texto) construyendo los fotogramas en caso de vídeo, sincronizar los diferentes flujos que pudieran existir (vídeo, sonido y subtítulos por ejemplo) y reproducirlo tanto gráfica como sonoramente.

En el apartado referido al control se pueden encontrar tres bloques: El establecimiento de sesión, que tiene lugar en el momento que el usuario realiza la petición de un recurso determinado. El control de la misma, que gestiona la sesión y conexiones pertinentes durante la transmisión. Y finalmente el intercambio de capacidades el cual, pese a no ser parte imprescindible para la comunicación (como sí sucede en los casos anteriores), puede realizar funciones para determinar las características de los terminales de forma optativa. En los apartados correspondientes se presentarán los protocolos utilizados en cada una de las diferentes tareas, los cuales se pueden ver resumidos en la Figura 3.

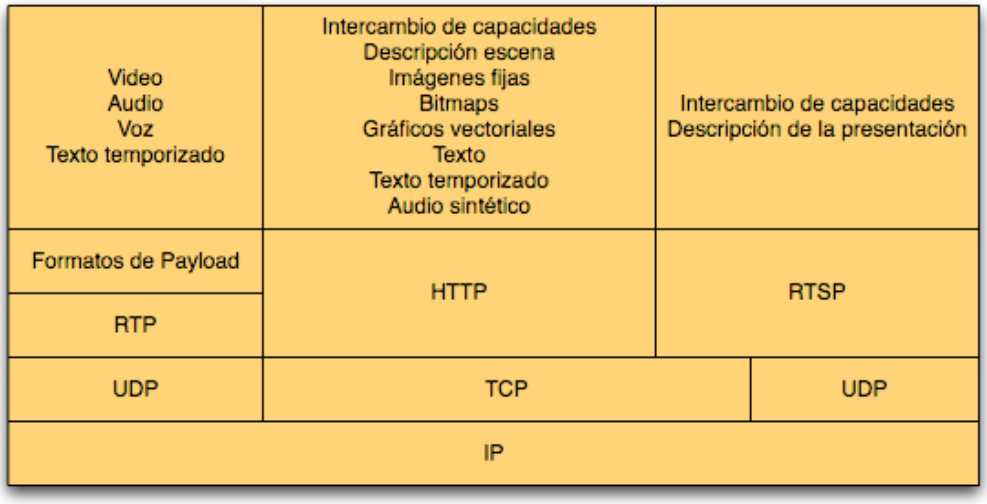

Figura 3: Tabla de los principales protocolos y funcionalidades en el marco PSS

PSS utiliza fundamentalmente los siguientes protocolos para *streaming*: Real Time Protocol RTP [3] Real Time Streaming Protocol RTSP [4], Session Description Protocol SDP [5] y Hypertext Transfer Protocol HTTP [6]. En cuanto al formato de archivo, se utilizará fundamentalmente vídeo encapsulado en el contenedor 3GP que se presentará en el siguiente apartado. Como se mostrará a continuación, el marco PSS está estrechamente relacionado con los archivos 3GP, de hecho, las diferentes versiones de archivos 3GP van ligadas a las versiones de PSS.

[13] En el interior del recuadro naranja

3.4. Contenedor de vídeo 3GP

3.4.1. Definición

Dentro del marco PSS del consorcio 3GP, se contempla la posibilidad de realizar *streaming* de vídeo a terminales móviles. Para ello, entre otras cosas, se define un sistema contenedor de vídeo llamado 3GP [7].

Los archivos 3GP nacen con la idea de proponer un sistema contenedor de vídeo común para todos los terminales[14], de manera que se asegure una compatibilidad entre dispositivos de distintos fabricantes. Además se piensa para compatibilizarlo con diferentes sistemas de acceso como GSM, WCDMA[15] o UMTS[16] entre otros, adaptando sus características a las tasas de transmisión disponibles. Es precisamente esta compatibilidad la que motiva el hecho de que sea el formato elegido para distribuir vídeo a los terminales móviles en este proyecto; de hecho, la mayoría de los fabricantes han adoptado este sistema. En el escenario 3GPP han aparecido dos extensiones de archivo, 3GP y 3G2, aunque las características son muy similares (difieren en la tecnología para la que se pensaron a la hora de distribuirse) en la realización de este proyecto se ha pensado fundamentalmente en 3GP. Cabe destacar que los archivos 3GP permiten, tanto la reproducción mediante *streaming* del contenido, como la reproducción desde un archivo previamente descargado; para la realización de este proyecto se considerará el primer escenario.

Como se ha comentado anteriormente, 3GP no es más que un contenedor de vídeo. Su estructura es muy similar a la de MP4 [9], de hecho, se puede considerar 3GP como una versión simplificada de MP4. La diferencia fundamental entre ellos son las codificaciones de audio soportadas y las resoluciones a las que pueden trabajar. En 3GP es habitual encontrar codificaciones como MPEG-4 y H.263 [10] en cuanto a vídeo y AMR-NB [12] o AAC-LC [13] en cuanto a audio. Es más usual utilizar H.263+AMR-NB ya que hay mayor número de terminales que soportan esta combinación pese a que las otras alternativas suelen ofrecer una mejor calidad. Es importante recordar que inicialmente el formato se plantea para ser utilizado mediante canales de acceso celulares como GSM o UMTS, las cuales proveen una velocidad de acceso relativamente baja.

3.4.2. Formato de cajas: ISO Boxes

Los archivos 3GP al igual que ocurre con los MP4 se basan en un sistema de cajas, las ISO Boxes [14]. Estas cajas contienen, además de las pistas de audio o vídeo del archivo, otra información adicional de control. La organización de las cajas sigue un sistema jerárquico, donde en un archivo ISO se encuentran dos grandes grupos:

Por una parte, la caja relativa a la película (*moov*) en la que se encuentran, al menos, tantas cajas como pistas de audio o vídeo tenga el archivo. De este modo, un vídeo típico estaría compuesto, al menos, por una pista de vídeo y una de audio y tendría, por lo tanto, una caja de tipo *trak* para cada

[14] Fundamentalmente teléfonos

[15] Wideband Code Division Multiple Access

[16] Universal Mobile Telecommunications System - http://www.umtsforum.net/

una de ellas. Igualmente existe la posibilidad de introducir múltiples pistas, tanto para la ejecución en paralelo como en serie. Una película con dos idiomas disponibles contaría con una pista de vídeo y dos de audio, formando un total de tres cajas *trak*. Dentro cada una de las cajas se encuentra todo un árbol de cajas de menor rango que contienen información relativa a dicha pista, desde cabeceras, información del códec contenido, tiempos de decodificación, grupo al que pertenece, etcétera.

Cabe comentar que estas cajas *moov* permiten introducir, además de diferentes flujos de vídeo y audio, unos nuevos elementos que se conocen como *hint tracks*. Estas pistas adicionales son utilizadas por el servidor de *streaming* como ayuda adicional. En este caso será necesario someter a los archivos a un proceso de *hinting* antes de almacenarlos en el servidor. Algunos servidores de *streaming* generan las pistas de hint en el momento en el que inician la reproducción, mientras que otros necesitan que el proceso sea ejecutado con anterioridad. Si se realiza un proceso de *hinting*, se generará una nueva pista (y por lo tanto, una caja) adicional por cada una de las pistas de audio y vídeo existentes. En la Figura 4 se puede apreciar un detalle de un archivo ISO y sus diferentes cajas. Además de las cajas de información, se encuentran las *mdat,* las cuales contienen los datos de cada una de las pistas. Se puede encontrar un listado de las diferentes cajas en el documento correspondiente.

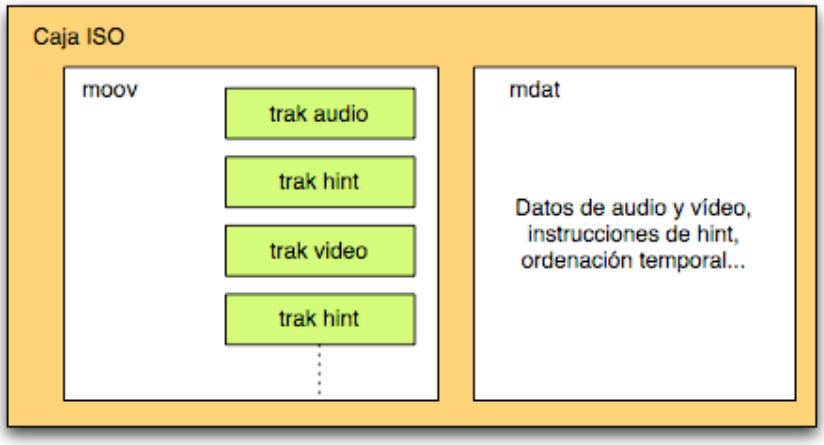

Figura 4: Ejemplo de caja ISO en un 3GP formado por diferentes pistas

3.4.3. Perfiles, grupos, adaptación y calidad de experiencia en 3GP

Dentro de las cajas ISO existen una serie de cabeceras que indican el tipo de archivo y su perfil. En 3GP se pueden encontrar diferentes versiones[17] y perfiles para cada una de ellas. En particular, este informe se centra en las *releases* 4, 5 y 6 [7]. La versión 5 contempla un perfil general para los archivos y es soportada por prácticamente todos los terminales del mercado hoy en día. La versión 6 introduce el concepto de perfiles, presentando el perfil general, básico, de *streaming*, de descarga progresiva y de presentación extendida. Estos perfiles se identifican en el campo *ftyp* de la caja ISO. En este caso interesa el perfil de *streaming*, también conocido como 3GS6. Este perfil introduce un concepto interesante: Permite definir lo que se conocen como *alternate groups* a los que cada flujo puede pertenecer; se considera que dos elementos de un mismo grupo contienen información equivalente y por lo tanto puede entregarse cualquiera de ellos. Pueden pertenecer a estos grupos,

17 Releases

por ejemplo, diferentes pistas que contengan la misma información codificada de diferente manera (*codec*, tasa, resolución...). En la *release* 6, además el perfil 3GS6 se definen: 3GG6, 3GP6, 3GR6 y 3GE6 referentes a perfil general, básico, de descarga progresiva y de presentación extendida respectivamente. Cada uno de ellos se plantea para un escenario concreto y cuenta con diferentes características, aunque este proyecto se centrará en los perfiles generales y de *streaming*.

La idea de estos grupos de alternativas es reunir varios flujos diferentes con el mismo contenido para que.,de este modo, se pueda seleccionar una u otra pista en función de las características y situación del terminal. Esta técnica permitirá empaquetar flujos de vídeo y audio a diferentes tasas en un mismo contenedor de manera que, accediendo al mismo archivo, se pueda obtener la transmisión de un vídeo a diferente tasa, con otra codificación o distinto tamaño. Será condición indispensable para aplicar estas propiedades, que sean soportadas por el servidor.

Este soporte, entre otros, viene dado por la funcionalidad denominada *3GPP-Adaptation-Support* cuya finalidad, como su nombre indica, es ofrecer un contenido adaptado a las circunstancias, funcionalidades y características del terminal. Esta característica es indicada tanto por el servidor como por el cliente a través de los diferentes mensajes del protocolo RTSP (ver apartado 3.7.2). Como se muestra en [7], este sistema se diseña inicialmente para proporcionar un sistema de adaptación durante la transmisión del archivo y se basa en los informes de estado que el cliente envía periódicamente al servidor; gracias a ellos el servidor puede evaluar el estado de la red y realizar la adaptación pertinente. En ningún momento se definen herramientas para realizar una adaptación antes de iniciarse la transmisión del flujo en sí aunque, como se muestra en apartados posteriores, el servidor tiene la posibilidad de conocer algunas características del terminal por adelantado.

En los documentos del 3GPP no se define en ningún momento el sistema a utilizar para realizar la adaptación, definiéndose ésta como dependiente de la aplicación. Cabe destacar que en la actualidad no existe ningún protocolo que permita por parte del cliente o del servidor el intercambio de pistas de *streaming*; del mismo modo los clientes no soportan que se realice un cambio de manera automática por parte del servidor.

Este tipo de contenedor ha sido seleccionado para el proyecto debido a su gran índice de compatibilidad con terminales. Gracias a que fue diseñado para soportar el *streaming* de vídeo desde redes celulares, muchos de los fabricantes han ido incorporando dicha funcionalidad en los terminales. Generalmente los terminales que proveen compatibilidad 3GP también soportan contenedores MP4 debido a su gran similitud. Pese a que los contenedores 3GP se diseñaron con la intención de almacenar contenido de baja tasa, es posible introducir en ellos vídeo de mucha mayor calidad con lo que el estudio de transmisión utilizando 3GP no se limita a terminales de baja capacidad.

3.5. Perfiles de cliente: UAProf

Además de la adaptación dinámica de contenidos ligada a las condiciones de la red, existe otro tipo de adaptación posible para los terminales: Aquella relacionada con las capacidades físicas de éste. En la actualidad existen un buen número de terminales y dispositivos móviles que permiten realizar comunicaciones inalámbricas y que difieren considerablemente en sus especificaciones técnicas. De este modo se pueden encontrar terminales con tamaños de pantalla muy dispares, que soportan diferentes codificaciones, con distintas interfaces gráficas o simplemente terminales idénticos que, debido a una versión más actual de su software, poseen nuevas funcionalidades.

La mayoría de las capacidades de los terminales está definida en su perfil de usuario agente o *UAProf* [15]. Estos perfiles están definidos en páginas XML[18] que varían según el modelo y la versión del dispositivo e incluyen todas las características y capacidades del mismo, útiles para los servidores WEB, de *streaming* u otros contenidos. Es importante destacar que no todos los terminales disponen de dicho perfil.

Para el desarrollo de este proyecto resulta relevante conocer la información contenida en los perfiles relativos al *streaming* como el mostrado en el extracto de la siguiente caja. En ella pueden apreciarse diferentes codificaciones disponibles, la compatibilidad con 3GPR6, el soporte de adaptación, el tamaño de la pantalla, el tipo de sonido disponible, etcétera.

```
<!-- Streaming -->
    <prf:component>
  <rdf:Description rdf:ID="Streaming">
    <rdf:type rdf:resource="http://www.3gpp.org/profiles/PSS/ccppschema-PSS5#Streaming"/>
    <rdf:type rdf:resource="http://www.3gpp.org/profiles/PSS/ccppschema-PSS6#Streaming"/>
    <pss6:StreamingAccept>
        <rdf:Bag>
        <rdf:li>audio/AMR</rdf:li>
        <rdf:li>audio/AAC</rdf:li>
        <rdf:li>audio/3gpp</rdf:li>
        <rdf:li>audio/MP4A-LATM</rdf:li>
        <rdf:li>audio/MP4A-ES</rdf:li>
        <rdf:li>video/3gpp</rdf:li>
        <rdf:li>video/mp4</rdf:li>
    <rdf:li>video/H263-2000;profile=0;level=45</rdf:li>
    <rdf:li>video/MP4V-ES</rdf:li>
        </rdf:Bag>
    </pss6:StreamingAccept>
    <pss6:ThreeGPPLinkChar>No</pss6:ThreeGPPLinkChar>
   <pss6:AdaptationSupport>Yes</pss6:AdaptationSupport>
   <pss6:ExtendedRtcpReports>Yes</pss6:ExtendedRtcpReports>
…
<pss5:AudioChannels>Stereo</pss5:AudioChannels>
      <pss5:RenderingScreenSize>240x320</pss5:RenderingScreenSize>
      <pss5:PssAccept>
….
```

Gracias a toda esta información, es posible entregar a un cliente una información mucho más acorde a las capacidades del terminal consiguiendo una mejor personalización de los contenidos. En apartados posteriores se muestra como, gracias a una de las implementaciones llevadas a cabo, el servidor interactuar con una base de datos que contiene información del *UAProf* para servir un vídeo personalizado a cada terminal según sus capacidades.

[18] Extensible Markup Language.

3.6. Estado del arte en la adaptación de tasa

Actualmente existen un buen número de aproximaciones teóricas a sistemas de adaptación de tasa en tiempo real como por ejemplo las mostradas en [25] y [26]. La mayoría de ellas contemplan la posibilidad de trabajar con archivos multi-pista que permitan realizar un intercambio de éstas en algún momento, lamentablemente en la actualidad no se encuentra ningún protocolo bien definido para solicitar este tipo de intercambios, ya sea desde el cliente o desde el servidor. Como se ha comentado con anterioridad, los sistemas basados en pistas alternativas todavía no se encuentran en un estado de madurez y resulta complicado encontrarse con un terminal lo suficientemente inteligente como para seleccionar entre un grupo de pistas alternativas; desde este punto de vista parece que la labor de éste está limitada a lanzar una serie de informes sobre los datos recibidos, estado del *buffer*, etcétera.

Existen soluciones experimentales basadas en codificaciones adaptativas como por ejemplo H.264 tal y como se presenta en los artículos [20] y [21]. En este tipo de soluciones se pretende utilizar el *feedback* presentado por el cliente para realizar la adaptación, pero partiendo siempre de una pista con codificación escalable. Esta solución es similar a la implementada en las últimas versiones de algunos servidores, permitiendo la eliminación de tramas menos relevantes según las condiciones de las que informa el cliente. Estos sistemas de eliminación de tramas están basados en una priorización de transmisión de cada uno de los paquetes según el nivel al que pertenezcan. Las soluciones presentadas son, en principio, experimentales por lo que no se dispone de implementaciones prácticas.

Otra solución interesante es la propuesta en [22], donde se insta a la utilización de un sistema de métricas relativo a la percepción de calidad del usuario, la cual no está siempre ligada a pérdidas de paquetes, búferes, etcétera. Se trata fundamentalmente de establecer un compromiso entre la calidad percibida por el usuario y las pérdidas estableciendo unos límites tolerables. El artículo basa la adaptación en un sistema similar al implementado aquí, mediante archivos de la familia MP4 o 3GP con múltiples pistas en los que el feedback del cliente es utilizado para realizar los cambios. De nuevo se trata de estudios experimentales.

Existen otro tipo de soluciones como la presentada en [23] donde se pretende utilizar algoritmos de corrección adaptativa a posteriori para conseguir recuperar información perdida en la transmisión. Para ello se utilizan los protocolos existentes añadiendo algunos parámetros adicionales que se intercambian cliente y servidor. Lamentablemente estas soluciones son difícilmente aplicables al escenario contemplado en este proyecto debido a las limitaciones propias de los terminales móviles.

Existen muchas más alternativas que aumentan considerablemente la complejidad de la implementación, desde sistemas con múltiples emisores hasta *proxys* de adaptación de tasa. En la bibliografía pueden encontrarse más artículos que se pueden considerar interesantes en este ámbito.

Desde el punto de vista de implementaciones, se comentará el estado de la técnica en el apartado servidores, no obstante es importante destacar que, en el momento de la redacción del proyecto no estaba disponible ningún servidor que realizara un cambio de flujos de forma automática[19], únicamente algunos eran capaces de interpretar los mensajes de tipo 3GPP que se presentarán posteriormente. Puede destacarse la excepción de los clientes QuickTime los cuales, mediante un protocolo propio de Apple llamado *reliable* UDP establecen una comunicación de control de estado con el servidor.

[19] Aunque existen soluciones de codificación variable, su uso resulta inviable en este proyecto.

Del mismo modo, los terminales disponibles para el experimento no tenían capacidades para la solicitud de dichos intercambios.

Parece evidente pues, que en el momento de realizar este proyecto, la mejor alternativa pasa por la utilización de las herramientas dispuestas por el grupo 3GPP tales como informes de cliente sobre estado del buffer y retardo. Dada la incompatibilidad de ciertos terminales con codificaciones como H.264 [11] se opta por una solución común de adaptación de tasa con archivos multi-pista del tipo MP4/3GP.

3.7. Protocolos de comunicación

3.7.1. Introducción

En este apartado pretende darse una visión general de los diferentes protocolos de comunicación implicados en una transferencia de *streaming* de vídeo entre un servidor y un cliente siguiendo las indicaciones del 3GPP. Se presentan los protocolos de establecimiento y gestión de sesión por parte del cliente (RTSP), los relativos al transporte y control (RTP/RTCP) y el de presentación (SDP) con las principales funciones y mensajes disponibles en cada uno de ellos relativas a este proyecto. Es recomendable referirse a los diferentes estándares y RFC para consultar información más detallada.

3.7.2. Protocolo Real Time Streaming Protocol RTSP

RTSP (IETF RFC 2326)[4] es parte fundamental de la arquitectura PSS y se utiliza para el establecimiento y control de la sesión. Gracias a este protocolo el usuario es capaz de iniciar, detener o pausar la reproducción de un vídeo. Además, si el reproductor se lo permite, podría dar saltos temporales en el *streaming*. RTSP puede utilizarse en combinación con diferentes protocolos aunque lo más habitual, como ocurre en este proyecto, es que gestione conexiones RTP/RTCP.

RTSP guarda ciertas similitudes con protocolos como HTTP[20] al ser ambos protocolos destinados a solicitar información de un elemento de la comunicación a otro; a diferencia de HTTP, RTSP es un protocolo que necesita mantener el estado de las conexiones, además, como se ha comentado anteriormente, la información se transporta en un protocolo diferente.

El protocolo de transporte utilizado para transferir RTP es siempre TCP[21], además cuenta con un puerto buen conocido (554), aunque puede ser modificado tanto en el cliente como en el servidor. Dicha conexión debe permanecer activa durante todo el proceso de comunicación, es decir, desde el momento en el que el cliente lanza la primera petición, hasta que termina de recibir los datos correspondientes al vídeo o audio y utiliza el propio protocolo RTSP para cerrar la sesión. En apartados posteriores se comenta el flujo de funcionamiento completo entre cliente y servidor.

[20] Hypertext Transfer Protocol: v1.1 IETF RFC 2616.

[21] Transmission Control Protocol: IETF RFC 793.

Código	Características
1XX	Informativo: Petición recibida, procesándola.
2XX	Éxito: Petición recibida, procesada y aceptada.
3XX	Redirección: La siguiente acción debe ser ejecutada para completar la petición.
4XX	Error en el cliente: Sintáxis errónea o imposible de procesar.
5XX	Error en el servidor: Error al ejecutar una petición aparentemente correcta.

Tabla 2: Códigos de respuesta en mensajes RTP.

Del mismo modo que ocurre en HTTP, los principales mensajes de los que compone el protocolo RTSP se pueden dividir en peticiones y respuestas. Las peticiones son de tipo textual y son siempre contestadas por un mensaje de tipo respuesta basado en un código numérico como se muestra en la Tabla 2. A continuación se presentan las principales y sus ejemplos:

•**OPTIONS**: Siempre aceptada, es utilizada para solicitar un listado con las peticiones disponibles en el servidor. Junto al código pertinente, la respuesta presenta información textual con todos los tipos de peticiones soportados. Suele ser el primer mensaje lanzado desde el cliente. Aunque en el ejemplo se incluye el recurso al que se desea acceder, no es obligatoria su inclusión.

```
OPTIONS rtsp://192.168.0.100/prueba.3gp RTSP/1.0
CSeq: 11
User-Agent: VLC media player (LIVE555 Streaming Media v2007.02.20)
```

```
RTSP/1.0 200 OK
Server: DSS/6.0.3 (Build/526.3; Platform/Linux; Release/Darwin Streaming Server; State/Development; )
Cseq: 11
Public: DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, OPTIONS, ANNOUNCE, RECORD
```

•**DESCRIBE**: Es una petición que se realiza referida a un recurso concreto, cuyo URL[22] RTSP debe ser incluido en el mensaje. Además del recurso se incluyen métodos aceptados para la recepción de la información de descripción. Es utilizada para conocer los pormenores del recurso solicitado y se responde, además de con un código, con la información de dicho recurso utilizando alguno de los protocolos indicados por el cliente. En este proyecto se considera siempre que el protocolo utilizado para la descripción del recurso es SDP, comentado más adelante.

[22] Uniform Resource Locator

```
DESCRIBE rtsp://192.168.0.100/prueba.3gp RTSP/1.0
CSeq: 12
Accept: application/sdp
User-Agent: VLC media player (LIVE555 Streaming Media v2007.02.20)
```

```
RTSP/1.0 200 OK
Server: DSS/6.0.3 (Build/526.3; Platform/Linux; Release/Darwin Streaming Server; State/Development; )
Cseq: 12
Last-Modified: Fri, 20 Jun 2008 09:49:21 GMT
Cache-Control: must-revalidate
Content-length: 771
Date: Mon, 08 Sep 2008 10:30:28 GMT
Expires: Mon, 08 Sep 2008 10:30:28 GMT
Content-Type: application/sdp
x-Accept-Retransmit: our-retransmit
x-Accept-Dynamic-Rate: 1
Content-Base: rtsp://192.168.0.100/prueba.3gp/
```

•**SETUP**: Tras conocer los componentes del recurso, se pueden lanzar múltiples peticiones SETUP para los diferentes flujos que se deseen establecer. En el mensaje de SETUP además del recurso se indica el protocolo a utilizar (presentado por el servidor en la respuesta al DESCRIBE) junto a los puertos disponibles para dicho flujo, que serán confirmados por el servidor en la respuesta al mensaje. A continuación el ejemplo para un único flujo RTP:

```
SETUP rtsp://192.168.0.100/prueba.3gp/trackID=65536 RTSP/1.0
CSeq: 13
Transport: RTP/AVP;unicast;client_port=32844-32845
User-Agent: VLC media player (LIVE555 Streaming Media v2007.02.20)
```

```
RTSP/1.0 200 OK
Server: DSS/6.0.3 (Build/526.3; Platform/Linux; Release/Darwin Streaming Server; State/Development; )
Cseq: 13
Last-Modified: Fri, 20 Jun 2008 09:49:21 GMT
Cache-Control: must-revalidate
Session: 353322952731 1269409
Date: Mon, 08 Sep 2008 10:30:28 GMT
Expires: Mon, 08 Sep 2008 10:30:28 GMT
Transport: RTP/AVP;unicast;source=192.168.0.100;client_port=32844-32845;server_port=6970-6971;
ssrc=7FED1FCB
```

•**PLAY**: Una vez inicializados los flujos deseados se procede a la solicitud de PLAY. Nuevamente es una solicitud a nivel de recurso RTSP y no flujo RTP, con lo que un único mensaje de PLAY inicia la reproducción de todos los flujos establecidos. Además de para iniciar la reproducción del recurso por primera vez, el mensaje PLAY se utiliza para reanudarla tras una pausa. Adicionalmente incorpora información sobre el rango a reproducir en el campo *npt*, por defecto el rango comprende desde el instante actual (0 al inicio y X durante la reproducción) hasta el final del recurso aunque es posible limitar este rango tanto superior como inferiormente. Las peticiones de PLAY sufren un sistema de cola en el servidor de manera que son ejecutadas de una en una al finalizar la anterior si existiera. La sesión a la que hace referencia la solicitud también viene indicada tanto en la petición como en la respuesta que además incluye información del número de secuencia a transmitir para cada uno de los flujos.

PLAY rtsp://192.168.0.100/prueba.3gp/ RTSP/1.0
CSeq: 17
Session: 3533229527311269409
Range: npt=6.069-
User-Agent: VLC media player (LIVE555 Streaming Media v2007.02.20)

RTSP/1.0 200 OK
Server: DSS/6.0.3 (Build/526.3; Platform/Linux; Release/Darwin Streaming Server; State/Development;)
Cseq: 17
Session: 3533229527311269409
Range: npt=8.40840-1370.16833
RTP-Info:
url=rtsp://192.168.0.100/prueba.3gp/trackID=65536;seq=64243;rtptime=1152163183,url=rtsp://192.168.0.100/prueba.3gp/trackID=65537;seq=61665;rtptime=732693889

•**PAUSE**: De manera análoga a la solicitud PLAY, existe la petición PAUSE utilizada, como su nombre indica, para pausar la reproducción en una sesión. Al igual que en el caso anterior funciona a nivel de recurso y viene acompañada del identificador de sesión.

PLAY rtsp://192.168.0.100/prueba.3gp/ RTSP/1.0
CSeq: 17
Session: 3533229527311269409
Range: npt=6.069-
User-Agent: VLC media player (LIVE555 Streaming Media v2007.02.20)

RTSP/1.0 200 OK
Server: DSS/6.0.3 (Build/526.3; Platform/Linux; Release/Darwin Streaming Server; State/Development;)
Cseq: 18
Session: 3533229527311269409

• **TEARDOWN**: Utilizado para detener la transmisión de un recurso determinado, TEARDOWN libera todos los recursos relativos a dicha comunicación, invalidando cualquier petición realizada sobre dicha sesión. Para volver a lanzar mensajes de petición es necesario realizar un nuevo SETUP de sesión.

```
TEARDOWN rtsp://192.168.0.100/prueba.3gp/ RTSP/1.0
CSeq: 21
Session:  353322952731269409
User-Agent: VLC media player (LIVE555 Streaming Media
v2007.02.20)
```

```
RTSP/1.0 200 OK
Server: DSS/6.0.3 (Build/526.3; Platform/Linux; Release/Darwin Streaming Server; State/Development; )
Cseq: 261
Session: 353322952731269409
Connection: Close
```

El protocolo RTSP cuenta con una serie de cabeceras bien definidas que se pueden encontrar en la Tabla 3. Además soporta otras cabeceras adicionales (extensiones) que se introducen de forma experimental, son aquellas precedidas por el campo x- como se puede apreciar en el ejemplo anterior de DESCRIBE donde se encuentra:

```
x-Accept-Retransmit: our-retransmit
x-Accept-Dynamic-Rate: 1
```

Cabecera	Tipo	Soportado	Método
Accept	R	Opcional	Entidad
Accept-Encoding	R	Opcional	Entidad
Accept-Language	R	Opcional	Todos
Allow	r	Opcional	Todos
Authorization	R	Opcional	Todos
Bandwidth	R	Opcional	Todos
Blocksize	R	Opcional	Todos menos OPTIONS, TEARDOWN
Cache-Control	g	Opcional	SETUP
Conference	R	Opcional	SETUP
Connection	g	Obligatorio	Todos
Content-Base	e	Opcional	Entidad
Content-Encoding	e	Obligatorio	SET_PARAMETER
Content-Encoding	e	Obligatorio	DESCRIBE, ANNOUNCE
Content-Language	e	Obligatorio	DESCRIBE, ANNOUNCE
Content-Length	e	Obligatorio	SET_PARAMETER, ANNOUNCE
Content-Length	e	Obligatorio	Entidad
Content-Location	e	Opcional	Entidad
Content-Type	e	Obligatorio	SET_PARAMETER, ANNOUNCE
Content-Type	r	Obligatorio	Entidad
CSeq	g	Obligatorio	Todos
Date	g	Opcional	Todos
Expires	e	Opcional	DESCRIBE, ANNOUNCE
From	R	Opcional	Todos
If-Modified-Since	R	Opcional	DESCRIBE, SETUP
Last-Modified	e	Opcional	Entidad
Proxy-Authenticate			
Proxy-Require	R	Obligatorio	Todos
Public	r	Opcional	Todos
Range	R	Opcional	PLAY, PAUSE, RECORD
Range	r	Opcional	PLAY, PAUSE, RECORD
Referer	R	Opcional	Todos
Require	R	Obligatorio	Todos
Retry-After	r	Opcional	Todos
RTP-Info	r	Obligatorio	PLAY
Scale	Rr	Opcional	PLAY, RECORD
Session	Rr	Obligatorio	Todos menos OPTIONS, SETUP
Server	r	Opcional	Todos
Speed	Rr	Opcional	PLAY
Transport	Rr	Obligatorio	SETUP
Unsupported	r	Obligatorio	Todos
User-Agent	R	Opcional	Todos
Via	g	Opcional	Todos
WWW-Authenticate	r	Opcional	Todos

Tabla 3: Cabeceras de RTSP

3.7.3. Protocolo de transmisión en tiempo real RTP y RTCP

El protocolo de transmisión en tiempo real RTP Real-time Transport Protocol (IETF RFC 3550) [3] es el encargado de transmitir la información del vídeo propiamente dicho. Como su nombre indica RTP es utilizado en la transmisión en tiempo real y ofrece funcionalidades como marcas temporales que permiten sincronizar la información en recepción o identificación del tipo de carga transportada. Utiliza como protocolo de transporte UDP y pese a ser un protocolo pensado para transmitir información en tiempo real, no provee ningún tipo de calidad de servicio ni garantía. Cada flujo RTP tiene una fuente para el cliente, llamado SSRC o Synchronization Source, la cual es única y viene identificada en el paquete. Esta fuente suele ser la que provee el flujo final al cliente, es decir, la última que realiza alguna modificación sobre los contenidos RTP; generalmente se trata del servidor, pero en ocasiones, como se puede apreciar en Figura 5, puede tratarse de un mezclador. Además de SSRC se pueden encontrar CSRC o Contributing Sources y son cada uno de los elementos que inicialmente generaron un flujo RTP y fue sometido a un proceso a algún proceso intermedio antes de llegar al cliente. También son indicadores únicos que, al no estar presentes en todos los casos, forman parte opcionalmente en los mensajes RTP.

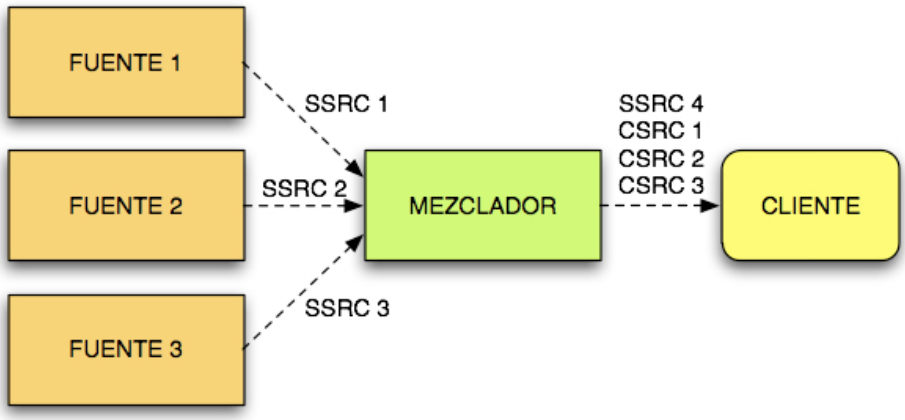

Figura 5: Cómo actúa un mezclador en RTP.

Tal y como puede verse en Figura 6, los campos posibles en un paquete RTP son:

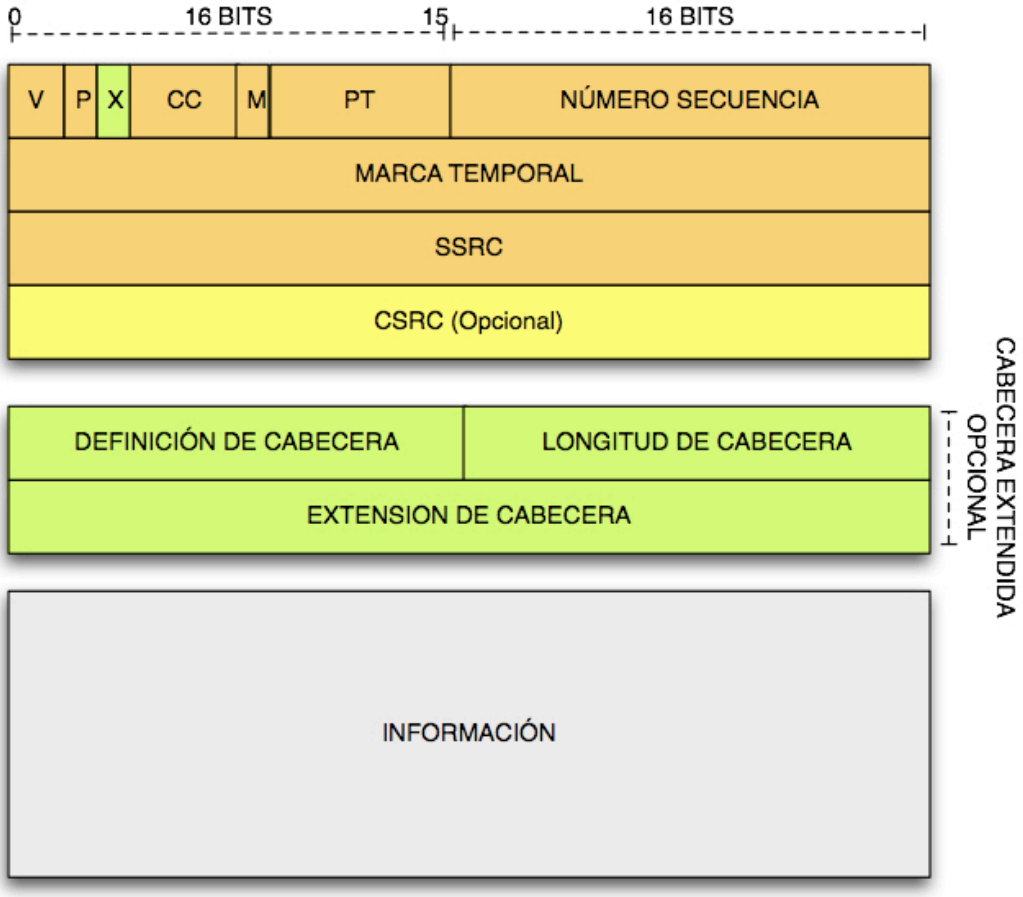

Figura 6: Estructura de un paquete RTP

•**V:** Versión de RTP utilizada; en este proyecto se considera la utilización de la versión 2 correspondiente al RFC 1889.

•**P:** Indica el uso de padding. El paquete RTP debería tener un múltiplo de 32 bits, en caso de no llegar a este número con los datos transportados se añadirán bits de relleno marcando en campo P a 1. El último Byte del paquete indica el número de octetos de relleno.

•**X:** Indica la existencia de una cabecera de extensión, es decir, al menos un bloque de 32 bits. Estos 32 bits se componen de 16 bits de definición de la cabecera y 16 bits indicando el número de palabras de 32 bits que siguen a continuación.

•**CC:** Contador de CSRC que indica el número de identificadores de este tipo adheridos a la cabecera, puede oscilar entre 0 y 15.

•**M:** Bit de marcado utilizado por aplicaciones específicas.

•**PT:** Tipo de datos contenidos en el paquete. Estos 7 bits presentan 128 opciones para indicar la codificación utilizada en los datos tal y como se aprecia en Tabla 4. Es importante destacar que, en caso de utilizar un valor dinámico, éste identifica a un flujo determinado. Como se verá en apartados

posteriores, es importante respetar el valor de PT en los cambios de flujo para que el cliente continúe la reproducción correctamente.

PT	Codec	Tipo	PT	Codec	Tipo	PT	Codec	Tipo
0	PCMU	A	14	MPA	A	28	nv	V
1	Reservado	A	15	G728	A	29	Sin asignar	V
2	Reservado	A	16	DVI4	A	30	Sin asignar	V
3	GSM	A	17	DVI4	A	31	H261	V
4	G723	A	18	G729	A	32	MPV	V
5	DVI4	A	19	Reservado	A	33	MP2T	V
6	DVI4	A	20	Sin asignar	A	34	H263	V
7	LPC	A	21	Sin asignar	A	35-71	Sin asignar	-
8	PCMA	A	22	Sin asignar	A	72-76	Reservado	N/D
9	G722	A	23	Sin asignar	A	77-95	Sin asignar	-
10	L16	A	24	Sin asignar	V	96-127	Dinámico	-
11	L16	A	25	CelB	V	din	G726-40	A
12	QCELP	A	26	JPEG	V	din	G726-32	A
13	CN	A	27	Sin asignar	V	din	G726-24	A

Tabla 4: Tipos de datos en paquetes RTP[23].

•**Número se secuencia:** Iniciado en un valor aleatorio e incrementado unitariamente, permite al receptor conocer los paquetes fuera de secuencia y realizar análisis de pérdidas.

•**Marca de tiempo:** Indica el tiempo de muestreo del primer octeto contenido en los datos, puede ser utilizado por el cliente para evaluar el *jitter* y mantener la sincronización de flujos. Incrementa de manera lineal.

•**SSRC:** Indica la fuente de sincronización para el cliente. Suele ser única y fija durante la transmisión. El valor utilizado también responde, inicialmente, a un número aleatorio.

Como se puede apreciar RTP transporta, además de los datos en sí, cierta información útil para realizar un cierto control de la comunicación, no obstante, carece de un mecanismo para informar del estado de la misma. Es por ello que trabaja junto a Real Time Control Protocol RTCP, el cual genera una serie de informes para controlar el estado de las transferencias gracias a que incluye información sobre los paquetes perdidos, el retardo, *jitter*, etcétera. Gracias a estos informes será posible conocer también el estado del buffer el cliente. RTCP se compone fundamentalmente de dos mensajes: los informes de emisor y receptor o SR (*Sender Report*) y RR (*Receiver Report*) respectivamente.

Estos informes son muy similares entre sí como se puede apreciar en la Figura 6 y Figura 7; Sus principales diferencias respecto a la cabecera de un paquete RTP son: La aparición del campo RC que supone la eliminación del bit de extensión, bit de marcado y CC. Se utiliza para indicar el número de

[23] Los tipos A/V corresponden a audio y vídeo respectivamente. Algunos códecs tienen un identificador por *bitrate*.

bloques de informe que contiene el paquete. La aparición de un campo de longitud en lugar del número de secuencia y la desaparición de la marca temporal.

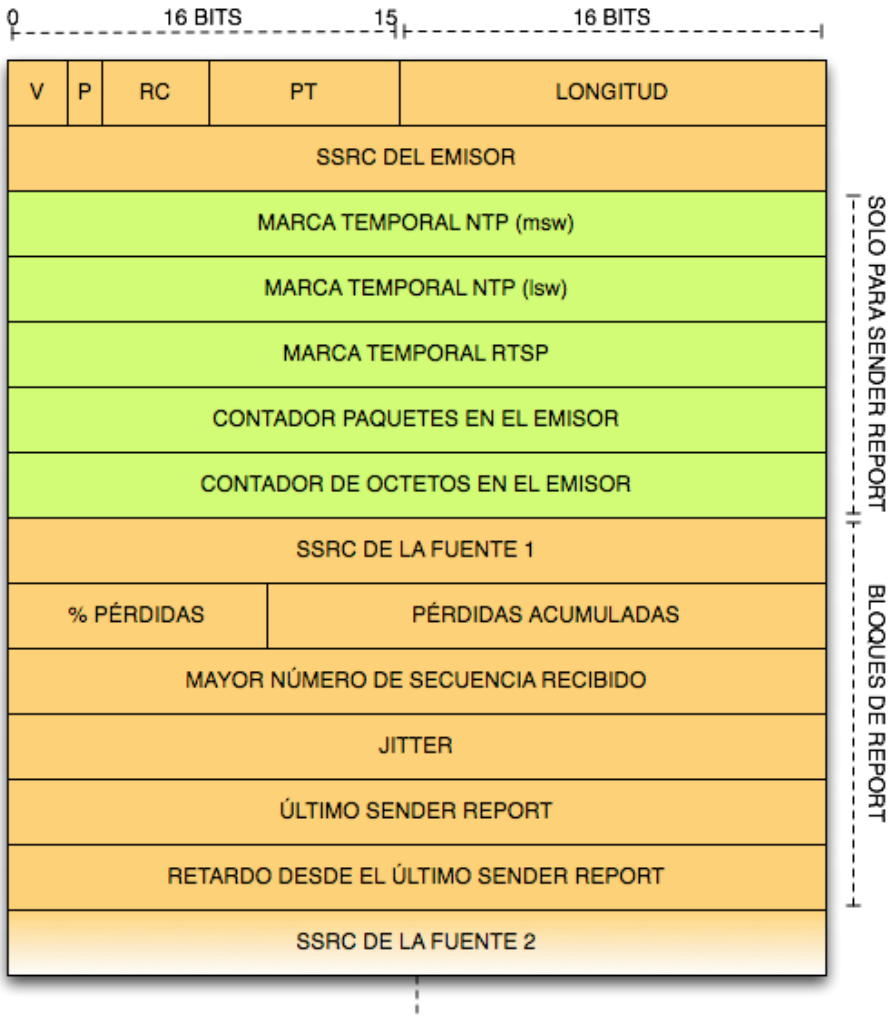

Figura 7: Estructura de los paquetes Sender Report y Receiver Report.

Además de la cabecera, en los informes de servidor se envía una serie de información sobre éste, tal y como se aprecia en la Figura 7[24]. Los campos no presentados anteriormente son:

- **Marca temporal NTP:** Indica el momento en el que se envió el informe. Puede ser interesante para el cliente ya que le ayuda a calcular el RTT[25].
- **Contador de paquetes RTP del servidor:** Indica el número de paquetes RTP enviados por el servidor hasta ese momento. Resulta útil para conocer cuál es el último

[24] Los campos en verde son exclusivos de los informes de servidor.

[25] Round Trip Time.

paquete enviado hasta el momento, así como para combinarlo con la información relativa a la marca temporal.

- **Contador de octetos RTP del servidor:** Análogo al campo anterior utilizando octetos en lugar de paquetes.

Además de la cabecera, los informes RTCP incluyen una serie de bloques de información, perteneciente a cada uno de los SSRC participantes en esa transmisión. La información contenida en dichos bloques, como se aprecia en la Figura 7 es bastante autoexplicativa y comprende desde fracciones y acumulados de pérdidas hasta contadores de SR y RR recibidos, pasando por cálculo de *jitter*.

Además de los mensajes comentados anteriormente existen otros como los informes SDES, que contienen información de las diferentes fuentes RTP. APP, propios de aplicaciones específicas y BYE, indicando el final de una participación. Para este proyecto resultan especialmente relevantes los informes propios de aplicación incluidos en paquetes NADU[26]. En la Figura 8 puede apreciarse la estructura de un paquete NADU.

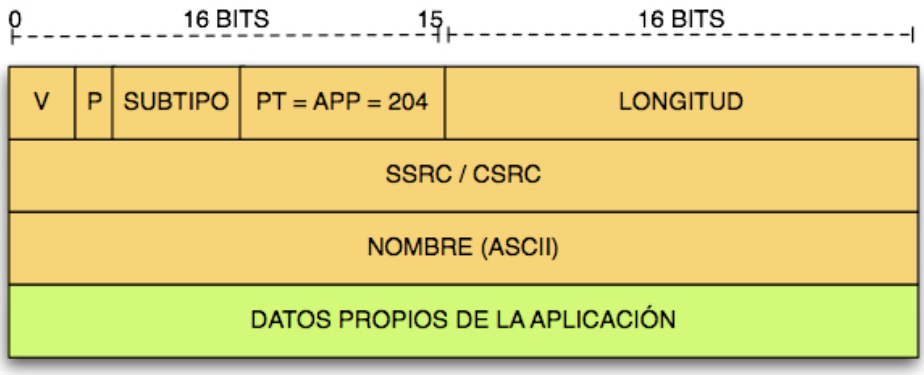

Figura 8: Estructura de los paquetes APP.

Este tipo de paquetes se utilizan para proporcionar información acerca del cliente sobre lo que se denominan Application Data Units o ADU, toda la información proporcionada por el cliente será relativa a dichas unidades. Éstas, como su nombre indica, son propias de la aplicación utilizada; en este caso variarán en función del tipo de datos según sea audio, vídeo y el tipo de codificación utilizada.

Los mensajes identifican por su nombre ASCII PSS0, su subtipo 0 y su tipo de *payload* 204. La longitud indica el número de palabras de 32 bits menos una que forman el paquete, también se incluye la información SSRC o CSRC. Respecto al campo de datos, en él se incluyen únicamente los campos presentados en la Figura 9.

Los campos más destacables de los datos son el SSRC, que indican el flujo al que referencia el cliente; el retardo que se define como la diferencia entre el tiempo de decodificación estimado del siguiente bloque de datos[27] y el instante de envío del siguiente paquete NADU, este campo da una

[26] Next Application Data Unit

[27] ADU

idea sobre la distancia temporal entre los datos almacenados y el instante para su reproducción. De esta forma el servidor puede calcular el tiempo restante hasta que se produzca una situación de *buffer* vacío. El campo NSN muestra el número de secuencia RTP de la siguiente ADU a decodificar perteneciente al SSRC. NUN indica el número de unidad de la siguiente ADU para decodificar, pueden existir múltiples ADUs en un paquete RTP. Finalmente se indica el espacio libre en buffer o FBS en bloques de 64 Bytes. Cabe la posibilidad de tener mayor espacio disponible que los 4194304 Bytes representables, en ese caso se utilizará el valor máximo. Esta variable junto al retardo serán las utilizadas en los algoritmos de adaptación de Darwin Streaming Server.

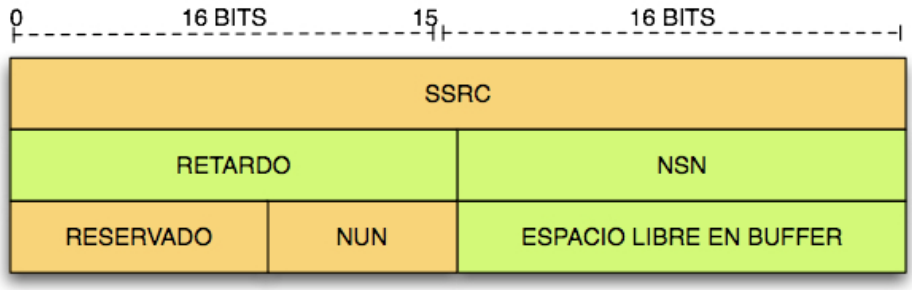

Figura 9: Estructura de datos en los paquetes NADU.

3.7.4. Protocolo de descripción de sesión SDP

Cuando un usuario accede a un enlace RTSP, lanza una petición DESCRIBE en la que el servidor le responde con una presentación de la información disponible. Dicha presentación se basa, generalmente, en el protocolo SDP. En SDP el servidor presenta los diferentes flujos disponibles, la codificación utilizada, el puerto asignado a cada flujo y el ancho de banda estimado de dicho flujo entre otros. A partir de esta información, el terminal decide cuáles son los flujos que descargará y lanza las peticiones a los puertos oportunos. Idealmente el terminal debería ser capaz de discriminar ofrecimientos en función del ancho de banda disponible, de la codificación utilizada, etcétera. En los casos reales no siempre se da este caso.

Además de anunciar características de los vídeos, SDP anuncia también algunas propiedades del servidor. En particular puede indicar si éste es capaz de soportar adaptación de tasa, funciones de Digital Rights Management DRM o ciertas propiedades de *buffering*, entre otros. En este caso, el servidor no soportará dicha adaptación de tasa, pero es necesario comentar que, en caso de implementarlo, debería anunciarse mediante SDP tal y como indican los estándares.

Un mensaje SDP se compone de información textual codificada en UTF-8 siguiendo la estructura <tipo>=<valor> donde el tipo debe ser bien conocido por ambos extremos y el valor debe seguir las indicaciones para cada tipo: generalmente se compone de valores separados por comas. Ambos campos son sensibles al uso de mayúsculas.

Descripción de sesión:
v= (versión de protocolo)
o= (identificador de sesión y origen)
s= (nombre de la sesión)
i= (información de la sesión)*
u= (URI de la descripción)*
e= (dirección de correo electrónico)*
p= (número de teléfono)*
c= (información de la conexión -- innecesaria si no se provee en todos los campos media)*
b= (información de ancho de banda)*
Descripciones de tiempo (ver tabla inferior)
z= (ajustes de zona horaria)*
k= (clave de encriptación)*
a= (atributos de la sesión)*

Dentro de la descripción que ofrece SDP se pueden diferenciar diversos bloques. El primer bloque, perteneciente a la descripción de la sesión incluye información sobre la versión de protocolo utilizada, el servidor que origina dicha información, el nombre de la sesión y cierta información de la conexión, incluyendo direcciones IP y ancho de banda estimado. También se incluyen información sobre criptografía y una serie de atributos de la sesión, entre los cuales se encuentra la duración de ésta, datos adicionales de control, información sobre las codificaciones utilizadas, etcétera.

Descripciones temporales
t= (tiempo que la sesión lleva activa)
r= (número de repeticiones)*

En segundo lugar se encuentra información relativa a la temporización de dicha sesión indicando principalmente el tiempo activo y el número de repeticiones a realizar.

Descripción de medios
m= (nombre y dirección de transporte del medio)
i= (título)*
c= (información de conexión si no se incluyó a nivel de sesión)*
b= (información de ancho de banda)*
k= (clave de encriptación)*
a= (atributos)*

Finalmente se encuentran una serie de bloques utilizados para descripción de medios[28]. En este bloque, cada elemento viene definido según su tipo, puerto, protocolo de transporte, formato, ancho

[28] Cada medio es, generalmente, una de las pistas del archivo.

de banda consumido, etcétera. Dentro de los atributos de cada medio se pueden encontrar características adicionales como información de la pista de *hint* correspondiente, el soporte de adaptación 3GPP, información textual de la codificación, tamaño del máximo cuadro del vídeo, grupo al que pertenece, etcétera. Toda esta información adicional puede ser utilizada por reproductores con capacidades adicionales. De este modo un reproductor "inteligente" podría discernir entre la elección de dos pistas alternativas ya sea por calidad, formatos soportados, ancho de banda consumido, etcétera. Cabe destacar en este apartado que la mayoría de terminales móviles no realizan esa distinción y suelen escoger el primer grupo disponible. A continuación se detalla la estructura de los principales atributos alternativos relevantes para este proyecto.

Atributos adicionales
- *a=alt:<id>:<línea SDP> (nombre y dirección de transporte del medio)*
- *a=alt-default-id:<id> (grupo por defecto)*
- *a=alt-group (información de grupo alternativo)*
- *a=3GPP-Adaptation-Support (información para adaptación de tasa disponible)*
- *a=3GPP-QoE-Metrics (información QoE disponible)*

3.8. Flujo de funcionamiento del servidor

Este apartado pretende presentar el proceso que se lleva a cabo a la hora de establecerse una transferencia de vídeo en tiempo real desde el servidor al cliente. Se ha adaptado el proceso al marco que nos ocupa en el que existe un servidor web que ofrece los recursos RTSP a los clientes. Fundamentalmente se trata de establecer una conexión entre el terminal y el punto de acceso, el cual dirigirá la petición del cliente hacia el servidor de *streaming*, momento en el cual se iniciará el diálogo para la transferencia de la información. Se comentan los pasos a continuación:

3.8.1. Conexión del terminal móvil

El primer paso a llevar a cabo para establecer la comunicación de vídeo es lograr que el terminal conecte al punto de acceso inalámbrico provisto en la plataforma. Para ello es necesario seleccionar el punto de acceso en el móvil. Se considera durante todo este proceso que el punto de acceso trabaja en un canal aceptable para los terminales (entre 1 y 11) ya que éstos son sensibles a esta selección. También es recomendable que la identidad de la estación sea distribuida para informar a los posibles clientes (SSID *broadcast*) y, en caso de establecer un sistema de protección (WPA/2, WEP...), informar a los potenciales usuarios para que puedan realizar la configuración oportuna. También se le debe proveer de la información IP necesaria (dirección, puerta de enlace, servidores DNS...), aunque la mayoría de terminales móviles trabajan únicamente con direcciones dinámicas.

El punto de acceso debe proveer conectividad al servidor de *streaming*, el cual utilizará por defecto el puerto 554 aunque es configurable. Además de dicho puerto, se utilizan otros puertos pares no "bien conocidos" para la transmisión de los diferentes flujos. Estos puertos también son configurables tanto en terminales como en servidor y son diferentes para cada comunicación. Es importante notar también, que ciertos reproductores móviles, en particular RealPlayer en Nokia, necesitan configurar el punto de acceso wifi al que acceden por defecto, una mala configuración supone conectividad de servicios como web, pero errores a la hora de realizar transmisiones de vídeo.

3.8.2. Acceso a recurso web

Una vez establecida la conexión inalámbrica, el cliente accederá al recurso web que desee. Actualmente la plataforma tiene implementado un sistema de redirección que envía al usuario a la página que se desee tras realizar la identificación de terminal independientemente de la URL que seleccione. A partir de ese momento el usuario inicia la navegación web con normalidad. Se entiende que, en un momento dado, el usuario deseará acceder a un vídeo en tiempo real con lo que deberá serle indicado mediante una URL a dicho recurso. El formato de la URL es el siguiente:

RTSP://IP_SERVIDOR:PUERTO_RTSP/RECURSO

Donde IP_SERVIDOR indica la dirección donde está ejecutándose el servidor. PUERTO_RTSP indica el puerto en el que está trabajando (por defecto 554) y RECURSO es el archivo al que se desea acceder, normalmente un archivo .3gp .mp4 o .sdp. Existe la posibilidad de servir el vídeo

directamente sin acceder previamente a una página HTML, aunque se entiende que es el usuario quien, a priori, debería seleccionar el vídeo que desea visualizar.

3.8.3. Diálogo entre cliente y servidor

Una vez que el usuario accede al recurso rtsp, si su terminal es capaz de entender el protocolo, iniciará automáticamente la ejecución del reproductor multimedia del terminal e iniciará el diálogo con el servidor. Los principales mensajes intercambiados en una comunicación típica pueden apreciarse en el siguiente gráfico:

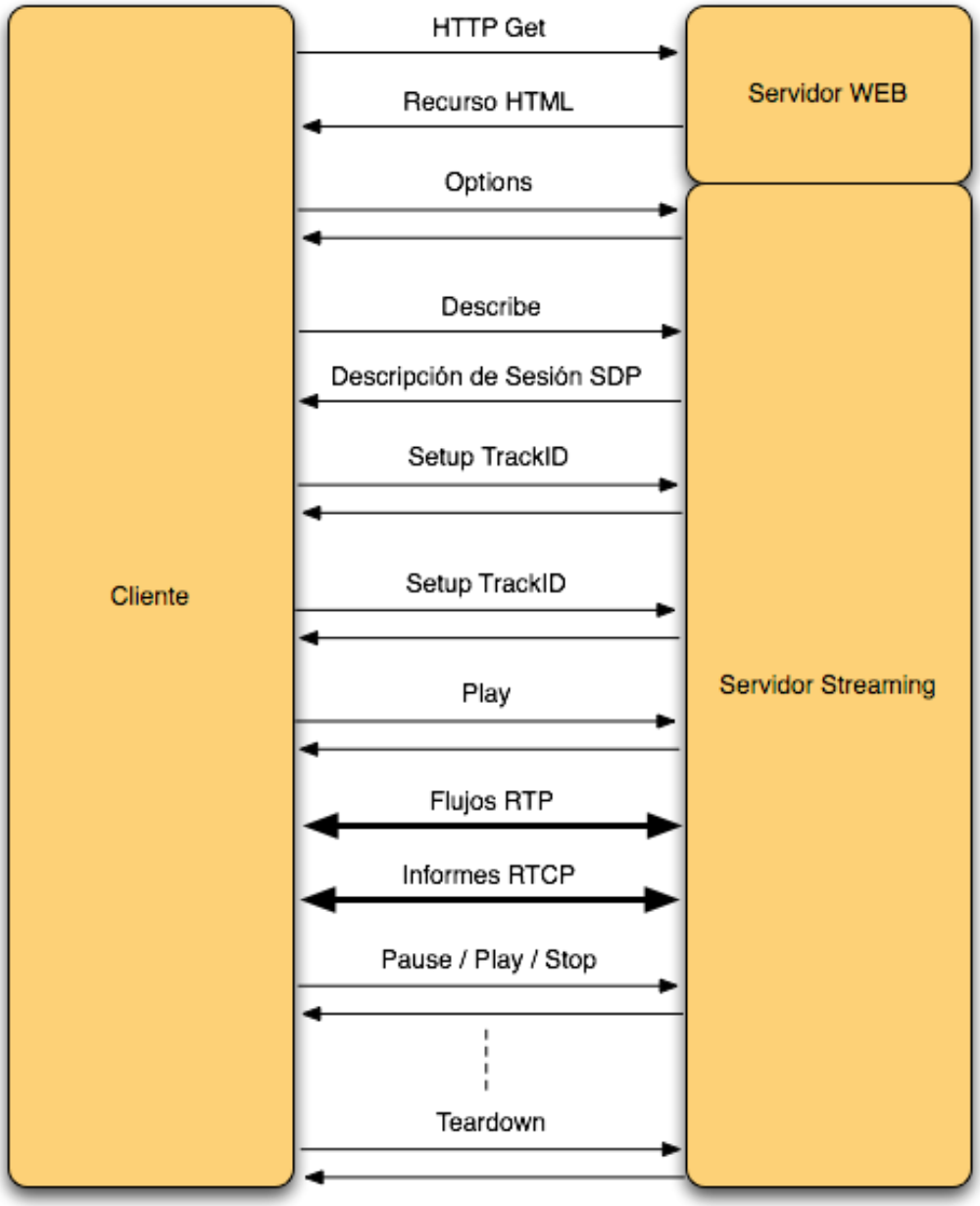

Figura 10: Intercambio de mensajes en una comunicación RTSP

Inicialmente se ve como el cliente se conecta al servidor web de la plataforma al que realiza una petición GET de HTML, que le proporcionará una página donde estén disponibles los enlaces a los diferentes vídeos. En este caso será el usuario el que escogerá el enlace que le resulte más interesante, aunque existe la posibilidad de ofrecer el *streaming* de vídeo embebido en la página HTML.

Una vez seleccionado el recurso a través del enlace RTSP, el terminal envía un mensaje OPTIONS sobre RTSP el cual le permitirá conocer qué mensajes (y por lo tanto funcionalidades RTSP) soporta el servidor. En este caso particular la respuesta incluirá las opciones DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, OPTIONS, ANNOUNCE y RECORD .

Conocidas las funcionalidades, el siguiente paso es enviar un mensaje de DESCRIBE sobre el recurso que se ha solicitado en el enlace. Un ejemplo de mensaje podría ser:

```
DESCRIBE rtsp://ip_servidor:puerto_rtsp/archivo.3gp RTSP/1.0
```

Durante este intercambio de mensajes el terminal informa al servidor del modelo del que se trata, el reproductor que utiliza y del ancho de banda que teóricamente soporta.

A partir de este momento el servidor analiza el recurso y enviará la descripción del mismo siguiendo el protocolo de descripción SDP. En esta descripción se incluye información sobre el archivo al que se desea acceder. En ella se informa fundamentalmente de las diferentes pistas que posee el archivo, la duración, su tipo (audio, vídeo, *hint*), la codificación utilizada, el ancho de banda que se necesita para descargar cada una de ellas y el identificador al que se debe acceder en cada caso, además se informa de las agrupaciones en pistas alternativas y las capacidades de adaptación del servidor. Idealmente el terminal debe tener en cuenta los diferentes parámetros de los que le informa el servidor, asegurarse de que soporta las codificaciones, dispone del ancho de banda, etcétera y lanzar las peticiones (SETUP) a los flujos que le interesen. En los casos reales pocas veces se da esta circunstancia ya que, aunque los reproductores tienen claro cuáles son los códecs de los que disponen (y por lo tanto rechazan otras codificaciones), los terminales no suelen estar bien informados del ancho de banda del que disponen. En particular, los terminales Nokia, aceptan conexiones que no superen el ancho de banda que tienen configurado como máximo en su reproductor (por defecto ~300Kbps), independientemente de las conexiones de la red. Del mismo modo, al presentarle diferentes codificaciones de un mismo archivo, el terminal debería escoger una de ellas utilizando un cierto criterio, idealmente se trataría de aquella que se ajusta a sus capacidades y condiciones, se verá como esto no es así. Otros clientes como VLC o QuickTime seleccionan todas las pistas anunciadas por el servidor para iniciar una reproducción conjunta de todas ellas, evidentemente este escenario no es en absoluto deseable.

Dado que la práctica totalidad de los terminales no realiza ninguna discriminación sobre las pistas de las que informa el servidor (incluso siendo compatibles con 3GP-Rel6), se llega a la conclusión de que resultaría interesante un servidor que tomara la decisión de incluir en el DESCRIBE sólo aquellas pistas que puedan interesar al cliente a partir de la información que conoce de éste.

Cuando el terminal ha decidido cuáles son los flujos a los que quiere/puede acceder, lanza una serie de peticiones SETUP, una para cada flujo de audio o vídeo. Típicamente se lanzan dos peticiones donde cada una de ellas tiene el siguiente aspecto:

SETUP rtsp://ip_servidor:puerto/archivo.3gp/trackID=65536 RTSP/1.0

Cada trackID pertenece a un flujo diferente, en este caso se podría tener el vídeo identificado como 65536 y el audio como 65537. En la petición el terminal incluye los puertos por los que está dispuesto a establecer la comunicación. Uno de ellos se utilizará para el flujo RTP de dicha pista y el otro para el control RTCP. Por lo tanto, si se solicitan dos flujos, se tendrán cuatro puertos dedicados a la comunicación.

A cada una de las peticiones de SETUP el servidor responde confirmando la solicitud e incluyendo, entre otra información, los puertos que él utilizara para dichos flujos. Una vez conocida esta información, la relación entre flujos y puertos queda ligada, con lo que el usuario será capaz de lanzar peticiones RTSP al servidor en su puerto común (554 por defecto) y éste responderá ejecutando la petición en los puertos concretados anteriormente. En particular, la transmisión iniciará con una petición PLAY del cliente al servidor el cual iniciará la transmisión de información desde sus puertos negociados, a los puertos definidos por el cliente.

Es importante destacar que todas las pistas de las que el cliente solicita un SETUP quedan asignadas dentro de su sesión con lo que los mensajes RTSP del tipo PLAY, PAUSE o TEARDOWN afectan a todas ellas por igual. A partir de ese momento el cliente espera información de todas ellas es decir, no se puede realizar un SETUP de cuatro pistas y reproducir únicamente dos ya que el terminal espera recibir los cuatro flujos y que los mensaje RTSP de confirmación incluyan información sobre ellos.

Mientras la transmisión se lleva a cabo, tanto el cliente como el servidor van enviando informes de su experiencia mediante el protocolo RTCP, es lo que se conocen como Sender Reports y Receiver Reports. Estos informes son interesantes para conocer el estado de la comunicación en los extremos, pudiendo resultar útiles para implementar técnicas de adaptación de velocidades y flujos ya que permiten conocer los paquetes recibidos, el retardo, el estado del buffer (aplicando ciertos algoritmos), etcétera. Son mensajes que se transmiten periódicamente pero a una frecuencia relativamente baja.

Durante la transmisión de información vía RTP, el cliente puede lanzar otras peticiones RTSP, tales como pausar, reproducir o detener la transmisión. Una vez que la reproducción ha finalizado se envía un mensaje de liberación TEARDOWN. A partir de ese momento los recursos quedan liberados y el servidor considera el cliente desconectado.

4. El servidor de *streaming*

4.1. Introducción

En este apartado se presentan algunas de las alternativas principales en materia de servidores de *streaming* aptos para el proyecto. Se presentan las principales características de cada plataforma con sus pros y contras y se argumenta la selección realizada. Debe considerarse que las versiones analizadas son aquellas disponibles en el momento del inicio del proyecto.

Posteriormente se explica el funcionamiento básico del servidor de *streaming* seleccionado. Su instalación, configuración, capacidades y alternativas de funcionamiento. Este punto pretende ser un pequeño manual descriptivo de las posibilidades iniciales del servidor desde el punto de vista de usuario.

4.2. Selección de un servidor de *streaming*

Como se ha comentado anteriormente, los requisitos fundamentales a la hora de escoger un servidor son: Soporte de archivos 3GP, soporte de *streaming* con funcionalidades RTSP, utilización de software abierto y gratuito y compatibilidad con las plataformas *InfoPoints* de Futurlink. Se apreciará especialmente aquella solución compatible con las últimas versiones 3GPP y que proporcione servicios de valor añadido como listas de reproducción, adaptación de tasa, multicast, etcétera. Así como aquella que aporte mayor flexibilidad para el desarrollo, goce de mayor documentación y presente mayores alternativas para la implementación del proyecto.

Una vez conocidos los requisitos fundamentales, a la hora de seleccionar el servidor se han barajado las siguientes alternativas:

4.2.1. Darwin Streaming Server: DSS

Servidor desarrollado por Apple[29] y distribuido bajo licencia Apple Public Source License[30]. En el inicio del desarrollo del proyecto se encontraba disponible versión 5.5.5 de Linux. Las principales características de ésta son:

- Se trata de una versión de código abierto de Quicktime Streaming Server.
- Es capaz de trabajar como servidor de archivos MP3, MP4 y 3GP *hinted*.
- Soporta RTP/RTCP, RTSP, SDP...

[29] www.apple.com/es

[30] www.opensource.apple.com/apsl/

•Permite creación listas reproducción mediante diferentes algoritmos como bucle continuo, basada en pesos, etcétera.

•Permite utilizar un sistema de *relays* para obtener o servir flujos de/a otra fuente o servidor.

•Soporta intercambio de capacidades mediante mensajes SDP aunque no las procesa ni, por lo tanto, ejecuta ningún tipo de adaptación.

•No soporta adaptación de tasa en tiempo real aunque se anuncie de manera experimental mediante el intercambio de cabeceras del mismo tipo.

•Posee una interfaz gráfica vía web para configuración y monitorización de las principales características aunque pueden ser gestionadas de manera manual:

Monitorización: Nombre servidor, status, tiempo activo, carga CPU, número conexiones, *throughput* actual, bytes servidos, conexiones servidas, usuarios conectados actualmente (IP, *bitrate*, Bytes enviados, porcentaje de pérdidas, tiempo activa, recurso solicitado).

Configuración: Directorio archivos, puerto RTSP, número máximo de conexiones simultáneas soportadas, *throughput* máximo soportado, método de autenticación, control de usuarios, preferencias en la generación de logs, creación de listas de reproducción...

Logs: Monitorización de actividad reciente, errores, acceso a archivos...

Contiene varios archivos de configuración modificables por el usuario que permite adaptar valores de *buffers*, tasas máximas de transmisión, listas de acceso etc..

Algunas de las principales ventajas del sistema:

• Robusto: 5ª gran versión del servidor. Se comporta de manera muy estable en las pruebas realizadas en las plataformas.

• Sencillo: No necesita ningún tipo de configuración adicional por parte del cliente. Se instala, ejecuta y se almacenan los archivos en las carpetas correspondientes.

• Soportado: Cuenta con un sistema de actualizaciones periódicas. Soporta los principales terminales, protocolos y tipos de archivos relacionados con PSS.

• Soporte para sistema *relay* y listas de reproducción: Interesante para configuraciones de múltiples puntos de acceso, despliegues tipo *mesh* y sistemas de distribución de contenidos.

• Módulos programables: Presenta una estructura modular para implementar módulos externos que no afecten al funcionamiento del núcleo, la idea es permitir que estos módulos puedan ser compatibles con versiones posteriores.

Algunas desventajas del sistema:

• Modificación compleja: Desventaja común a la mayoría de servidores de *streaming*. DSS cuenta con un extenso sistema de roles, módulos, estructuras y funciones que debe ser analizado antes de poder realizar algún tipo de modificación.

• No implementa adaptación de tasa ni descubrimiento de capacidades: En la versión analizada se contemplaba el intercambio de mensajes con el perfil WAP del cliente pero no se realizaba ninguna

acción al respecto, en cuanto a adaptación de tasa se presentaban una serie de cabeceras experimentales que no eran interpretadas por los clientes, únicamente poseía un algoritmo de adaptación para clientes específicos como QuickTime, los cuales utilizaban el protocolo propietario *Reliable UDP*.

• Los archivos 3GP tienen que tener pistas *hint*: Una desventaja menor ya que el proceso de *hinting* es trivial con cualquiera de las herramientas disponibles como MP4Box de Linux.

• Estadísticas poco fiables (CPU, throughput...). Otro de los problemas comunes de los servidores es la imprecisión de las estadísticas mostradas.

4.2.2. Catrastreaming (Open Streaming Server)

Proyecto Open Source disponible en SourceForge[31] publicado bajo General Public License[32]. Se ha trabajado con la versión 1.7 para Linux. Sus principales características son:

• Servidor de código abierto.

• Soporta los principales tipos de archivo y protocolos como RTP, RTSP, SDP.

• Compatible con archivos 3GP *hinted* o no, realiza el proceso de manera local.

• No permite creación listas reproducción.

• No soporta intercambio de capacidades (al menos directamente).

• No soporta adaptación de tasa en tiempo real.

• Permite realizar una petición http en cada intento de conexión de usuario. Inicialmente se plantea para autorizar o no el acceso de los mismos mediante un *servlet* (ej, tener una lista de direcciones IP autorizadas). El sistema permite pasar algunos parámetros del cliente como IP, url solicitada, mensaje... Podría utilizarse para llevar un control del tipo de usuarios conectados al sistema (x usuarios a tasa r, y usuarios a tasa s, z usuarios a tasa t). Esto no solucionaría el problema de movilidad. Del mismo modo permite enviar peticiones http una vez desconectado. Se puede utilizar para "liberar espacio".

• Teóricamente permite pasar parámetros adicionales en las peticiones de conexión. En concreto se supone que una petición del tipo rtsp://IP:PUERTO/RECURSO?StartTimeInSecs=<tiempo> debería solicitar la reproducción del recurso a partir de tiempo solicitado. El sistema no funciona, obteniendo como resultado en el servidor un error de parámetro desconocido y el vídeo inicia su reproducción de manera habitual (desde el segundo 0). Este método podría ser efectivo para implementar una solución intermedia entre el cliente y servidor que solicitara la reproducción de un vídeo de diferente calidad en un momento determinado a partir del instante en el que las condiciones del cliente cambiaron. Ej: el cliente solicita video_300kbps.3gp y en el segundo 15 su *throughput* se reduce a 200kbps. En ese momento se podría solicitar el recurso video_150kbps.3gp?StartTimeInSecs=15.

[31] sourceforge.net

[32] http://www.gnu.org/copyleft/gpl.html

• No tiene GUI aunque permite conexión por servidor http y páginas XML: Resulta incómodo ya que debe hacerse una petición http para cada consulta. El mejor sistema es generar una pequeña página HTML con los enlaces directos a las peticiones más utilizadas.

• El sistema de monitorización es aún más dudoso que en DSS. Algunos parámetros tardan mucho en refrescarse. Posible problema a la hora de implementar una solución basándose en los datos devueltos por este sistema.

• Configuración: Existe la posibilidad de modificar ciertos parámetros de configuración a través de peticiones http. Solo útil para usuarios remotos. Logs: Monitorización de actividad reciente, errores, acceso a archivos... similar a DSS. Al igual que DSS contiene un archivo de configuración modificable por el usuario (CatraStreamingServer.conf) que permite adaptar valores de *bufferes*, tasas máximas de transmisión, listas de acceso, etc...

Ventajas del sistema:

• Abierto.

• Configurable.

• Soporta todo tipo de 3gp.

Desventajas del sistema:

• Sin soporte (el responsable del proyecto trabaja en servidores cerrados).

• En cierto tiempo será "*deprecated*".

• Problemas con la instalación (problemas para registrarlo en el sistema).

• Sin interfaz.

• Parece ser menos robusto que DSS.

• Estadísticas poco fiables.

4.2.3. Helix DNA Server

Helix DNA Server es una versión gratuita de Helix Server disponible en su página de proyecto[33]. Es una iniciativa del grupo Real, se distribuye bajo General Public License y se ha trabajado con la versión de Linux 11.1. Principales características:

• Servidor open source

• Soporta RTP, RTSP, SDP...

• No soporta archivos 3gp. Únicamente .rm .ra. rv. mp3 de manera nativa.

[33] www.helixcommunity.org

• Anuncia compatibilidad con 3GP Release 6 (incluye entre otras, adaptación de tasa en tiempo real).

• Helix DNA server es la versión gratuita de Helix universal Server (privado) y se plantea como la alternativa open source desde la que construir un servidor a medida. Lamentablemente No tiene soporte 3GP y parece ser que no hay mucho interés en que esto cambie. En los foros de helix community no hay información al respecto.

• Para utilizarlo con otros formatos sería necesario implementar un paquetizador para cada cada uno de ellos.

• Tiene GUI para monitorizar y configurar el servidor. Al no cumplir de momento con el requisito de soportar archivos 3GP no se han realizado excesivas pruebas salvo comprobación de funcionamiento y stream de archivos mp3 a terminales fijos con resultado correcto.

4.2.4. VLC:

VLC es un reproductor multimedia de código abierto, multiplataforma que permite la reproducción y distribución de contenidos bajo demanda utilizando los protocolos RTP/RTCP y RTSP y archivos 3GP. Para ello se define un archivo que se utilizará como referencia en la ejecución del programa. Este archivo contiene una serie de recursos que apuntan a un archivo local y se definen como VoD enabled. Ejemplo:

```
new archivo.3gp vod enabled
setup archivo.3gp input "/usr/local/movies/archivo.3gp"
```

Aquí el recurso archivo.3gp apunta a un archivo con el mismo nombre y se define como Vod. Se ha utilizado el mismo nombre para el recurso que para el archivo para así no tener que modificar los enlaces. Si el cliente realiza una petición RTSP://IP:PUERTO/RECURSO accederá al archivo indicado.

Pese a que VLC es una gran herramienta para la transcodificación o transmisión multicast no demuestra ser robusto ni cómodo para la transmisión de vídeo bajo demanda. Pese a que no es una opción interesante para el servidor principal, sí puede ser utilizado como transcodificador o incluso como servidor alternativo. Se presentará alguna de sus posibilidades en la sección de relays.

4.2.5. Otros Servidores

4.2.5.1. Helix server:

Helix server es la versión comercial de Helix DNA presentado en el apartado anterior. Entre sus características principales se incluye la de ser compatible con la Release 6 de 3GPP permitiendo, entre otros, adaptación de tasa en el servidor. Además provee toda un paquete de software que se puede utilizar, entre otras cosas, para realizar la codificación del material. No se ha considerado al no tratarse de un sistema gratuito.

4.2.5.2. Quicktime Server:

Del mismo modo que Helix Server es la versión comercial de Helix DNA, Quicktime Streaming Server (QTSS) es la versión comercial de Darwin Streaming Server. Contempla todas las características propias de este tipo de servidores pero también se trata de una versión no gratuita. De hecho solo se distribuye con una versión server del sistema operativo Mac OS X.

4.3. Instalación y funcionamiento

4.3.1. Instalación

Darwin Streaming Server está disponible en dos versiones, código fuente y precompilada. La versión de código fuente nos permite realizar modificaciones en el código o programar nuevos módulos con el fin de añadir nuevas funcionalidades, una guía completa de como programar para este servidor se puede encontrar en [17]. En ella se encuentran además instrucciones para compilar los programas correctamente. En ambos casos una vez compilados simplemente hay que ejecutar el instalador. En el proceso de instalación se crea un usuario QTSS pensado para ejecutar el servidor de manera segura ya que no cuenta con permisos de administrador. Además se solicita la introducción de un nombre de usuario y contraseña para la gestión mediante interfaz gráfica.

Durante la instalación se crean los siguientes directorios por defecto:

- /etc/streaming:

En él se encuentran los principales archivos de configuración del sistema. Destacan los utilizados para configurar el servidor *streamingserver.xml* y el del sistema de relays *relayconfig.xml*. Además se encuentra información sobre grupos y usuarios que tienen acceso al servidor y configuración para el cliente de *streaming* que provee el sistema.

- /usr/local/movies:

Directorio donde se almacenará la información multimedia por defecto. Es el directorio por defecto en el que el servidor busca los elementos que puede servir, ya sean vídeos o audio. En él deben encontrarse los archivos MP3 y vídeos preparados (empaquetados en 3GP y *hinted*) con permisos de acceso para el usuario que ejecute el servidor.

- /var/streaming:

Aquí se pueden encontrar principalmente las carpetas de *logs* y *playlists* donde es posible configurar éstas de forma manual o automática. Se explicará más detalladamente como realizar la configuración en apartados posteriores.

4.3.2. La interfaz gráfica

Una vez instalado correctamente el servidor es posible ejecutarlo utilizando dos comandos diferentes. En caso de desear acceder a la interfaz gráfica, se debe ejecutar:

- streamingadmin.py

Con este comando además de iniciar el servidor como tal, se activará el acceso mediante interfaz gráfica web, accesible en el puerto 1220 (por defecto) de la IP local. Para acceder es necesario incluir el nombre de usuario y contraseña especificados en el proceso de instalación. Es posible configurar el sistema para acceder al interfaz mediante SSL pero es preciso tener configurado un sistema de emisión de certificados (Cert. Authority, OpenSSL...).

En la pantalla principal del servidor existe información básica de éste, incluyendo:

- Nombre de la máquina.
- Tiempo activo.
- Estado.
- Versiones.
- Carga de CPU.
- Conexiones actuales / totales.
- Throughput actual / total.

Esta información no se refresca automáticamente, por lo que es preciso refrescar la página o acceder de nuevo para tener la información actualizada.

En el menú lateral se puede acceder a dos páginas con estadísticas: Connected Users y Relay Status. En ellas se puede ver información actualizada tanto de los usuarios conectados (dirección IP, tiempo conectado, Bytes transmitidos, recurso al que accede...) como del estado de los *relays* (origen, destino, tiempo, Bytes transmitidos...), en ambos casos se puede definir el periodo de refresco manualmente.

En el siguiente grupo de opciones se pueden realizar los diferentes ajustes del servidor:

- General Settings: Permite configurar el directorio donde se almacena la información que se puede transmitir. Es importante recordar que la modificación de este directorio implica dotarlo de permisos para el usuario que ejecute el servidor, de lo contrario no será capaz de servir el material. Por otra parte es posible limitar el número de usuarios y el througput. Este sistema es interesante en caso de tener una red cableada bien dimensionada con un throughput máximo fijo y relativamente estable. Si un usuario termina de visualizar un vídeo y no cierra el reproductor sigue siendo considerado como un usuario conectado. También se puede modificar el tipo de autenticación y las contraseñas.

- Port Settings: Permite habilitar el *streaming* de vídeo utilizando un sistema de túnel a través del puerto 80. Sólo es recomendable activarlo en casos concretos en los que el servidor se encuentra tras sistemas de filtrado o NAT. En cualquier otro caso entraría en conflicto con el servidor web.

• Relay Settings: En este apartado se puede configurar un sistema de *relays* particular. Se verá más información en el apartado dedicado a ellos.

• Log Settings: Utilizado para configurar el sistema de almacenamiento de eventos.

A continuación existe una sección Playlists donde se pueden configurar las listas de reproducción del servidor. Se dedicará un apartado exclusivo a éstas.

Finalmente aparecen enlaces a los registros más recientes, al histórico de accesos el cual lleva una contabilidad del número de veces que se ha solicitado cada elemento y un enlace a logout para salir del administrador. Además del menú lateral de la izquierda existe una pestaña superior que permite habilitar o deshabilitar el servidor de *streaming*.

Todas las configuraciones que se realicen mediante interfaz gráfico, pueden ser llevadas a cabo de manera manual editando los diferentes archivos comentados anteriormente. Si se desea ejecutar el servidor obviando la interfaz gráfica se debe ejecutar:

```
DarwinStreamingServer
```

El servidor permite ejecutarse en primer plano, mostrando información de debug, etcétera. Para ello solo es necesario ejecutarlo con los parámetros adecuados (-D, -d...) puede utilizarse DarwinStreamingServer -h para conocerlos.

En ambos casos se ejecutarán al menos dos procesos llamados DarwinStreamingServer, en caso de tener interfaz también se ejecutará streamingadmin y en caso de tener listas de reproducción PlaylistBroadcaster. Si se quiere detener la ejecución del servidor se pueden matar los procesos DarwinStreamingServer con kill -9 y el identificador de proceso. En caso de ejecutarse en primer plano es suficiente con un ctrl+c.

4.3.3. Listas de reproducción

Una de las principales características del servidor es la capacidad de configurar listas de reproducción. Una lista de reproducción es un conjunto de archivos de la misma clase (todos de audio o todos de vídeo) que se reproducen de manera automática en el servidor. De este modo cuando un usuario accede al recurso de dicha lista, reproduce los contenidos tal y como fueron programados por el administrador.

En el momento de programar la lista, existen disponibles tres modos de reproducción:

• Sequential Looped: Los archivos seleccionados se reproducen en orden de manera continua.

• Sequential: Los archivos se reproducen en orden.

• Weighted Random: Los archivos se reproducen de manera aleatoria según el peso que se les asigne. También se realiza de forma continua.

Cada lista de reproducción tiene asignado un nombre y un punto de montaje. El nombre es un valor meramente identificativo, aunque dependiendo de él se creará un directorio u otro dentro de la carpeta playlists. El punto de montaje es el recurso al que se deberá acceder para ejecutar la lista de reproducción. Normalmente este punto de montaje tiene una extensión .sdp. Ejemplo:

Lista de reproducción.
Nombre: Lista1
Punto de montaje lista1.sdp
Recurso para acceder rtsp://ip_servidor:puerto/lista1.sdp

Una lista de reproducción suele tener como destino la dirección IP local, es decir, se lanza la lista contra el propio servidor para posteriormente reflejarla a los clientes conectados, esto se realiza gracias a un módulo llamado *Reflector*. Este módulo, al detectar la adhesión de un cliente, empaqueta la información lanzada por la lista de reproducción de una forma compatible con el cliente, es decir, utilizando paquetes RTP bien formados en términos de número de secuencia, marcas temporales, etcétera.

El sistema de listas envía la misma información a todos los usuarios conectados a ella, es decir, todos los usuarios que conectan a una misma lista reciben la misma información, del mismo modo que si se tratase de una multidifusión de vídeo (DVB). Lógicamente los usuarios no tienen capacidades de interacción con la lista de reproducción salvo reproducir y detenerla localmente. Pese a que la información transmitida es la misma, cada usuario tiene un flujo exclusivo de datos, con lo cual no se reduce el tráfico en la red. Los archivos pertenecientes a la lista se reproducen de manera continua, sin pausas, de manera que resulta transparente para el usuario.

Las listas de reproducción son lanzadas por un nuevo proceso llamado PlaylistBroadcaster, éste se ejecuta de manera automática si se configura la lista de reproducción desde GUI mientras que requiere ser ejecutado manualmente si se genera la lista de reproducción desde consola. Veamos ambos métodos para construir nuestra lista:

4.3.4. Utilizando la interfaz gráfica

Para generar una lista de reproducción utilizando la GUI simplemente se deben seguir los siguientes pasos:

Se accede a Playlists desde la página principal: Aquí se encuentran ordenadas las diferentes listas de reproducción almacenadas en nuestro sistema. Se muestra el nombre y el estado, además se permite iniciarlas o detenerlas. Desde aquí se tiene acceso a editar, eliminar y añadir listas de reproducción. En este caso se quiere añadir una lista media (vídeo).

Una vez en el menú para añadir lista se encuentran una serie de campos para introducir el nombre y el recurso, estos campos fueron comentados con anterioridad. También se puede seleccionar el tipo de reproducción y el número de vídeos que precisan ser reproducidos hasta poder repetir otro. En la lista de la izquierda aparecen los vídeos disponibles (por defecto en la carpeta /usr/local/movies), mientras que a la derecha se muestran los elementos añadidos a la lista. Es recomendable generar listas de reproducción que contengan archivos codificados con el mismo

sistema, bitrate, etcétera. De lo contrario pueden encontrarse problemas, tanto a la hora de generar la lista, como a la hora de reproducirlo en ciertos terminales.

Con los archivos, nombre, montaje y características seleccionadas, se puede optar por almacenar información de la lista (log) o enviarla a otro servidor como si se tratara de un sistema de relay. Tras seleccionar las características deseadas guardamos los cambios y se vuelve a la pantalla anterior donde se puede iniciar la reproducción. El servidor arrancará automáticamente la aplicación PlaylistBroadcaster e iniciará el servicio de la lista cada vez que arranque el servidor. También generará el recurso .sdp y las carpetas pertinentes dentro de playlists.

4.3.5. Desde una consola

En caso de no contar con interfaz gráfica, se puede generar y reproducir las listas de forma manual. En este caso los pasos a seguir son los siguientes:

Para crear una lista de reproducción es preciso editar dos archivos de manera manual. Para ello se debe ir al directorio /var/streaming/playlists y crear un nuevo directorio con el nombre de nuestra lista. Por ejemplo se puede crear plist: mkdir plist.

Dentro del directorio se editarán los dos archivos necesarios. Empezando por un archivo plist.playlist, para ello se puede usar cualquier editor (vi, nano...). Este archivo debe tener una apariencia similar a esta:

```
*PLAY-LIST*
"/usr/local/movies//archivo.3gp" 5
```

Simplemente se indica que se trata de una playlist y los archivos implicados. El 5 es el valor del peso del archivo, utilizado únicamente en sistemas weighted random. A continuación se edita el archivo plist.config que debería tener este aspecto:

```
playlist_file /var/streaming/playlists/plist/plist.playlist
play_mode sequential
destination_ip_address 127.0.0.1
#broadcast_name "plist"
sdp_file "/var/streaming/playlists/plist/plist.sdp"
destination_sdp_file "plist.sdp"
broadcast_SDP_is_dynamic enabled
pid_file "/var/streaming/playlists/plist/plist.pid"
```

En este archivo se está indicando donde se encuentra el archivo .playlist, el modo de reproducción, la IP de destino, el archivo SDP a utilizar (si existe), el SDP que generará el broadcaster, se indica si el SDP es dinámico y finalmente se asigna un identificador de archivo. A continuación se ejecuta el PlaylistBroadcaster con los siguientes parámetros:

PlaylistBroadcaster plist.config

El cual generará automáticamente un .sdp, un .current, .log, .pid y .upcoming. El archivo .sdp necesita ser copiado a la carpeta /usr/local/movies para ser accesible desde los clientes. Mientras PlaylistBroadcaster esté ejecutándose el recurso será accesible remotamente. Si el servidor o el proceso de broadcast se detienden, la lista dejará de ser accesible. Aunque se reinicie el servidor, una vez copiado el .sdp, simplemente se deberá ejecutar el comando recuadrado anteriormente con las listas de reproducción que se deseen, incluso es posible configurarlo para que lo haga automáticamente en un pequeño script.

PlaylistBroadcaster permite gestionar más de una lista simultáneamente, ver las opciones (-h) para comprobar como añadir, eliminar y modificar reproducciones en curso.

4.3.6. Creación de una lista de reproducción multicast

Como se comentó con anterioridad, las listas de reproducción suelen lanzar la información contra el propio servidor, no obstante, es posible definir una dirección de destino diferente. Dado que lanzar una lista de reproducción contra una única máquina no tiene mucho sentido[34] se estudia la posibilidad de hacerlo hacia una dirección multicast. Para ello únicamente es necesario realizar unas pequeñas modificaciones en los archivos de configuración para editar la dirección de destino y un TTL[35] válido. Al no permitir la interfaz gráfica este tipo de modificaciones, la ejecución deberá realizarse desde la consola.

A partir de este momento se genera un SDP válido que debe ser entregado al cliente para permitirle conectarse al flujo multicast.

[34] Para ello existe el sistema de *relays* comentado en el apartado posterior.

[35] Time To Live que indica el número de saltos permitidos a los paquetes del flujo.

```
v=0
o=QTSS_Play_List 3326963450 2369604953 IN IP4 192.168.1.100
c=IN IP4 239.255.12.43/5
b=AS:2097200
t=0 0
a=x-broadcastcontrol:RTSP
a=isma-compliance:2,2.0,2
m=video 5004 RTP/AVP 96
b=AS:2097151
a=rtpmap:96 H264/90000
a=control:trackID=1
a=cliprect:0,0,480,380
a=framesize:96 380-480
a=fmtp:96 packetization-mode=1;profile-level-id=4D401E;sprop-parameter-
sets=J01AHqkYMB73oA==,KM4NiA==
a=mpeg4-esid:201
m=audio 5006 RTP/AVP 97
b=AS:48
a=rtpmap:97 mpeg4-generic/22050/2
```

Darwin Streaming Server no permite la distribución bajo demanda de archivos SDP con direcciones de conexión multicast, de hecho típicamente machaca el campo de conexión descrito por la dirección local de la máquina; por este motivo, la mejor manera de distribuir un SDP de estas características es embeber su ejecución en una página HTML para que los clientes compatibles la reproduzcan a través del navegador con el *plug-in* correspondiente.

4.3.7. Sistema de relays

En ocasiones resulta interesante localizar la fuente de vídeo en un lugar diferente al del servidor. Esto suele ocurrir principalmente cuando los contenidos están generados de forma continua remotamente o la plataforma en la que se implementa el servidor de vídeo no tiene capacidad de almacenamiento o proceso suficiente. Estos sistemas nos permiten tener la información almacenada en una máquina y anunciar su presencia a uno o más servidores. De este modo los usuarios conectados a un servidor pueden acceder, no sólo a la información almacenada localmente, si no también a aquella provista por el sistema de *relays*.

Para configurar un sistema de *relay* se pueden utilizar dos tipos principales de fuente: Un *broadcaster* u otro servidor de *streaming*. La principal diferencia entre ellos es que el *broadcaster* es un dispositivo pensado para actuar como fuente de vídeo para otros elementos, mientras que un servidor de *streaming*, puede actuar como fuente o como "puente", pero esa no es su principal misión. A efectos prácticos, los servidores que actúan como destino no tienen por qué diferenciar se de los otros. Se han realizado pruebas utilizando un PC convencional con un servidor de *streaming* configurado como fuente y un dispositivo Futurlink como receptor del flujo y servidor para terminales.

Para establecer una relación de *relay* (o de origen/destino) de vídeo es suficiente con definir dos parámetros; la fuente de origen y la de destino según de donde procede la información y hacia donde será encaminada, además se debe indicar el tipo de información que se espera recibir/enviar. En

muchas ocasiones es posible que el origen o el destino sea nuestra propia máquina, incluso se puede definir múltiples destinos para un mismo origen incluyéndonos al propio servidor y, evidentemente, es posible configurar más de un origen. A continuación se comentan dos casos principales. En primer lugar, dos DSS actuando tanto como servidores de *streaming*, como de *relay* fuente y destino respectivamente. En segundo lugar se presenta una alternativa para generar contenido apto para la distribución desde otras fuentes.

4.3.8. Servidor de streaming actuando como fuente de vídeo

Lo que se pretende en este caso, es que parte de la información que se tiene disponible localmente, esté también disponible para los clientes de otro servidor de streaming. Para ello se va a configurar un sistema de *relay unicast* con una fuente y un destino. Se va a utilizar la configuración por defecto que proporciona el sistema para este propósito. En este caso se configura, ya sea manualmente (/etc/streaming/relayconfig.xml) o mediante interfaz gráfico el *default_relay* añadiéndole el destino que se quiera para la información.

Los destinos de los sistemas de *relay* pueden ser *unicast* o multicast y anunciado o no. En las pruebas realizadas hasta el momento se ha utilizando siempre un sistema *unicast* dado que se ha trabajado con un punto de acceso. Utilizar un destino anunciado supone enviar información al receptor sobre el flujo de información disponible, mientras que un sistema no anunciado trabaja en un puerto determinado.

Inicialmente se ha configurado una lista de reproducción en el PC y se ha agregado la dirección IP del punto de acceso como destino de un *relay* UDP no anunciado. El dispositivo Futurlink se ha configurado para recibir el flujo desde la dirección del PC y almacenarla localmente. El sistema ha demostrado ser bastante sensible dado que la interrupción de la lista de reproducción o modificación en los parámetros locales o remotos, puede suponer la interrupción de la transmisión. También se han presentado algunos problemas de autenticación entre punto de acceso y PC que no se han dado en sistemas locales. A continuación se muestra un ejemplo de una configuración local, aplicable a condiciones remotas:

```
<RELAY_CONFIG>
<OBJECT TYPE="relay" NAME="***qtssDefaultRelay***">
   <OBJECT CLASS="source" TYPE="announced_source">
   </OBJECT>
   <OBJECT CLASS="destination" TYPE="announced_destination">
      <PREF NAME="password"></PREF>
      <PREF NAME="dest_addr">192.168.57.100</PREF>
      <PREF NAME="name"></PREF>
      <PREF NAME="url"></PREF>
   </OBJECT>
</OBJECT>
<OBJECT TYPE="relay" NAME="client">
   <OBJECT CLASS="source" TYPE="rtsp_source">
      <PREF NAME="password"></PREF>
      <PREF NAME="name"></PREF>
      <PREF NAME="url">lista.sdp</PREF>
      <PREF NAME="source_addr">192.168.57.100</PREF>
   </OBJECT>
   <OBJECT CLASS="destination" TYPE="announced_destination">
      <PREF NAME="password"></PREF>
      <PREF NAME="dest_addr">127.0.0.1 </PREF>
      <PREF NAME="name"></PREF>
      <PREF NAME="url">/prueba.sdp</PREF>
   </OBJECT>
</OBJECT>
</RELAY_CONFIG>
```

En este caso el servidor fuente tiene una lista de reproducción ejecutándose llamada lista.sdp la cual anuncia a la IP 192.168.57.100 (su misma IP privada, aquí la IP debería ser el destino deseado). No se incluyen los campos *url*, *user* y *password* ya que no se desea autenticación y se anuncian todas las listas. No es estrictamente necesario realizar el anuncio si en destino se define la fuente como RTSP.

En recepción se indica que existe una fuente en la dirección indicada, la cual no requiere autenticación y se desea utilizar el recurso lista.sdp de manera que todo lo que reproduzca dicha lista esté disponible localmente. Para realizar este almacenamiento local se añade como destino la dirección IP local (127.0.0.1) y se indica como *url* el nombre que se desea dar al recurso. A partir de este momento los usuarios móviles pueden acceder a prueba.sdp y visualizar lo que se distribuye mediante lista.sdp. Es importante que el directorio de destino (por defecto /usr/local/movies) pertenezca al usuario y el grupo correctos. Por defecto en los sistemas Futurlink el grupo es distinto a *root* provocando problemas en la generación automática de archivos *sdp*.

4.3.9. Utilizar VLC como servidor no anunciado localmente

Una de las configuraciones interesantes a la hora de utilizar fuentes de vídeo externas es el uso de programas que hagan transcodificación, como por ejemplo Video Lan Client VLC. La idea principal

es adaptar los contenidos provistos por una fuente externa a las características exigidas por nuestro servidor, además este sistema debe permitirnos realizar el *streaming* de dicho contenido de manera continua, es decir, sin transcodificar y almacenar la información de manera manual.

En este caso se utiliza el programa VLC para capturar el flujo de vídeo de la fuente, puede tratarse de información previamente almacenada (archivo de vídeo en otra extensión, disco óptico, etcétera) aunque resulta especialmente interesante capturar la información de una fuente continua (DVB, TV sobre IP, cámaras...). Para que VLC genere contenidos que se puedan utilizar con DSS es posible utilizar diferentes comandos. En primer lugar se muestra como generar dinámicamente un archivo .sdp de manera que sea útil para DSS.

```
vlc /usr/local/movies/archivo.3gp --sout '#rtp{dst=127.0.0.1,port=1234,sdp=file:///usr/local/movies/vlc.sdp}'
```

Con este comando se está abriendo un archivo de vídeo 3GP y sirviéndolo de manera local vía RTP a nuestra propia máquina. Nótese que el archivo se almacena en la carpeta por defecto del servidor DSS, de esta manera cualquier usuario remoto que acceda al recurso vlc.sdp reproducirá la información generada por el programa VLC. Se utiliza un puerto aleatorio que no cree conflicto con ninguna otra aplicación. Utilizando los parámetros propios de VLC para transcodificar vídeo, se puede realizar el mismo proceso con cualquier otro tipo de archivo no 3GP.

Si se desea aplicar el proceso desde una fuente dinámica, se debe utilizar un comando similar a este:

```
vlc  v4l:/dev/video0:norm=secam:frequency=543250:size=640x480:channel=0:adev=/dev/dsp:audio=0
--sout
'#transcode{vcodec=mp4v,acodec=mpga,vb=256,ab=64,deinterlace}:rtp{dst=127.0.0.1,port=1234,sdp
=file:///usr/local/movies/vlc.sdp}'
```

En esta ocasión, esta fuente es una cámara de vídeo reconocida por Video 4 Linux como dispositivo /dev/video0, se utiliza captura en formato SECAM e indica tasa y resolución. Se puede apreciar como se realiza un proceso de transcodificación utilizando vídeo MP4 desentrelazado y audio MPG con bitrates 256Kbps y 64Kbps respectivamente. De nuevo el destino es la carpeta de vídeos, de manera que cualquier usuario remoto podrá conectar al recurso vlc.sdp y apreciar en tiempo real (aunque con cierto retardo debido a los distintos procesados), lo que ocurre en la cámara.

Es lógico pensar que la plataforma InfoPoints no está pensada para realizar este tipo de procesado, por ello resulta útil generar esta información de manera remota (en un servidor) y posteriormente anunciarla a la plataforma como si se tratara de una lista de reproducción más, utilizando una configuración relay.

4.3.10. Evaluación de viabilidad para PUSH de vídeo

Uno de los puntos propuestos para estudio en la segunda fase del proyecto *InfoPoints* fue la posibilidad de realizar un *push* de información hacia el terminal móvil. El término *push* se puede definir como la oferta de información no solicitada por parte del cliente. La idea es observar las posibilidades

de llevar a cabo este push tanto por parte del servidor como del cliente. Inicialmente se piensa en *push* relativos al sistema de vídeo. En este apartado se pretenden evaluar las posibilidades técnicas tanto a nivel de protocolos como de las aplicaciones disponibles.

El servidor DSS es un servidor de vídeo bajo demanda pasivo, esto implica que se dedica fundamentalmente a reaccionar ante las peticiones realizadas por clientes. Si bien es cierto que tiene también la capacidad de emitir flujos de vídeo no solicitados como listas de reproducción o actuando como relay. En el escenario de un *push* de vídeo es necesario no sólo la transmisión de un vídeo, si no el anuncio de este de forma específica a cada uno de los clientes. Para poder realizar este tipo de anuncios es preciso detectar la presencia de los clientes, la cual resulta compleja a nivel de servidor de vídeo. Una alternativa más accesible es la realización del proceso de detección a nivel de asociación debido a que se utiliza el sistema de detección de capacidades del terminal el cual puede devolver una URL al recurso deseado para realizar el *push*.

Respecto a la tecnología implicada, RTSP es un protocolo que contempla la posibilidad de establecer comunicaciones con una petición lanzada desde el servidor en lugar del cliente tal y como se define en el estándar. El problema en este apartado es que DSS, tal y como se comentó con anterioridad, es un servidor pasivo que no lanza peticiones hacia los clientes y que, aunque así fuera mediante una modificación de software a través de un nuevo módulo, todos los mensajes derivados del cliente deberían seguir la estructura presentada en la Figura 10, ya que todos los roles del servidor están definidos tal y como se comentará en posteriores apartados Todas estas consideraciones indican la complejidad del desarrollo de un sistema push en DSS.

Independientemente de los problemas causados a nivel de servidor, el sistema de *push* tiene su principal obstáculo en el cliente. En primer lugar no se posee acceso al software de dichos dispositivos, con lo que la generación de una interrupción a partir de un *push* no es una opción disponible. Los terminales que reciben un mensaje RTSP simplemente lo ignoran y continúan con sus tareas. Para establecer algún tipo de reacción ante tales estímulos sería necesario diseñar un software específico para la plataforma en cuestión. No sólo no se genera una interrupción en el estado reposo del terminal si no que tampoco ocurre cuando éste se encuentra ejecutando un reproductor, con lo que la ejecución de un hipotético push hacia un reproductor tampoco tendría éxito. En principio no se considera la posibilidad de instalar software adicional en los terminales.

Actualmente se proponen algunas soluciones basadas en un *push* bluetooth o vía WAP en las cuales se ofrece el recurso deseado para cada terminal vía mensaje. A partir de ese momento el cliente ataca al recurso proporcionado siguiendo la estructura de flujo habitual presentada con anterioridad. Esto supone no realizar ninguna modificación en el servidor de vídeo. Los beneficios de estas soluciones son, además de mantener la estructura del servidor intacta, trabajar con un sistema de interrupciones conocido y bien definido ya que la totalidad de los terminales muestra avisos visuales y sonoros ante la llegada de un mensaje. La principal problemática introducida es el hecho de necesitar una disponibilidad por parte del terminal a recibir contenido bluetooth o WAP, tanto a nivel de capacidades como a nivel de configuración. Por otra parte, muchos terminales trabajan con el interfaz bluetooth desactivado.

Existen diferentes soluciones basadas en push HTML y *pushletts* java que pueden utilizarse durante la navegación del cliente. Este tipo de push permite enviar información HTML de manera periódica, esta información llega al cliente siempre que éste se encuentre navegando en las páginas indicadas. La información enviada al cliente puede ser una página HTML personalizada según las circunstancias del cliente (localización, capacidades, estado del enlace, etcétera). Generalmente estos sistemas se basan en temporizadores que lanzan de manera periódica información al cliente.

La solución presentada incluye la programación de una página HTML con *javascript* que consta de tres marcos. El primer marco muestra una página cualquiera; en el ejemplo se trata de una serie de texto por la que el cliente puede ir navegando. El segundo marco, denominado *display*, es el objetivo en el que se mostrarán los vídeos embebidos. Finalmente el tercer marco permanece oculto y en él se encuentra el código encargado de hacer el *push* como puede apreciarse en la Figura 11. En el ejemplo presentado este marco tiene una lista de URL que entrega periódicamente al marco de vídeo embebido utilizando una serie de temporizadores; de este modo se consigue que el cliente reciba un vídeo diferente en cada ocasión durante un tiempo determinado.

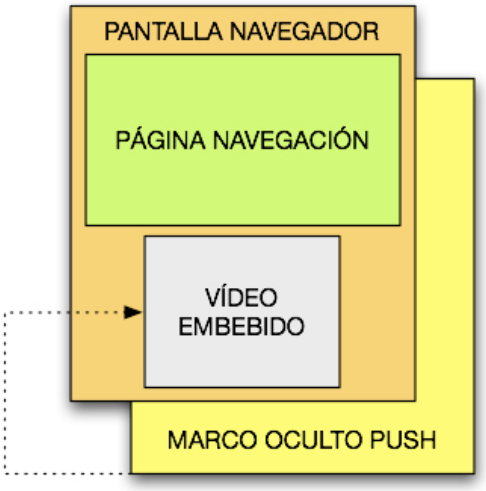

Figura 11: Estructura solución PUSH.

Además de entregarse un vídeo embebido, puede entregarse directamente el recurso RTSP. Esto supone que los clientes ejecutan automáticamente el reproductor; es importante destacar que durante este tiempo no se pueden realizar nuevos *push* ya que se requiere que el cliente se encuentre navegando.

5. Programación de módulos para DSS

5.1. Introducción

Una de las características principales del servidor Darwin Streaming Server, es la capacidad que ofrece para la implementación de modificaciones. Estas modificaciones pueden resultar interesantes para adaptar las características del servidor a las necesidades de los usuarios, implementar nuevas funcionalidades, añadir compatibilidad con formatos y terminales o simplemente modificar el comportamiento del servidor en diferentes circunstancias.

Existen fundamentalmente dos tipos de modificaciones; la primera que afecta directamente al código ya escrito del servidor, y la segunda que consiste en la implementación de módulos adicionales que incorporen diferentes funcionalidades. Desde un punto de vista práctico resulta más útil realizar las modificaciones en módulos externos ya que tienen una mayor probabilidad de ser compatibles con futuras versiones del servidor (siempre que éste no modifique las estructuras, funciones, etcétera). Por este motivo, siempre que sea posible, se evitará modificar el código fuente ya provisto. A continuación se presentarán las directrices básicas a seguir para implementar un módulo para el servidor: Su estructura, los roles que puede desempeñar, los parámetros que puede utilizar y las estructuras de los datos en cada circunstancia. La mayor parte de la información incluida en este apartado, así como definiciones más concretas de estructuras y funciones, pueden encontrarse en [17].

Toda la programación llevada a cabo en este punto se basa en el lenguaje C++ en el que está implementado el servidor Darwin Streaming Server.

5.2. Roles en el servidor

El servidor DSS funciona en base a un sistema de roles. Cada módulo puede pertenecer a uno o varios roles dependiendo de las funcionalidades que deba tener; los roles que se indiquen en el módulo definirán qué datos puede tratar el programa y en qué momento será llamado por el servidor. Los roles se pueden clasificar fundamentalmente en aquellos que son estáticos y dinámicos. Los estáticos son aquellos a los que se les llama al inicializar o finalizar el servidor, mientras que los dinámicos serían aquellos a los que se llama al ocurrir un evento concreto como recibir un paquete RTSP. La estructura de todos los roles disponibles es la mostrada en la Tabla 5.

Nombre	Función
Register	Registra nuestro módulo y sus roles en el servidor.
Initialize	Inicializa el módulo.
Shutdown	Realiza tareas de limpieza tras la ejecución.
Reread Preferences	Permite leer preferencias de archivos de configuración.
Error Log	Permite almacenar información de errores.
RTSP Filter	Filtra peticiones RTSP.
RTSP Route	Permite modificar peticiones RTSP.
RTSP Prepocessor	Permite procesar las peticiones antes que el servidor.
RTSP Request	Procesa las peticiones RTSP si ningún otro módulo lo hace.
RTSP Posprocessor	Permite realizar tareas tras la respuesta RTSP.
RTP Send Packets	Envía paquetes RTP
Client Session Closing	Actúa cuando la sesión de cliente finaliza.
RTCP Process	Procesa mensajes RTCP.
Open File Preprocess	Procesa peticiones de apertura de archivos.
Open File	Procesa las peticiones no resueltas por Preprocess.
Advise File	Responde a llamadas QTSS_Advise.
Read File	Permite leer un archivo.
Request Event File	Informa de cuando un archivo está listo para lectura.
Close File	Permite cerrar archivos.

Tabla 5: Roles disponibles en Darwin Streaming Server

Dentro de el proceso de paquetes RTSP el servidor llama a los módulos por orden estricto en función de la respuesta obtenida por los anteriores. El orden a seguir es el presentado en esta tabla (filter, route, preprocessor, request, postprocessor), es decir: Al llegar un paquete RTSP el servidor llamará a aquellos módulos que hayan registrado el rol de RTSP Filter, en caso de que no haya ningún módulo o los módulos llamados no devuelvan la respuesta esperada al cliente, el servidor llama a los módulos registrados en el siguiente rango. Si tras llamar a los módulos registrados en el rol de postproceso no se recibe respuesta, el sistema continúa la ejecución normal. Dentro de un mismo rol, los módulos son llamados de forma aleatoria.

Es importante considerar en qué rol debe registarse el módulo para poder procesar los mensajes antes o después que el servidor si así se desea, y minimizar el impacto del módulo en el funcionamiento normal de éste. Dependiendo del rol escogido, los parámetros que el servidor proporcionará al módulo serán diferentes lo que permitirá realizar unas u otras operaciones. A continuación se muestra un ejemplo de una función de registro de roles llamada *Register*, la cual es llamada al iniciarse el servidor y permite definir en qué momento deberá invocarse el módulo:

```
QTSS_Error Register(QTSS_Register_Params* inParams)
{
    //Set the Roles
    (void)QTSS_AddRole(QTSS_Initialize_Role);
    (void)QTSS_AddRole(QTSS_RereadPrefs_Role);
    (void)QTSS_AddRole(QTSS_RTSPRoute_Role);

    // Tell the server our name!
    static char* sModuleName = "Selector";
    ::strcpy(inParams->outModuleName, sModuleName);

    return QTSS_NoErr;
}
```

En este caso se registran los roles de inicialización, lectura de preferencias y enrutamiento RTSP, el de registro no es necesario. Además se indica al servidor el nombre del módulo.

5.3. Estructura básica de un módulo

En este apartado se presenta la estructura que debe tener un módulo para ser compilado y ejecutado en el servidor Darwin Streaming Server.

5.3.1. Archivos

Un módulo se compone fundamentalmente de dos archivos, un archivo fuente .cpp y una cabecera .h. Su uso es el mismo que en cualquier programa escrito en C++. El archivo de cabecera será referenciado desde el servidor como se explicará en los siguientes apartados.

5.3.2. Funciones

A la hora de programar un módulo para DSS es imprescindible que cuente, al menos, con las siguientes dos funciones o rutinas:

• **Main :** Es la función principal del módulo y es ejecutada en la inicialización del servidor. En ella se debe incluir referencia a la función *Dispatch*, la cual es ejecutada cuando el servidor llama al módulo,

es decir, cada vez que el módulo sea requerido durante la ejecución del servidor. A continuación se muestra un ejemplo simple, donde un módulo llamado Selector define su función *Main*:

```
QTSS_Error Selector_Main(void* inPrivateArgs)
{
    return _stublibrary_main(inPrivateArgs, SelectorDispatch);
}
```

La función *Main* siempre debe retornar un error bien definido en el entorno del servidor DSS conocido como QTSS_Error. En este caso se indica que la rutina *Dispatch* se ha definido como *SelectorDispatch*, gracias a la función *stublibrary*, el servidor podrá llamar al módulo.

- **Dispatch**: Es la rutina que se ejecutará cada vez que el módulo sea requerido por el servidor, por ejemplo a la llegada de un paquete RTSP. En este ejemplo el módulo tiene una función *register*, llamada al iniciarse el servidor que define su funcionamiento en diferentes roles (*Register*, *Initialize*, *RTSPRoute* y *RereadPrefs*):

```
QTSS_Error  SelectorDispatch(QTSS_Role inRole, QTSS_RoleParamPtr inParamBlock)
{
    switch (inRole)
    {
        case QTSS_Register_Role:
            return Register(&inParamBlock->regParams);
        case QTSS_Initialize_Role:
            return Initialize(&inParamBlock->initParams);
        case QTSS_RTSPRoute_Role:
              ProcessRTSPRequest(&inParamBlock->rtspRequestParams);
        case QTSS_RereadPrefs_Role:
            return RereadPrefs();
    }
    return QTSS_NoErr;
}
```

- Los dos primeros y el último son referentes a la inicialización del servidor y se utilizan para registrar roles, inicializar el módulo y leer las preferencias desde el archivo de configuración respectivamente. En particular, durante el periodo de registro se llama a la función de registro de roles *Register*, definida en el punto anterior. Además se registra el rol *RTSPRoute* para tratar este tipo de peticiones. Cada vez que el servidor detecta una de ellas llama al módulo siguiendo la jerarquía explicada con anterioridad. Cada rol tiene implícita una rutina que es ejecutada dependiendo del motivo por el que se ha llamado al módulo.

Además de estas dos funciones, cada rol implica la ejecución de una o más funciones adicionales. Es importante tener en cuenta de dependiendo del rol escogido los parámetros con los que trabajarán dichas funciones son diferentes. En el siguiente apartado se comentan las principales estructuras de datos que maneja el servidor.

5.4. Estructuras de datos

5.4.1. Introducción

Al igual que ocurre con los roles, en el servidor existen una serie de estructuras de datos bien definidas con las que se trabaja en cada una de las peticiones. Dependiendo del rol que desempeñe un módulo concreto, el servidor le proporcionará una estructura de datos diferente, la cual podrá leer y, en ocasiones, escribir. Estas estructuras contienen información relacionada con atributos de usuario, sesiones, flujos etcétera, y son conocidas como objetos. A continuación se enumeran los objetos bien definidos en Darwin Streaming Server, de nuevo esta información puede encontrarse más detallada en [17]:

Nombre	Contiene
qtssAttrInfoObjectType	Información sobre un atributo.
qtssClientSessionObjectType	Información sesión cliente.
qtssConnectedUserObjectType	Información de un usuario conectado.
qtssDynamicObjectType	Utilizado para generar objetos dinámicos.
qtssFileObjectType	Información sobre la apertura de un archivo.
qtssModuleObjectType	Descripción de un módulo.
qtssModulePrefsObjectType	Contiene atributos con preferencias de los módulos.
qtssRTPStreamObjectType	Atributos de un flujo RTP (audio o vídeo).
qtssRTSPHeaderObjectType	Información en las cabeceras de peticiones RTSP.
qtssRTSPRequestObjectType	Atributos que describen una petición RTSP.

Tabla 6: Principales objetos disponibles en Darwin Streaming Server

Dentro de cada uno de los objetos definidos, existen una serie de atributos que pueden ser de diferentes clases (caracteres, enteros de 32 bits, boleanos, tipos definidos exclusivamente para el servidor, etcétera). Además cada uno de ellos puede ser de lectura, escritura o ambos. Para obtener información más detallada es interesante acceder al manual de programación del servidor.

A continuación se comentan algunos de los objetos que se utilizan de forma más frecuente en la programación de módulos:

5.4.2. Sesión de cliente

Cada cliente que accede a recursos disponibles en el servidor tiene asociada una sesión única representada como un atributo. Además se incluye información adicional sobre el recurso que solicitó el cliente, los paquetes y Bytes RTP transmitidos en esta sesión, el estado de la misma, el tiempo conectado, la IP del cliente, la URL que solicitó, la duración de la película, el tamaño de la misma, etcétera. En ningún momento se relaciona la sesión del cliente con los flujos que transmite.

Toda la información relacionada a la sesión de cliente se pasa como atributo a aquellos módulos que trabajan en roles los cuales reciben como parámetro un objeto del tipo *QTSS_ClientSessionObject*. Todos los atributos vienen definidos por el servidor, el cual los envía posteriormente al módulo. Dependiendo del rol en el que se implique el módulo, esta información podrá ser modificada antes o después de ser procesada por el servidor (otro módulo), pero en ningún caso podrá modificarse en el cliente.

5.4.3. Usuario conectado

Este objeto proporciona información sobre un usuario conectado independientemente del transporte utilizado. En particular se enfoca a clientes que no descargan películas, ya que estos se encuentran representados en el apartado anterior. Este objeto es interesante para ser enfocado hacia usuarios que reproducen MP3 u otro tipo de objeto.

Los atributos del objeto son similares a los de sesión de cliente incluyendo: Tasa de transmisión, bits enviados, hora de creación, punto de montaje, porcentaje de pérdidas, tiempo de conexión, etcétera.

5.4.4. Flujo RTP

Un flujo RTP hace referencia a una transmisión de datos referente a un flujo multimedia, ya sea de audio o vídeo. Un mismo usuario dentro de una sesión tiene varios flujos RTP. Cada flujo RTP tiene un identificador único (típicamente igual al track ID anunciado por el servidor mediante el protocolo SDP), información sobre la reproducción tal como retardo del buffer, números de secuencia, timestamp, escala utilizada para dichas marcas de tiempo, tipo de información que se transmite, SSRC, retardo medio, porcentaje de llenado en el buffer del cliente, jitter, tasa de cuadros por segundo, paquetes total recibidos y perdidos, etcétera. Además informa de si el flujo es transmitido sobre TCP o UDP, entre otros.

Los valores de los atributos, como en la mayoría de casos, vienen dados por el servidor. El hecho de modificar los identificadores para un flujo RTP no permite el cambio de flujo en la transmisión con lo que no es un sistema válido para adaptación de tasa. Del mismo modo, la modificación de otros parámetros del flujo RTP no afecta al comportamiento del cliente.

5.4.5. Petición RTSP

Se entiende como una petición RTSP aquella que contiene información en el campo Request. Estas peticiones se utilizan para consultar las opciones disponibles en el servidor (OPTIONS), solicitar información sobre un recurso (DESCRIBE), inicializar su transmisión (SETUP), iniciar el flujo (PLAY), pausarlo (PAUSE), detenerlo (SHUTDOWN), etcétera. Una petición RTSP está ligada a una dirección URL, típicamente con el formato rtsp://IP:PUERTO/recurso. Además de este atributo y sus derivados (dirección truncada, recurso, dígito final de la petición, etcétera) se incluyen también la petición completa (con todos sus campos), el método, el modo de petición, la velocidad soportada, el tiempo en el que se solicita un PLAY, etcétera.

Las peticiones RTSP son útiles para modificar el contenido de éstas, especialmente antes de iniciar la transmisión de un flujo. De hecho se ha utilizado este modo para seleccionar un flujo adecuado previo a la primera reproducción del servidor. Para ello es necesario un módulo cuyos roles permitan la evaluación de peticiones RTSP antes de que lo haga el propio servidor (u otros módulos). Una vez interceptada la petición se puede utilizar el criterio que se considere oportuno (wap profile, IP del usuario, información almacenada, etcétera) para entregar al cliente uno u otro flujo modificando la URL a la que el cliente desea acceder. Esta modificación es invisible tanto para el servidor como para el usuario ya que, por una parte el servidor no ha recibido notificación ninguna del cliente respecto a un recurso concreto y, por otra parte, el cliente desconoce la ubicación exacta del recurso al que accede (no tiene por qué ser igual a la URL a la que accede). Para más detalles consulte el apartado referente a dicho módulo.

5.4.6. Sesión RTSP

Una sesión RTSP consiste en una relación establecida entre un servidor y un cliente mediante dicho protocolo. Dentro de los atributos disponibles en este objeto es posible encontrar el identificador de sesión, dirección IP y puerto local y remotos, flujo de referencia y tipo de sesión entre otros.

5.4.7. Interacción con los objetos

Para el desarrollo de un programa dentro del servidor de streaming es imprescindible acceder y modificar algunos de los atributos de los objetos presentados anteriormente. Para ello se cuenta con una serie de funciones bien definidas:

- QTSS_GetValue: Permite obtener la información de un atributo.
- QTSS_GetValueAsString: Idéntico a GetValue pero devolviendo caracteres.
- QTSS_GetValuePtr: En este caso se obtiene la información como puntero.
- QTSS_SetValue: Permite modificar el valor de un atributo.
- QTSS_SetValuePtr: Idéntico desde un puntero.

Estas cinco funciones deben utilizarse con atributos que sean *preemptive safe* para evitar conflictos de valores. Además es recomendable bloquear

el acceso a atributos cuando no sean *preemptive safe* mientras se realizan diferentes operaciones entre las lecturas y escrituras sobre ellos. Para ello se pueden utilizar las funciones *QTSS_UnlockObject* y *QTSS_LockObject.*

5.5. Integración del módulo en el servidor

5.5.1. Código fuente del servidor

En primer lugar cabe destacar que, para añadir módulos, es necesario trabajar con el código fuente del servidor en lugar de utilizar una versión precompilada. El código fuente está disponible en la misma dirección que las versiones preparadas para diferentes sistemas operativos, todos los requisitos necesarios para su compilación vienen indicados en los documentos de asistencia dentro del propio archivo comprimido.

Además del código fuente y los módulos predefinidos del servidor, se encuentran una serie de carpetas y ficheros con funciones comunes y utilidades relativas al servidor. Estas utilidades permiten procesar peticiones, truncar URLs, generar respuestas y un largo etcétera. Es interesante consultar dichos archivos ya que el manual no ofrece excesiva información sobre todas las funciones implementadas. Las funciones están disponibles en las diferentes carpetas de herramientas y utilidades dependiendo de su finalidad.

Además de añadir los módulos como tales, es necesario modificar parte del código fuente del propio servidor de streaming añadiendo algunas líneas de forma que se indique la presencia de los nuevos módulos implementados. En particular, el archivo a modificar se encuentra en: *directorio_del_servidor/Server.tproj/QTSServer.cpp*

Las modificaciones a las que debe someterse son, en primer lugar, indicar la inclusión de nuestra cabecera y en segundo lugar, añadir nuestro módulo a la sección de módulos dinámicos. Pueden apreciarse las modificaciones a realizar para añadir un módulo llamado Selector:

Referencia a la cabecera del módulo en la sección de *includes*:

```
#include "Selector.h"
```

Adición del módulo en la sección de módulos dinámicos, dentro de la función *QTSServer::LoadCompiledInModules():*

```
QTSSModule* Selector = new QTSSModule("Selector");
(void)Selector->SetupModule(&sCallbacks, &Selector_Main);
(void)AddModule(Selector);
```

En este caso *Selector_Main* es la función *Main* de nuestro módulo y Selector es el nombre de éste.

5.5.2. El Makefile

Además de modificar el archivo principal, es necesario modificar el archivo *Makefile.POSIX* incluido en la carpeta raíz del servidor, de manera que al ejecutar el *build,* nuestras carpetas de módulos se encuentren incluidas. Para ello es necesario realizar tres modificaciones:

En primer lugar se añade a la ruta de librerías aquellas que se quieran utilizar. En este ejemplo: libpqxx.

```
LIBS= … -lpq -lpqxx
```

En segundo lugar se añade la cabecera del nuevo módulo en las *CCFLAGS* de la siguiente manera:

```
CCFLAGS += -IAPIModules/Selector
```

Siendo que el módulo Selector se encuentra incluido en la carpeta *APIModules/Selector.*

Por otra parte se añaden los archivos principales *CPP* del módulo dentro de la sección *CPPFILES*:

```
APIModules/Selector/Selector.cpp
```

En este caso el módulo cuenta con un único archivo llamado *Selector.cpp* dentro de la carpeta mencionada anteriormente.

5.5.3. Compilación e instalación

Una vez programados los módulos, modificado el servidor y enlazados los archivos puede compilarse con normalidad mediante la instrucción *./Buildit.* En caso de no obtener errores la instalación se lleva a cabo a continuación mediante el comando *./Install* del mismo modo que se explicó en el apartado de instalación. La creación de carpetas y archivos es idéntica y el servidor conserva todas las funcionalidades (GUI, *relays*, listas de reproducción...) salvo las modificadas por los módulos recientemente añadidos.

5.5.4. Generación de un paquete instalable

Una vez implementados los módulos se pude generar una versión precompilada que sea compatible con máquinas de la misma arquitectura. En este caso se han implementado las modificaciones en un PC convencional y se ha generado un archivo precompilado para ser instalado en los puntos de acceso. Esto es posible gracias a que comparten la arquitectura y sistema operativo Linux, las versiones de Mac necesitan una compilación diferente.

Una vez generados todos los archivos relativos a los módulos, modificado el código del servidor y el archivo de *Makefile*. Se procede con la instrucción *./buildtarball,* la cual genera un archivo *.tar.gz* preparado para descomprimirse e instalarse como una versión precompilada.

5.5.5. Las preferencias

Una de las alternativas para controlar la gestión de los módulos, ya sean predefinidos o genuinos son las preferencias del servidor. Con éste instalado y funcionado, se pueden modificar algunos de sus comportamientos mediante el archivo de preferencias disponible en /etc/streaming/streamingserver.xml. Este archivo permite tanto habilitar o deshabilitar un módulo como modificar los parámetros de éste. De este modo resulta más cómodo modificar el archivo de preferencias, en lugar de variar los parámetros de manera local en el archivo cpp y recompilar el servidor. Por comodidad resulta interesante utilizar el mismo archivo de configuración en lugar de generar otro para cada módulo. A continuación se comenta como añadir preferencias para un módulo particular en el archivo común y como interactuar con él desde el propio programa:

Para añadir preferencias destinadas a un módulo en particular es suficiente con definir un nuevo tag del tipo MODULE que incluya el nombre del nuevo módulo y a continuación el nombre, tipo y valor de cada una de las preferencias relativas a éste. Ejemplo:

```
<MODULE NAME="Selector" >
  <PREF NAME="selector_enabled" TYPE="Bool16" >true</PREF>
  <PREF NAME="wap_method" TYPE="Bool16" >true</PREF>
</MODULE>
```

En este caso se tiene el módulo llamado Selector (el cual informa del nombre al servidor en su función de registro) y se desea agregar dos preferencias para él. En primer lugar se define el *tag* con el nombre del módulo y a continuación se añaden dos nuevos *tags* adicionales. En esta ocasión son del tipo preferencia e incluyen los campos PREF NAME y TYPE. El nombre es útil para obtener la información desde el código del propio módulo y el tipo indica la clase de datos que almacenará. En este ejemplo se trabaja con dos boleanos, el primero activa el funcionamiento del módulo y el segundo activa una clase de selección en particular. Además de los atributos de la preferencia, se debe indicar el valor actual de ésta.

Para permitir la lectura de preferencias por parte del módulo, es imprescindible añadirle el rol de *RereadPrefs* como se muestra a continuación:

```
QTSS_Error  SelectorDispatch(QTSS_Role inRole, QTSS_RoleParamPtr inParamBlock)
{
    switch (inRole)
    {
        case QTSS_Register_Role:
            return Register(&inParamBlock->regParams);
        case QTSS_Initialize_Role:
            return Initialize(&inParamBlock->initParams);
        case QTSS_RTSPRoute_Role:
              ProcessRTSPRequest(&inParamBlock->rtspRequestParams);
        case QTSS_RereadPrefs_Role:
            return RereadPrefs();
    }
    return QTSS_NoErr;
}
```

A continuación un ejemplo de la función de lectura de preferencias:

```
QTSS_Error RereadPrefs()
{
QTSSModuleUtils::GetAttribute(sPrefs, "selector_enabled",  qtssAttrDataTypeBool16, &sModuleEnabled,
&sDefaultModuleEnabled, sizeof(sDefaultModuleEnabled));

QTSSModuleUtils::GetAttribute(sPrefs, "wap_method", qtssAttrDataTypeBool16, &sWapEnabled,
&sDefaultWapEnabled, sizeof(sDefaultWapEnabled));

return QTSS_NoErr;
}
```

En este ejemplo se utiliza la función *GetAttribute*, para obtener las preferencias relativas a cada caso. Estas preferencias se almacenan en las diferentes variables definidas con anterioridad:

```
// Default values for preferences
static Bool16           sDefaultModuleEnabled   = true;
static Bool16           sDefaultWapEnabled      = true;

// Current values for preferences
static Bool16           sModuleEnabled          = true;
static Bool16           sWapEnabled             = true;
```

En caso de no obtener un valor correcto se trabajará con los valores por defecto.

5.6. Flujo de trabajo en el servidor

En este apartado se pretende comentar brevemente cuál es el flujo de trabajo del servidor respecto a las llamadas que realiza a módulos en una transmisión normal.

Tras la instalación del servidor, el primer paso a seguir es proceder a su ejecución, inicializando así los módulos, tanto estáticos como dinámicos. Las inicializaciones dependen de los roles asignados a los diferentes módulos.

Una vez inicializado el servidor se entra en el proceso de respuestas a peticiones y envío de paquetes RTP. A partir de este momento, el servidor espera la llegada de al menos una petición RTSP, la cual generará la llamada a los módulos de cada uno de los roles de manera jerárquica, hasta obtener la respuesta para el cliente. Una vez establecida la conexión de los diferentes flujos RTP (ver Figura 12) el servidor entra en un estado en el que, mientras tiene paquetes RTP a enviar, continúa con el flujo. Durante este periodo, acepta más peticiones RTSP tanto del mismo como de diferentes clientes, las cuales procesa del mismo modo que anteriormente. También tiene la capacidad de atender mensajes RTCP, los cuales se procesan en los módulos que tengan indicado el rol pertinente.

En caso de que se detenga la ejecución del servidor, se entra en una fase en la que el servidor se desactiva como tal, a continuación se llama a los módulos con rol *shutdown*, y finalmente se finaliza la ejecución de éste. El flujo de trabajo puede apreciarse en la siguiente figura:

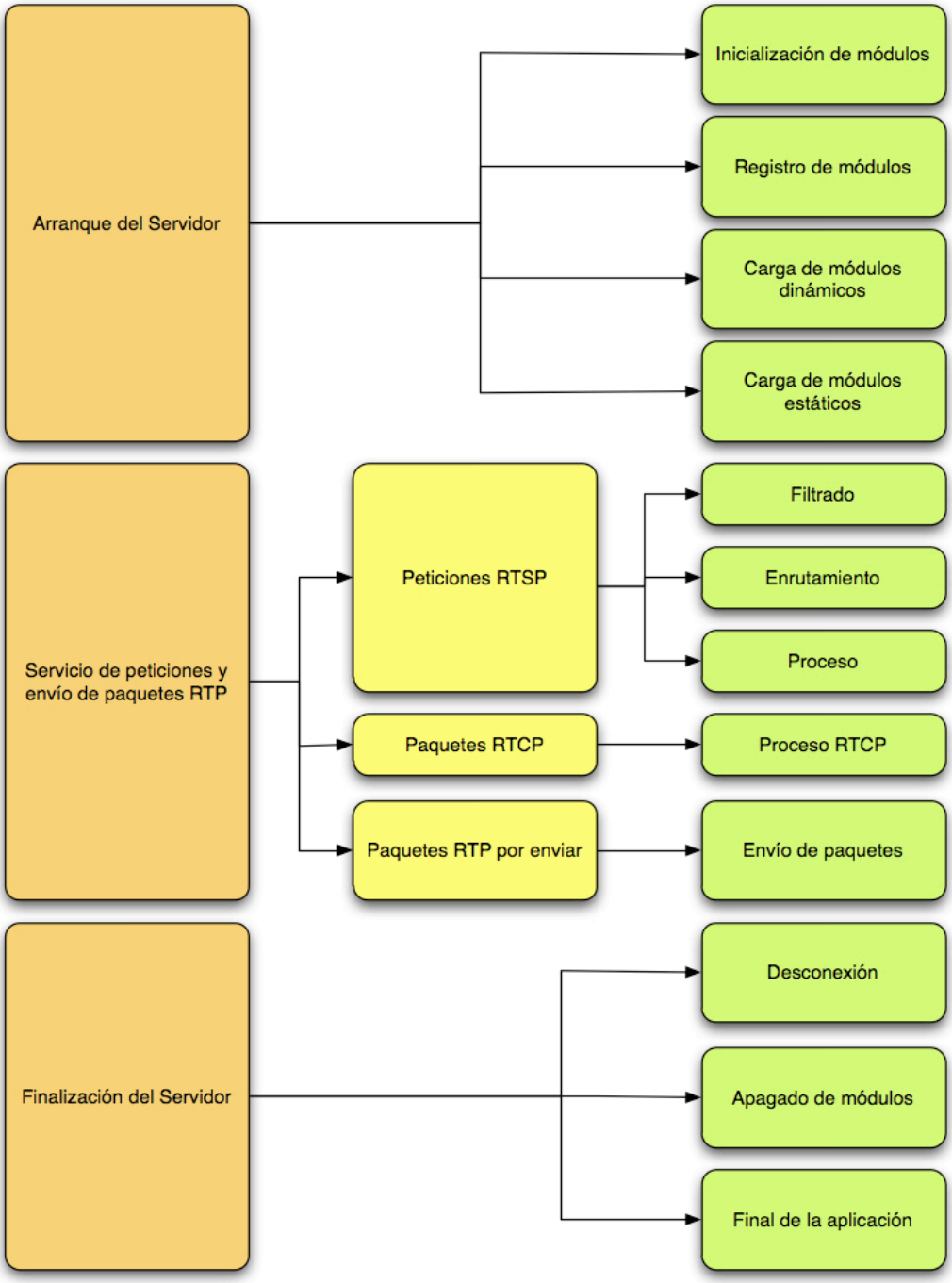

Figura 12: Flujo de trabajo del servidor

6.Implementaciones para Darwin Streaming Server

6.1. Implementación de un módulo para la selección de flujo al inicio de una transmisión

Tal y como se presentó en apartados anteriores, existen gran variedad de terminales móviles, cada uno de ellos con diferentes características y capacidades. La adaptación de contenidos a las capacidades de los terminales es un punto fundamental a la hora de transmitir multimedia; un usuario que utiliza un dispositivo reciente puede sentirse frustrado a la hora de recibir contenido muy por debajo de las capacidades de su terminal mientras que, en el caso contrario, se corre el peligro de verse imposibilitada la reproducción de los contenidos por obsolescencia del dispositivo. Además de los factores relacionados con la experiencia de usuario, hay otros relativos a la utilización de recursos tanto en la red como en la máquina que ejerce como servidor; es interesante evitar el malgasto de recursos producido al ofrecer un contenido por encima de las posibilidades del cliente, de este modo tanto el propio usuario como el colectivo de la red incrementarán su calidad de experiencia. Además de estas características propias del terminal pueden utilizarse otras variables para la toma de decisiones tales como el estado de la red respecto a carga, usuarios, etcétera.

Para maximizar la precisión a la hora de distribuir contenidos multimedia se ha implementado un módulo para seleccionarlos en función del perfil del usuario (Sección 3.5). Para ello se dispone de un sistema detector de capacidades que utiliza el perfil de usuario cuando éste se conecta para añadirle a una base de datos describiendo sus características de terminal. Cuando el servidor de vídeo reciba una petición por parte de un cliente accederá a dicha base de datos para entregarle el vídeo que mejor se ajuste a su perfil. Una de las opciones para adaptar los contenidos ofrecidos a los usuarios es modificar el recurso ofrecido a partir de una petición de éste. Para ello se precisa, por una parte interceptar la petición de reproducción de un archivo, posteriormente identificar el terminal y finalmente ofrecerle una respuesta modificada según convenga.

Como se explicó en la Sección 3.8, el flujo que sigue una petición RTSP se compone típicamente de una secuencia OPTIONS, DESCRIBE, SETUP y PLAY. La petición en la que se identifica el recurso a la que se desea acceder es DESCRIBE, en la que, además, se obtiene información de dicho recurso. Esta es la información que el servidor deberá proveer modificada al cliente, incluyendo la ruta que mejor se ajuste a sus características. Para ello es preciso interceptar la petición antes de que sea procesada por otros módulos del servidor. De hecho, la misión fundamental de este módulo será interceptar y modificar la petición para que posteriormente sea procesada por el servidor de la manera habitual, de esta forma se ahorra tener que realizar todo el procesado de la petición y generar la respuesta para el cliente. Todo el código relativo a este ejemplo puede encontrarse como anexo.

6.1.1. Roles y filtrado de peticiones

Como se explicó en el apartado 5.2, el rol asignado al módulo indicará el orden en el que el servidor lo ejecutará. Con la intención de ejecutarse antes que los módulos que procesan la petición y tener acceso al objeto *RTSPRequest* que define a ésta, se asigna al módulo el rol de *RTSPRoute*. De este modo, cada vez que el cliente lance una petición RTSP, ésta será procesada en primer lugar por el módulo implementado. En este caso particular, resulta interesante interceptar las peticiones

DESCRIBE e ignorar el resto, para ello será necesario implementar una lógica basada en casos que descarte cualquier otro tipo de petición. Las peticiones pueden ser identificadas utilizando el atributo *qtssRTSPReqMethod* del objeto *RTSPRequest* cuyos valores pueden ser, ente otros *qtssDescribeMethod, qtssSetupMethod,* etcétera. Ejemplo de obtención y procesado del tipo de petición:

```
...
QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqMethod, 0, (void**)&theMethod, &theLen);

    switch (*theMethod)
    {
       case qtssDescribeMethod:
        {
         ...
         break;}
    default:
         break;
    }
```

En el ejemplo anterior se obtiene el valor del atributo *qtssRTSPReqMethod* en tipo puntero, proveniente de *inRTSPRequest,* que se trata de la petición del usuario que el servidor nos pasa como parámetro dentro del elemento *inParamBlock*. Se almacenan el valor y la longitud de éste en sendas variables y se comprueba el tipo de petición. La función de obtención de valor puede retornar un error o valores inconexos entre longitud y valor, por lo que es recomendable asegurarse de que el valor retornado es, al menos, distinto de error y válido desde el punto de vista de que el valor recibido y su longitud coinciden y son no nulos.

En este ejemplo se realiza un *switch* del valor devuelto del método, si se trata de un DESCRIBE se ejecutará una serie de código (representado por los puntos suspensivos), mientras que en otro caso se saldrá de la evaluación. Esto se debe a que el primer mensaje por parte del cliente en el que se identifica un recurso concreto al que se desea acceder es precisamente el DESCRIBE. En él, el cliente solicita información sobre las pistas implicadas en un archivo, la intención del módulo es devolverle la información del archivo que interese al servidor sin que el cliente identifique el cambio.

6.1.2. Formato de archivo

Para este ejemplo, se considera que la estructura de los archivos sigue la forma *archivo_xx.ext* siendo *xx* el nivel de calidad del vídeo (*hi*, *me*. o *lo*) y *ext* la extensión de éste, pudiendo ser *.3gp* o *.mp4*. De esta forma se servirá un vídeo de una de las tres calidades con la extensión deseada. Los criterios a seguir a la hora de escoger uno u otro nivel de calidad pueden ser múltiples (resolución, usuarios conectados, throughput...), aunque en este caso se ha seguido la lógica de resolución de pantalla y soporte para audio AAC.

Dado que desde los terminales se controla el contenido web servido al cliente, es necesario que todos los enlaces referentes al servidor de streaming cumplan la estructura mencionada anteriormente, de lo contrario el módulo no la identificará como una petición bien formada. Todas las peticiones por defecto deberían solicitar el recurso de extensión 3GP y menor calidad, es decir, *rtsp://recurso_lo.3gp*.

6.1.3. Interacción con la base de datos

Una de las finalidades fundamentales del módulo es la adaptación dinámica a la hora de seleccionar el contenido siguiendo diferentes criterios. En este caso el criterio que se ha seguido es la evaluación de las diferentes capacidades del terminal a la hora de soportar diferentes tipos de archivo. Para ello se interactuará con la base de datos encargada de almacenar las capacidades de éstos. El proceso a seguir es definir una conexión con ésta y realizar las diferentes consultas relativas al cliente. A continuación se presenta un ejemplo de lógica de decisión.

Para comunicar el módulo con la base de datos *postgres*, se han utilizado las librerías *pqxx*[36]. Esta serie de librerías permite la interacción de programas *C++* con bases de datos *postgres*. En este caso son necesarias las librerías de desarrollo y, para no interferir con el software instalado en las plataformas, en lugar de realizar la instalación desde el gestor de paquetes (que presentaba problemas de dependencias) se ha utilizado un paquete precompilado en una máquina externa a la que se le agregó una nueva carpeta de librerías. Las librerías vienen incluidas en el paquete precompilado para la instalación, nótese que el archivo *Makefile* (disponible como anexo) sufre una modificación para incluir el nuevo directorio de las librerías. Para permitir la interacción del módulo con la BBDD es necesario incluir una serie de librerías y añadir espacios de nombres de la siguiente manera:

```
#include <pqxx/pqxx>
using namespace std;
using namespace pqxx;
```

A partir de ese momento, existe la posibilidad de definir una conexión, que se utilizará para las consultas. Dicha conexión recibe como parámetro una tira de caracteres que definen particularidades del servidor de BBDD tales como dirección, nombre de usuario, contraseña, etcétera. En caso de resultar imposible la conexión se devuelve una excepción que es tratada por el módulo, en ese caso se entregará al cliente el flujo por defecto. Este parámetro tendrá un valor por defecto pese a que será configurable como variable externa desde el archivo de configuración. Es importante destacar que, en este caso, se almacenarán en claro datos como la máquina destino, el nombre de usuario o la contraseña. Se considera que en futuras versiones, este parámetro puede ser eliminado ya que, en general, el acceso tendrá lugar en la máquina local, pudiendo dejar la variable con un valor fijo. En caso de que la máquina destino no responda (no sea alcanzable por la red), el programa puede quedar atascado esperando la respuesta, por ello es importante asegurarse de que la ruta a la base de datos (en caso de ser remota) es correcta y alcanzable. En caso de respuesta negativa, se obviará la conexión y se servirá el flujo de menor categoría. A continuación un ejemplo de conexión:

```
connection C("hostaddr=123.45.67.89 dbname=mydatabase user=myuser password=mypassword");
```

Con la conexión definida se añade una variable *work* y se procede con normalidad a la serie de consultas hacia la base de datos. En este caso particular, se desea conocer en primer lugar si el cliente que solicita el vídeo tiene un perfil almacenado en dicha BBDD, para ello se consulta si su dirección IP está almacenada. Esta consulta requiere dos pasos: la obtención de la dirección IP del cliente (provista

[36] Pqxx.org

en la petición RTSP) y la consulta en sí. A continuación se muestra un ejemplo de como obtener la dirección y, posteriormente, ejecutar la consulta:

```
(void)QTSS_GetValueAsString(inParamBlock->inRTSPSession,
qtssRTSPSesRemoteAddrStr, 0, (char**)&theAddress);
```

Se puede apreciar como el primer comando *GetValueAsString* propio del servidor *DSS* se ejecuta contra el parámetro *qtssRTSPSesRemoteAddrStr* del objeto *inRTSPSession* del bloque de entrada. Ésto se debe a que dentro de los parámetros de entrada del rol utilizado se incluye la sesión RTSP, dentro de la cual se encuentra el parámetro que nos proporciona la dirección IP del cliente en formato decimal con puntos. Obtenida esta dirección se puede realizar la consulta al servidor pasándole el parámetro de dirección IP como un *cast* al tipo de *string* definido por el lenguaje *PGSTD* ya que éste es el parámetro que espera para concatenar con el texto introducido.

```
result R = W.exec("SELECT id FROM summary WHERE id_dev='"+ (PGSTD::string)theAddress +"'");
```

El tamaño de la variable resultado nos indicará el número de coincidencias obtenidas. En este caso debería ser nula o igual a uno ya que nunca se tendrán dos direcciones IP almacenadas en la base de datos. Si el resultado es satisfactorio se realizarán el resto de consultas y, se procederá a aplicar la lógica de decisión. Todos los pasos y consultas están disponibles en el código fuente del módulo.

6.1.4. Lógica de decisión

Tal y como se comentó anteriormente, la lógica de decisión de este módulo es relativamente sencilla. Tras realizar las consultas necesarias a la base de datos, se dispone de una serie de variables booleanas que indican las capacidades del terminal. En un primer escenario, estas variables se refieren a alta resolución, soporte MP4 y soporte AAC; aunque podrían incluir muchas otras. En función de las capacidades del terminal se modifica la petición que éste realizaba inicialmente al servidor, sustituyéndola por una más acorde. De éste modo, por ejemplo, se considera que si el terminal es capaz de soportar audio AAC y alta resolución obtendrá el archivo de mayor calidad, si sólo dispone de resolución alta optará a la calidad media y, en otro caso, deberá conformarse con la calidad más baja. En cualquier caso el archivo se le servirá en formato 3GP salvo que indique compatibilidad con MP4. Para la evaluación de la capacidad en términos de resolución, se utiliza el ancho de pantalla en píxeles ya que se considera que dicho valor es más restrictivo que el alto. Para definir a partir de qué alto se considera alta resolución, se utiliza un parámetro externo definido como preferencia en el archivo destinado a ello. Este parámetro tiene un valor por defecto y se considerará que todas las pantallas de mayor número de píxeles horizontales tiene alta resolución. Además de la resolución de pantalla, existe un parámetro denominado resolución de renderización el cual no se utiliza ya que es dinámico. Cabe destacar que en este ejemplo sólo se filtran peticiones que apunten a archivos con extensiones 3GP y MP4 ya que se considera que otras extensiones (listas de reproducción) no son susceptibles de ser modificados.

Para modificar la petición del cliente, se debe interceptar ésta en primer lugar utilizando el rol indicado. Posteriormente se extrae el recurso al cual iba dirigida mediante:

```
(void)QTSS_GetValueAsString(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0,
(char**)&thePath);
```

La cual devuelve un puntero al recurso solicitado del tipo */recurso_xx.ext*. Posteriormente se sitúa un puntero en el guión bajo y se modifica, si fuera necesario, la calidad del vídeo para finalmente escoger la extensión. En el siguiente ejemplo, la variable thePtr apunta a la dirección de memoria del guión bajo si existe o a *NULL* en caso contrario. En este caso se evalúa la posibilidad de escoger un vídeo de alta calidad:

```
if (thePtr!=NULL && CliAAC==true && CliHiRes==true)
{
        *thePtr++='_';
        *thePtr++='h';
        *thePtr++='i';
}
```

Del mismo modo se realizan el resto de evaluaciones sobre calidad y, posteriormente, sobre tipo de archivo soportado.

Una vez modificado el recurso a acceder de la manera deseada, se escribe éste en los parámetros que recibirá el módulo encargado de realizar la descripción del vídeo para que describa el nuevo recurso en lugar del solicitado inicialmente por el cliente. Para ello se utiliza:

```
theErr = QTSS_SetValue(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0, thePath,
strlen(thePath));
```

Todo este proceso es transparente, tanto para el cliente como para el servidor ya que, por una parte, el servidor recibe una petición correcta, bien formada y apuntando a un recurso que existe, con lo cual realiza el proceso de descripción del vídeo de manera natural. Por otra parte, el cliente, al solicitar un recurso de vídeo, desconoce el paradero real de éste y, por lo tanto, acepta la descripción de un recurso que se encuentra físicamente en otro lugar o tiene otro nombre. Se puede apreciar un ejemplo de lógica de decisión en la Figura 13.

6.1.5. Alternativas

En este punto se ha presentado un módulo que selecciona un recurso de vídeo apropiado a las características del terminal. En el ejemplo particular se utilizan capacidades como el tamaño de pantalla, el soporte de audio AAC, de vídeo MP4, etcétera. Este módulo únicamente pretende ser una guía y demostración de las capacidades de interacción entre servidor de *streaming* y base de datos, pudiendo basarse la lógica en otros algoritmos o parámetros.

Por otra parte, además de un sistema selector basado en características del terminal, se podría implementar uno basado en condiciones del servidor, es decir, dependiente del número de clientes conectados, del *throughput* servido, o incluso una combinación entre ambos.

Además de utilizar la dirección IP como identificador, se puede tratar de obtener el perfil wap del cliente. Este perfil es una URL facilitada por el cliente al servidor en el momento de realizar la petición RTSP. En la URL se incluyen las capacidades del terminal.

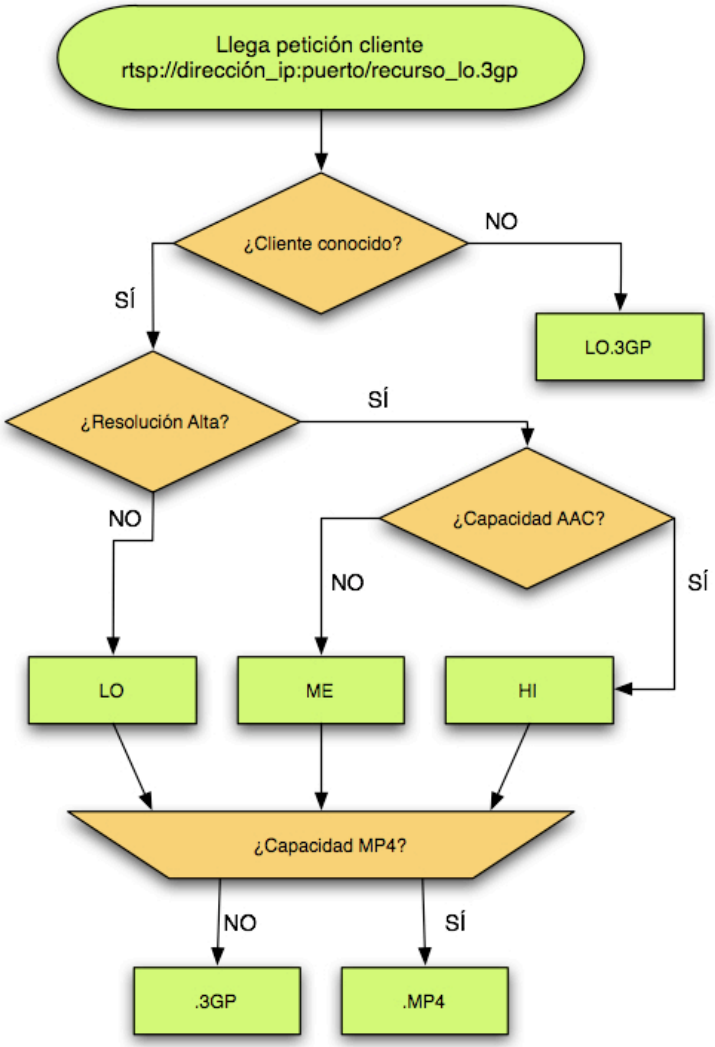

Figura 13: Flujo decisiones

6.2. Modificación del servidor para permitir la adaptación de tasa durante la reproducción de contenidos

Uno de los puntos propuestos durante el segundo proyecto *InfoPoints* es el estudio de un sistema de adaptación de tasa en tiempo real durante la transferencia de vídeo a terminales móviles. Con ello se pretende paliar la variación del comportamiento de las redes inalámbricas compartidas debido al incremento de usuarios y movimiento de éstos, como es el caso en los puntos de acceso utilizados.

Esta adaptación de tasa debería tener como base los sistemas implementados con anterioridad y cumplir con los mismos requisitos. De este modo, en el desarrollo llevado a cabo en este apartado, se considera la implementación de la solución en una plataforma *Darwin Streaming* sobre un punto de acceso *InfoPoint*; aunque la solución de software propuesta es independiente de la plataforma. Se considera también que los principales destinatarios de este sistema de adaptación son usuarios de terminales móviles; principalmente teléfonos. Por ello deben tenerse en cuenta las restricciones existentes en este tipo de dispositivos.

La adaptación llevada a cabo por el servidor debe ser en todo momento compatible con los diferentes RFC y estándares propuestos tanto a nivel IETF como 3GPP. Debe procurarse, en la medida de lo posible, que las modificaciones realizadas en el servidor afecten lo mínimo al núcleo del mismo; del mismo modo es recomendable que la solución sea lo más flexible y adaptable posible.

En este apartado se comentará el estado del arte de las soluciones basadas en adaptación de contenidos tanto a nivel 3GP y sus *releases* como a nivel de implementación. Posteriormente se indican las posibilidades del servidor y los requisitos; finalmente se plantea una implementación de la solución.

6.2.1. Estado de la técnica

Tal y como se comentó en apartados anteriores, el sistema de evolución de 3GP se basa en *releases*. Actualmente se encuentra desplegada la sexta versión que incluye mecanismos para permitir la adaptación de contenidos para *streaming* de vídeo. Dentro de dichos mecanismos se incluye: Información sobre calidad de experiencia o QoE, definición de archivos basados en grupos de pistas alternativas, nuevos atributos a nivel SDP para intercambiar mensajes entre clientes y servidores, un nuevo sistema de *feedback* para que el cliente pueda informar sobre el estado de su *buffer* y para evaluar el estado del enlace.

Los archivos basados en grupos de pistas alternativas no son más que archivos en los que las pistas de vídeo o audio van agrupadas entre sí. De este modo, dentro de una caja ISO (ver 3.4.2) se encuentran diferentes pistas marcadas como alternativas como puede verse en la Figura 14; de este modo el servidor debería ofrecerlas como tal de modo que el cliente comprendiera que la reproducción de cada una de ellas es excluyente de las demás. Cada una de las pistas de vídeo contiene el mismo material utilizando diferente tasa de codificación.

Llegados a este punto, el caso ideal es aquel en el que el cliente tiene la capacidad para tomar la decisión de cuál es la pista que le resulta más interesante (cosa que no sucede en el caso de los terminales móviles).

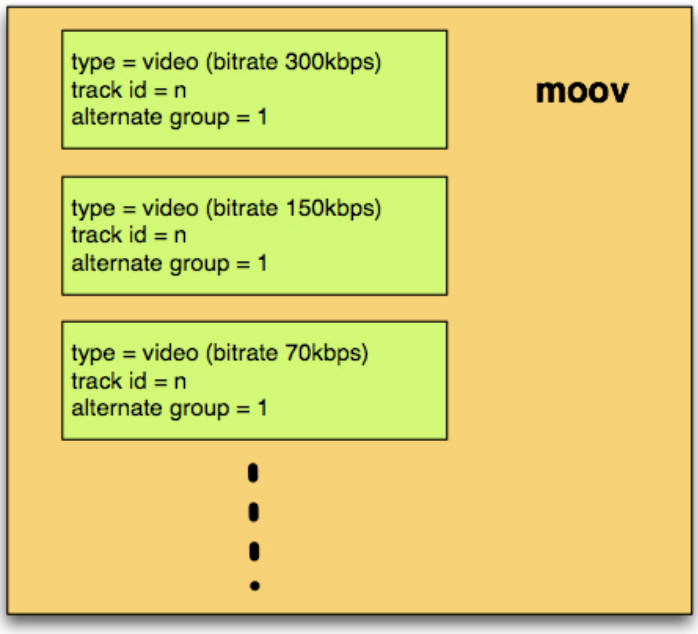

Figura 14: Diferentes pistas alternativas en una película

A la hora de responder a un mensaje de tipo DESCRIBE lanzado por el cliente, un servidor compatible con la *Release 6* de 3GPP, debería lanzar una respuesta SDP incluyendo, entre otra información, las diferentes agrupaciones posibles (generalmente una pista de audio y una de vídeo con sus respectivas *hint tracks*) y el ancho de banda utilizado por cada una de ellas. El objetivo principal de este sistema de pistas alternativas es facilitar el proceso de escoger un grupo de los posibles en función de las circunstancias del terminal. En ningún momento se define un mecanismo para el intercambio entre pistas durante la reproducción de las mismas.

Dentro del apartado de funcionalidades adicionales en la información SDP se pueden encontrar algunos campos nuevos; entre estos campos destacan aquellos relativos a los grupos de pistas alternativas comentadas anteriormente "*alt*", "*alt-default-id*" y "*alt-group*", los campos que indican compatibilidad con el sistema de adaptación de tasa "*3GPP-Adaptation-Support*" y los relativos a métricas adicionales de calidad de experiencia "*3GPP-QoE-Metrics*". Todos estos mensajes deben estar implementados en clientes y servidores compatibles con la sexta versión de 3GPP. Puede encontrarse información adicional en el documento correspondiente.

Además de toda la información proporcionada en los informes tanto de servidor como de cliente vía RTCP, la *release 6* de 3GPP define un nuevo sistema de información para obtener *feedback* del *buffer* de cliente. Esta información forma parte de los paquetes RTCP APP utilizando los campos específicos para cada aplicación. El nombre concreto para este tipo de paquetes es PSS0, en los campos propios de la aplicación se incluye, entre otros, información relativa al buffer disponible en el cliente en el momento de lanzar el informe. De esta manera es posible tener un control más exhaustivo del mismo. De nuevo las especificaciones del paquete pueden encontrarse en el documento pertinente.

Es importante destacar que cualquier sistema de adaptación de tasa es dependiente de la plataforma utilizada, es decir, el estándar no define ningún sistema en particular lo que supone que cada cliente y servidor deben coincidir en el mecanismo a utilizar.

6.2.2. Posibilidades actuales del servidor

En el momento de la redacción de este informe se encuentra disponible la versión 6.0.3 de *Darwin Streaming Server*. Pese a que inicialmente solo ha sido lanzada en versión Mac OSX, existen una serie de parches que permiten su correcta instalación y ejecución en sistemas Linux. Todas las modificaciones realizadas actualmente sobre el servidor DSS tienen como base dicha versión 6.0.3.

La principal novedad de la última versión es la compatibilidad con la *Release 6* de 3GPP. Gracias a ello se han implementado todo el sistema de intercambio de mensajes SDP y evaluación de estado de *buffer* de cliente. Estas herramientas permiten tener un mejor control del estado actual de recepción de vídeo en el cliente y por lo tanto, la posibilidad de desarrollar una serie de algoritmos más precisos para la adaptación de tasa.

En la transmisión de vídeo existen tres tipos de tramas o marcos; en primer lugar existen las tramas I o intra-tramas las cuales no toman como referencia a ninguna otra y proveen toda la información para su reproducción en sí mismas; en segundo lugar existen las tramas P o predictivas, en este caso se utilizan como referencia las tramas I, esto implica una menor carga de información pero una menor tolerancia a pérdidas. Finalmente las tramas B o bidireccionales referencian tanto a tramas I como P dando lugar a las tramas con menos peso pero más sensibles a errores. La pérdida de una trama B únicamente afecta a sí misma, la pérdida de una P afecta a varias B y la pérdida de una I afecta tanto a P como B. Puede observarse un gráfico del ancho de banda consumido por las diferentes tramas en la Figura 15.

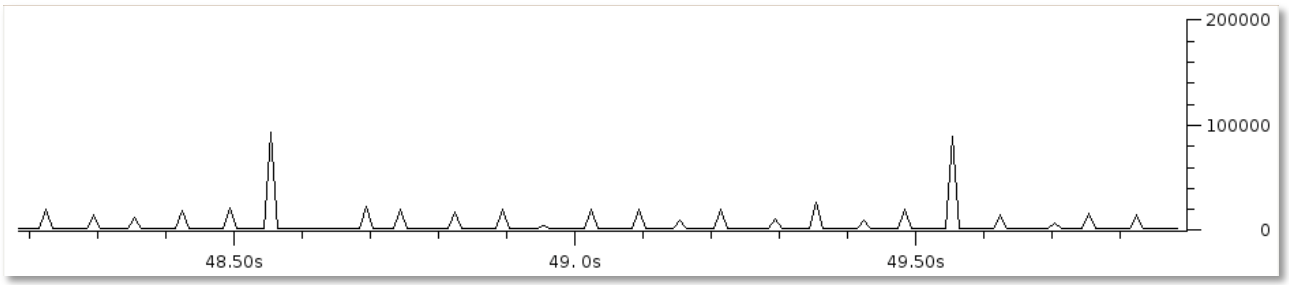

Figura 15: Ancho de banda utilizando en la transmisión de vídeo.

Darwin Streaming Server ofrece actualmente un algoritmo de adaptación de flujo basado en los informes generados por el cliente[37]. Este algoritmo permite reducir el ancho de banda utilizado en el flujo suprimiendo ciertos marcos del vídeo que se está transmitiendo. Este algoritmo establece una serie de niveles (típicamente 6) e inicia la transmisión del vídeo considerando el nivel de mayor calidad. En función de los informes del cliente se va ajustando un nivel diferente de calidad. Estos

[37] Véase apartado protocolos de comunicación.

niveles engloban, desde la transmisión de todos los marcos del vídeo hasta la transmisión de únicamente las tramas I; los niveles intermedios contemplan la supresión de diferentes porcentajes de tramas B y P.

El resultado obtenido de cara al espectador es un vídeo con menos fluidez de movimiento debido a la pérdida de tramas mientras que la calidad de imagen se conserva respecto al original. Desde el punto de vista de ancho de banda, la eliminación de un pequeño número de tramas no proporciona una excesiva reducción ya que, como puede observarse en la Figura 15 el ancho de banda promedio con y sin algunas tramas P y B no es significativamente diferente. La diferencia significativa viene dada en el momento en el que se elimina un gran número de éstas.

Este sistema ayuda a reducir el ancho de banda consumido por cada uno de los clientes. El otro beneficio obtenido con esta técnica es el hecho de que el cliente mantiene durante más tiempo la conectividad y observa una caída progresiva de la fluidez de la reproducción lo que permite que, en caso de producirse por un desplazamiento lejos del punto de acceso, éste vuelva a aproximarse; del mismo modo pueden evitarse colisiones y pérdidas al eliminar ciertos tiempos de transmisión de paquetes aunque la mejora no es sustancial.

Uno de los problemas fundamentales del algoritmo es su poca agresividad ante pérdidas de paquetes; esto se debe fundamentalmente a que está ideado para trabajar en entornos más estables. En la versión anterior del servidor, la cual únicamente permitía interacción con mensajes RTCP normales, se implementó un algoritmo mejorado al disponible por defecto resultando en un mejor comportamiento del mismo. Se intentó simular el comportamiento de control de TCP, proporcionando un incremento de calidad lento y un decremento agresivo.

La principal desventaja desde el punto de vista del usuario es la dramática pérdida de fluidez en entornos con muchas pérdidas, aunque estáticamente la calidad de la imagen sigue siendo buena, la transición entre tramas es muy brusca. Desde este punto de vista es interesante implementar un algoritmo que ofrezca mayor fluidez sacrificando algo de calidad.

La conclusión que puede extraerse del sistema de adaptación es que, a pesar de modificar la tasa de transmisión para clientes compatibles evitando así algunas pérdidas y colisiones, la calidad de experiencia para el usuario sigue sin ser excesivamente satisfactoria ya que el algoritmo no deja de ser un sistema de pérdidas controladas. Teniendo en cuenta el estado de la técnica disponible respecto a agrupación de pistas, informes de cliente, etcétera, se decide implementar un sistema de intercambio de pistas con la finalidad de mejorar esa experiencia de usuario transmitiendo el mismo contenido con diferente calidad en lugar del contenido a máxima calidad con pérdidas.

6.2.3. Algoritmo utilizado originalmente

En la solución presentada anteriormente se utiliza un algoritmo basado en cuatro factores fundamentales tal y como puede apreciarse en la Figura 19. Dichos factores son utilización del *buffer* disponible, retardo en la reproducción, RTT y número de informes consecutivos indicando pérdidas. Toda esta información viene dada en los paquetes NADU y únicamente es procesada en aquellas sesiones donde el cliente indica compatibilidad con la *Release 6* de 3GPP.

El algoritmo efectúa dos modificaciones, una sobre el nivel de calidad y otra sobre el tiempo de transmisión. Para entender el algoritmo es interesante introducir una serie de elementos:

- Nivel de calidad: Inicialmente definido en una escala de 6 elementos, el nivel de calidad indica el ancho de banda disponible para cada una de las pistas de un elemento multimedia de forma independiente. En la solución inicial, este nivel de calidad está ligado con una proporción de tramas desechadas; el nivel de calidad máxima transmite todas las tramas, el medio descarta por ejemplo el 70% de las tramas B y el mínimo nivel descarta todas las B y todas las P. Generalmente el nivel de calidad 1 implica máxima calidad[38] y el nivel 6 mínima calidad[39]. Estos valores pueden variar en función del número de niveles definidos.[40]

- Ajuste o gestión temporal: Se puede entender el ajuste temporal como la gestión del rango de tiempo para el cual los paquetes que pertenecen a dicho rango pueden ser enviados, es decir, varía el instante en el que cada paquete puede ser transmitido, incrementando o decrementando la tasa en un instante determinado. La gestión temporal comprende la modificación de dicho rango de forma que el servidor sea capaz de enviar más o menos paquetes según convenga, permitiendo así aligerar la carga al cliente o tratar de incrementar la información almacenada en su buffer.

  Por ejemplo, un cliente en peligro de *buffer underrun*[41], necesita un incremento en dicho rango, para que el servidor transmita más paquetes en menos tiempo y éste pueda acomodarlos en su buffer, reduciendo el peligro.

  La gestión temporal se basa típicamente en modificar una variable conocida como tiempo de transmisión de paquete teniendo en cuenta el instante actual, el último instante de transmisión y el incremento o decremento comentado anteriormente. Inicialmente el desfase u *offset* es nulo y se va incrementando o decrementando en función de la resolución del algoritmo. Típicamente en fases de *buffering* y necesidad de crecimiento se reduce a la mitad. A continuación en la Figura 16 se puede apreciar el resultado obtenido a partir de un sencillo ejemplo:

[38] En el algoritmo clásico, transmisión de todas las tramas I, P y B.

[39] En el algoritmo clásico, transmisión únicamente de las tramas I.

[40] Aunque el nivel de calidad es inversamente proporcional al número que le describe en este documento se considera incremento de calidad al aumento de ésta y no al nivel numérico propiamente dicho.

[41] Estado en el que la tasa de datos enviados es menor a la de consumidos y provoca que el cliente no tenga datos que procesar causando, en este caso particular, la desconexión del terminal móvil.

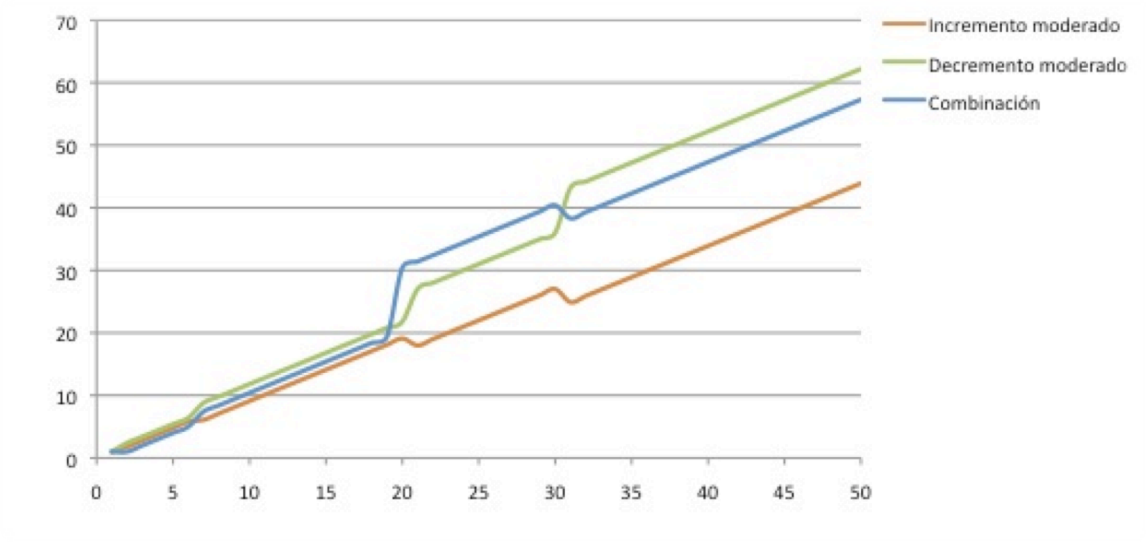

Figura 16: Paquetes enviados dependiendo del *offset*

Se consideran tres flujos diferentes durante la transmisión de 50 paquetes a razón de un paquete por unidad de tiempo. Inicialmente el *offset* de todos ellos es nulo con lo que, en un sistema sin variaciones. el tiempo de transmisión del quincuagésimo paquete debería coincidir con la unidad de tiempo 50. El gráfico del ejemplo muestra en el eje de ordenadas el tiempo de transmisión del paquete respectivo. En el ejemplo se considera un flujo que recibe incrementos moderados en la tasa de transmisión en los instantes de tiempo correspondientes a los paquetes 2, 7, 21 y 31, representado por el color anaranjado. Análogamente el color verdoso representa un caso opuesto de decremento moderado en los mismos instantes. Finalmente, la línea azul marca un comportamiento combinado que incluye un incremento agresivo[42] en el instante del paquete 2, un decremento moderado en el 7, un gran decremento en el 20 y un incremento moderado final en el 31.

Los ajustes en el offset a la hora de transmitir paquetes suelen realizarse en intervalos de tiempos definidos. En el algoritmo inicial proporcionado este ajuste se realiza cada 50 milisegundos. Debe considerarse que durante estos 50 milisegundos es muy probable que no exista una variación en el criterio de ajuste, es decir, que cada 50 milisegundos se aplique la condición, ya sea adelantando o retrasando el tiempo de transmisión de los paquetes.

Puede observare en el gráfico cómo el tiempo de transmisión de los paquetes varía según el incremento o decremento de la tasa recibidos. De este modo en el caso de incremento moderado, el quincuagésimo paquete podría ser transmitido en el instante de tiempo 43,9 mientras que en el caso de decremento, debería esperarse hasta el 62,2. La casuística combinada solo pretende mostrar la posibles variaciones en los tiempos de transmisión, demostrando que éste es siempre dependiente del *offset* inmediatamente anterior, es decir, un decremento anula la acción de un incremento en cuanto a *offset*[43] pero mantiene el retardo acumulado en la transmisión.

[42] Del 50%.

[43] Dejándolo a 0.

En los casos reales observaremos como el desfase durante la transmisión es prácticamente siempre negativo, es decir, los paquetes se pueden transmitir antes de su tiempo inicial. Esto se debe a que durante un tiempo prolongado[44] el servidor marca un incremento de tasa lo que propicia un tiempo extenso de expansión del rango temporal.

Lo realmente relevante de esta solución es la variación instantánea a corto plazo, es decir, que en caso de incrementos agresivos se envían varios paquetes rápidamente, mientras que en casos contrarios la transmisión es más pausada durante un breve tiempo aunque en ambos casos tiende a estabilizarse en la tasa de transmisión inicial; esto se debe a que la finalidad del sistema es únicamente realizar pequeñas correcciones a nivel del *buffer* del cliente.

En particular, puede apreciarse en la Figura 17 la evolución del retardo en un periodo de *buffering* típico. Al no tener ninguna otra instrucción que aplicar al algoritmo durante el llenado de éste, el retardo va decreciendo, propiciando el envío de más paquetes por unidad de tiempo. La cantidad de paquetes transmitidos en función del tiempo según la variación temporal anterior viene representada en la Figura 18.

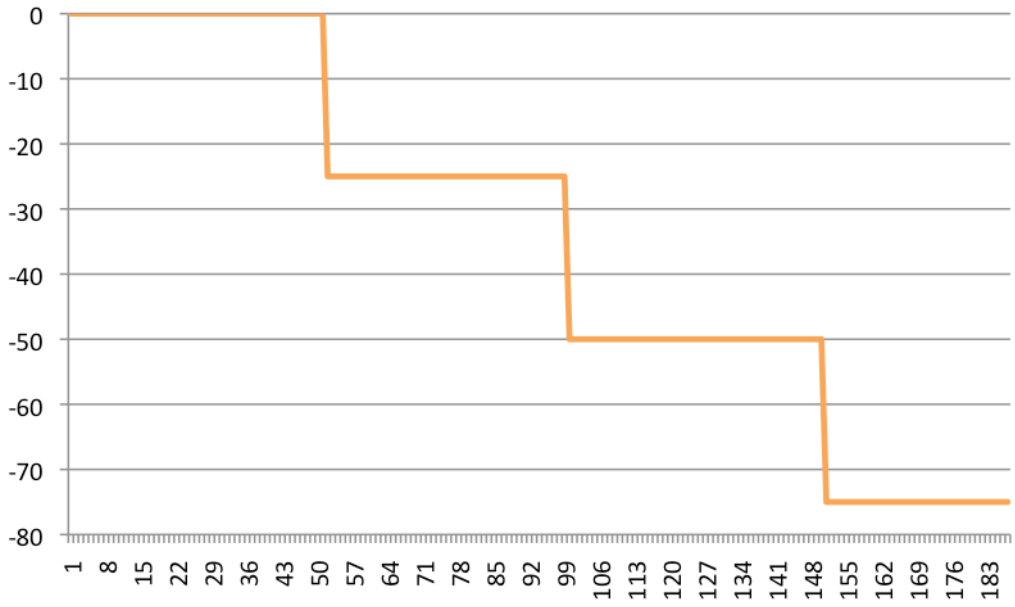

Figura 17: Evolución del retardo en un periodo de *buffering*

[44] Llenado de buffer o 25 segundos.

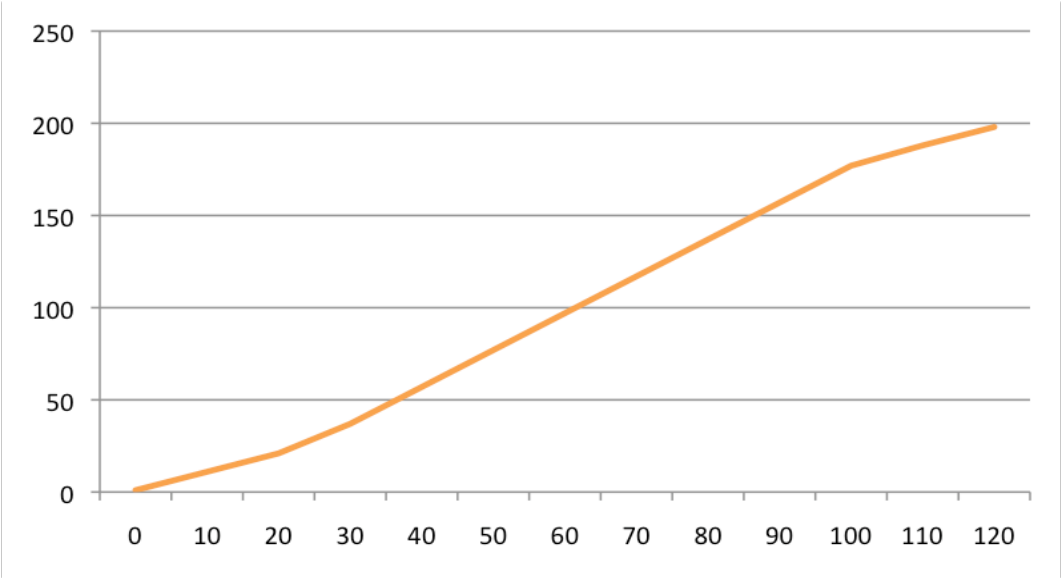

Figura 18: Paquetes enviados dependiendo del *offset*

- Funciones de incremento y decremento: Una vez tomadas las decisiones pertinentes por el algoritmo, existen una serie de funciones para manipular tanto el nivel de calidad como el rango de transmisión.

  En el primer caso, se trabaja fundamentalmente con funciones de incremento secuencial y decremento multiplicativo ya que se pretende utilizar sistemas agresivos con las pérdidas y de recuperación progresiva. No existen incrementos agresivos ni decrementos secuenciales.

  En el segundo caso sí existen los cuatro escenarios comentados anteriormente. Los incrementos y decrementos agresivos modifican en un 50% el tiempo de espera para los paquetes, es decir, se duplica o divide entre dos la velocidad; por otra parte los incrementos moderados aumentan un 10% la velocidad, mientras que los decrementos la reducen en un 20%.

  Éstas son las funciones utilizadas en los gráficos anteriores.

- Aumento de calidad: Cuando se hace referencia a un aumento o incremento de calidad, se considera que se incrementa el nivel de ésta, es decir, se tiende hacia un flujo de mayor calidad. Véase pie de página 40.

- Aumento de rango de transmisión: En este documento se considera un aumento en el rango de transmisión a un incremento del número de paquetes que pueden ser transmitidos en un rango de tiempo determinado, es decir, a la realización de una gestión temporal con la finalidad de incrementar la tasa independientemente de la calidad del flujo. Véase el ejemplo de la Figura 16.

Para implementar el algoritmo se utilizan una serie de variables tales como:

- *fAdjustSize:* Indica la acción a realizar respecto al tamaño[45], generalmente se define con una serie de valores entre los que se comprenden: *kAdjustDownDown, kAdjustDown, kNoChange, kAdjustUp, kAdjustUpUp.* Dichos valores representan respectivamente un decremento agresivo, moderado, nula modificación, incremento moderado y agresivo. Inicialmente esta variable se aplica a un entero que indica el nivel de calidad actual. Una variación negativa agresiva puede suponer una reducción a la mitad de calidad, mientras que una variación moderada suele implicar un incremento o decremento de la misma.

- *fAdjustTime:* Variable análoga a la anterior que realiza el mismo proceso pero respecto al tiempo de envío de los paquetes.

- *limitTargetDelay:* Variable booleana que indica si se lleva a cabo una limitación del retardo mínimo del que informa el cliente (ver siguiente variable).

- *adjustTargetDelay:* Retardo de *buffering* indicado por el cliente en el intercambio de información con el servidor. Esta cota mínima calcula el tiempo necesario para procesar los datos; se considera que una vez se encuentra por debajo de dicha cota, se corre riesgo de *buffer underrun*.

- *maxTargetDelayMilli:* Esta variable define un límite de retardo igual a la variable anterior. En caso de que *limitTargetDelay* sea cierto, *adjustTargetDelay* tomará el mínimo valor entre si mismo y *maxTargetDelayMilli.*

- *fBufferSize:* Tamaño de *buffer* indicado por el cliente según el estándar del 3GPP.

- *maxUsabeBufferSizeBytes:* Variable que indica la cantidad de Bytes útiles como *buffer*, generalmente viene definida por el tamaño máximo indicado por el cliente. Este valor deberá ir obligatoriamente acotado por el máximo valor referenciable de *buffer* disponible en los paquetes NADU. Este valor puede actualizarse a partir de la tasa de la película y el tiempo deseado a utilizar.

- *extraBufferMultipleInSeconds:* Factor de multiplicidad para aplicar a la variable *adjustTargetDelay,* suele variar entre 2 y 3.

- *maxUsableDelaySecs:* Resultado de la ecuación *extraBufferMultipleInSeconds * (adjustTargetDelay / 1000)*. Indica el tiempo útil en segundos para el retardo en el *buffer* del cliente.

- *mobieByteRate:* Tasa en Bytes[46] de la película reproducida.

- *bufferUsage:* Buffer utilizado definido por la diferencia entre su tamaño y el último informe de espacio disponible por parte del cliente. En caso de ser un dato erróneo (por ejemplo mayor al tamaño máximo) este se ajusta típicamente a la mitad del tamaño máximo.

- *bufferDelay:* Retardo existente en el *buffer* extraído de los informes del cliente.

[45] *Bitrate*.

[46] 8 bits.

De todas las variables presentadas, las más importantes para llevar a cabo el algoritmo son la tasa de utilización de *buffer* y el retardo del mismo, a partir de los cuales se construye el algoritmo de adaptación.

Inicialmente, en el algoritmo original existe un tiempo de llenado de *buffer* limitado, bien por una variable temporal, bien por un porcentaje de llenado, durante el cual no se ejecuta ningún tipo de adaptación en el contenido, es decir, hasta que no se alcance un llenado del buffer del 70% o hayan transcurrido más de un número determinado de segundos desde el inicio de la transmisión, se ignora la información generada por el cliente.

En caso de procesar los datos, la primera comprobación responde a la variable relativa al retardo de *buffer*. Inicialmente dicha variable toma un valor igual al tamaño máximo de *buffer*; se considera que el valor de dicha variable será válido si difiere del inicial, es decir, si el cliente ha proporcionado alguna información. SI efectivamente, el cliente informa del retardo de *buffer*, se ejecutarán una serie de comprobaciones diferentes a si no lo ha hecho.

Si el cliente no informa del retardo de *buffer*, todo el algoritmo está basado en el grado de ocupación del mismo. Puede observarse en la Figura 19 las diferentes comprobaciones realizadas y decisiones tomadas en función de la utilización del mismo. Las consideraciones tomadas en la elaboración del algoritmo son:

- Uso del buffer mayor al 90%: El cliente sufre riesgo de *buffer overflow*[47] con lo que se opta por decrementar agresivamente el nivel de calidad del flujo y reducir el rango de transmisión.

- Uso del buffer mayor al 80%: Idéntico al anterior, únicamente trata de reducir el rango temporal manteniendo intacta la calidad del flujo.

- Uso del buffer mayor al 70%: En los comentarios del código se presenta como una ocasión para tratar de evitar el *overflow* reduciendo tamaño y rango; no obstante, únicamente se reduce el rango, incrementándose la calidad del flujo.

- Uso del buffer mayor o igual al 50%: Se considera una zona oportuna para iniciar un incremento de tasa para ello se aumenta el nivel de calidad, pero se reduce el rango de transmisión, posiblemente para evitar un incremento radical en el uso del buffer tras el aumento de calidad[48].

- Uso del buffer mayor al 40%: Idéntico al caso anterior, en esta ocasión sí se incrementa el rango de transmisión además del nivel de calidad.

- Uso del buffer mayor al 30%: Se considera que se entra en peligro de *underflow* con lo que se incrementa agresivamente el rango de transmisión. El nivel de calidad, por otra parte, permanece intacto.

- Uso del buffer mayor al 20%: Igual al anterior, además del incremento agresivo del rango de transmisión se reduce la calidad.

[47] Estado en el que la tasa de datos enviados es mayor a la de consumidos y provoca que el cliente no tenga espacio en el que almacenar los datos recibidos causando la pérdida de los mismos.

[48] Esta reducción puede resultar contraproducente una vez alcanzado el nivel de calidad máximo, ya que únicamente se conseguiría reducir los datos almacenados, conduciendo a un peligro de *underflow*.

- Uso del buffer menor al 20%: Teóricamente la situación más crítica de todas. La solución llevada a cabo es idéntica a la anterior pero con un incremento moderado en lugar de agresivo del rango de trasmisión[49].

En caso de que el cliente informe del retardo en el buffer correctamente el algoritmo a llevar a cabo es diferente, en la Figura 19 pueden apreciarse las comparaciones y medidas tomadas en cada uno de los casos. Puede apreciarse como, generalmente, las modificaciones en el rango de transmisión se efectúan a partir de la evaluación del retardo, mientras que las modificaciones de tamaño responden a un análisis en la cantidad de *buffer* utilizado.

Independientemente de si la evaluación cuenta o no con información relativa al retardo, después de las evaluaciones sobre los datos del cliente se procede a comprobar el número de mensajes RTCP que informan de pérdidas. En caso de existir se considerará una pérdida más o menos aleatoria si el RTT es menor a 10; en caso contrario se presupone un ancho de banda insuficiente, por lo que se tiende a reducir la tasa. Por otra parte, si el número de mensajes RTCP sin pérdidas es menor o igual a una variable determinada[50], se considera que la situación aún no es estable y se espera al siguiente ciclo para iniciar el incremento de tasa igualando la variable *fAdjustSize* al mínimo entre *kNoChange* y su valor actual, es decir, si se tomó la decisión de incrementar pasa a no variar y si se decidió decrementar se mantiene.

A continuación se evalúa la variable de cambio de tamaño y se actúa en consecuencia. Por su parte, la variable de variación temporal es comprobada en la función *GetAdjustedTransmitTime* del mismo fichero *RTPStream3gpp.cpp*. Además de estas comprobaciones se realiza un ajuste del nivel máximo de calidad en función del número de informes de RTT grandes ya que se considera que el router está sufriendo congestión.

[49]En este caso se considera una congestión elevada de la red y se opta por no saturarla más a riesgo de perder al cliente.

[50] Por defecto, 2.

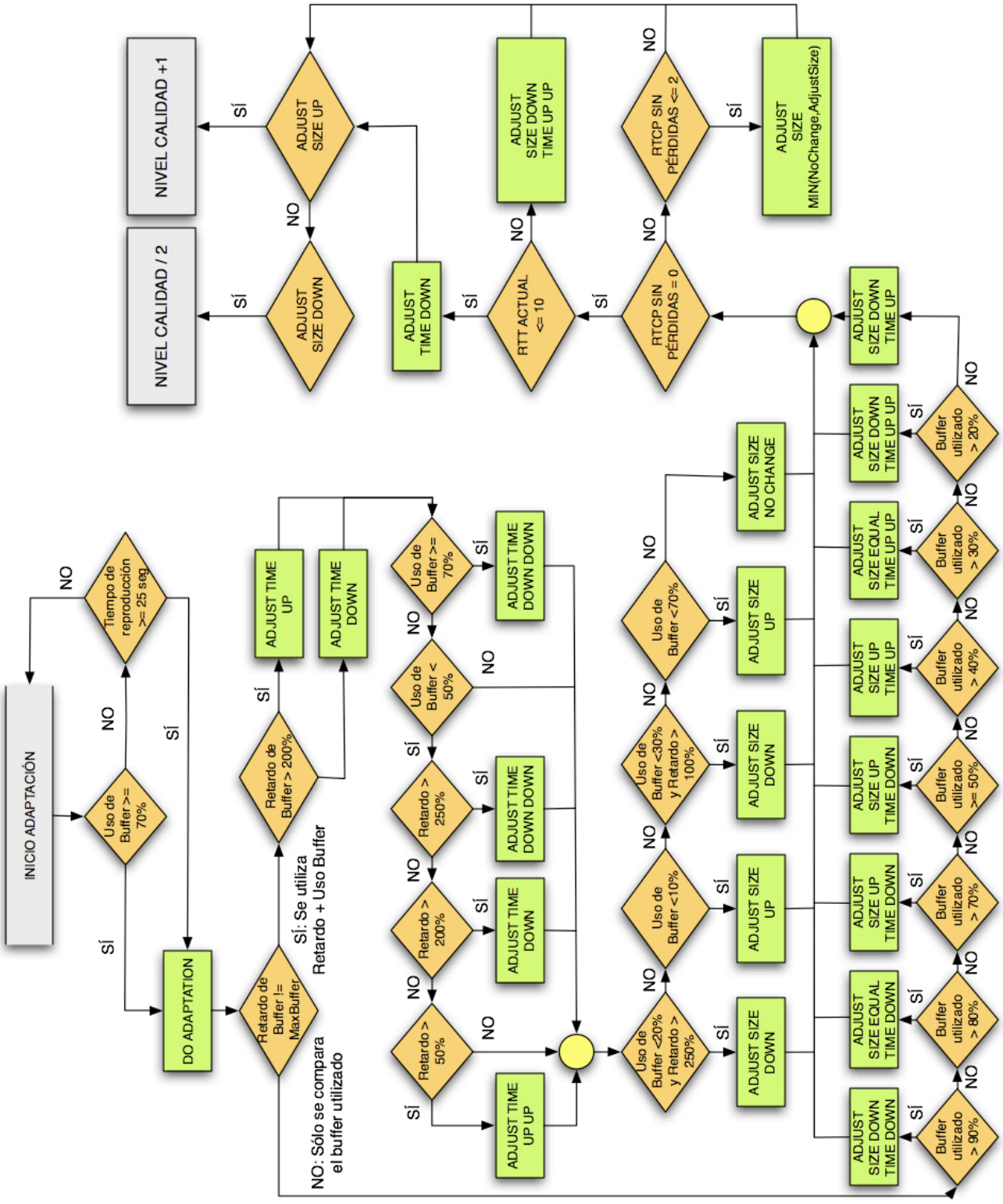

Figura 19: Algoritmo de adaptación de contenidos para archivos 3GP

6.2.4. Modificaciones necesarias en el servidor: Introducción y objetivos

Como se comentó con anterioridad, en el estándar del 3GPP se define la adaptación de tasa como dependiente de la plataforma, es decir, no se especifica ningún protocolo adicional para llevarla a cabo. En este proyecto, al trabajarse con terminales móviles como clientes en los cuales no existe la posibilidad de modificar el sistema de reproducción, se ha optado por una solución transparente de cara a éstos; esta solución implica la imposibilidad de añadir nuevos mensajes entre cliente y servidor, además todas las decisiones deberán ser tomadas por este último.

El objetivo es utilizar un sistema de múltiples pistas tanto de audio como de vídeo equivalentes en codificación pero diferentes en tasa con la finalidad de transmitir al cliente un único par de éstas según las condiciones del canal. De cara al cliente éste recibe la descripción de un archivo con dos pistas, audio y vídeo las cuales seleccionará para que el servidor inicie la transmisión de unos datos cuya fuente variará en función de los informes lanzados por el cliente. Para el terminal siempre se recibirá el mismo flujo ya que se mantendrá toda la información de éste tanto en el servidor como en los paquetes. Será trabajo del servidor decidir en qué momento debe efectuarse el cambio de calidad que en todo momento pasará desapercibido para el cliente. Con este tipo de cambios se pretende lograr una auto adaptación de los elementos de la red ante circunstancias adversas tales como un incremento en el número de clientes del punto de acceso, una pérdida de tasa de transmisión al desplazarse o simplemente colisiones entre paquetes.

En el apartado anterior se ha presentado un algoritmo de adaptación de tasa basado en la eliminación de diferentes tramas de un mismo flujo; en la solución propuesta en este proyecto se pretende realizar un cambio de flujo completo, no obstante, puede utilizarse el concepto de niveles de calidad y las métricas y evaluaciones presentadas en el algoritmo original. En la primera solución se presentará una modificación de dicho algoritmo adaptándolo a las necesidades del proyecto.

La idea al inicio de este proyecto era mantener, en la medida de lo posible, las variables, estructuras y sistemas utilizados por el servidor realizando las modificaciones necesarias en módulos externos para prevenir futuros problemas de compatibilidad. Por otra parte es importante ceñirse a las especificaciones tanto de los protocolos como de los diferentes estándares implicados, comentados en la sección de estado del arte y tecnologías implicadas.

En los siguientes apartados se presentan, a grandes rasgos, las principales modificaciones llevadas a cabo en el servidor para soportar la adaptación de tasa, tanto a nivel conceptual como de programación. Toda la información adicional referente al software puede encontrarse en los diferentes anexos y adjuntos de este documento.

Las principales modificaciones incluyen:

- Creación de un sistema de almacenamiento, gestión y control de grupos de pistas alternativas sin la solicitud del cliente.
- Generación de mensajes SDP siguiendo el estándar por una parte y mostrando al cliente únicamente la información que necesita en cada momento.
- Adición de pistas alternativas a las solicitadas por el cliente de forma transparente para gestionar los diferentes cambios de calidad.

- Gestión de los mensajes RTSP enviados hacia el cliente para respetar la información que éste conoce.
- Modificación del algoritmo existente para adaptarlo al nuevo sistema de cambio de calidad.
- Funciones para intercambio entre flujos de forma transparente para el cliente.

6.2.5. Concepto para la gestión de pistas

Ante la imposibilidad de trabajar con nuevos protocolos, mensajes de señalización, etcétera y los problemas generados en algunos clientes, se opta por la siguiente solución: Gestión interna de las diferentes pistas disponibles en un archivo. Presentación al cliente de aquellas que resulten inicialmente más interesantes y gestión transparente del cambio entre éstas; para ello se realiza una identificación interna, ordenado y gestión de cambios independientemente de que el archivo esté codificado o no con información compatible con la *Release* 6 de 3GPP. La idea puede apreciarse en la Figura 20 y muestra como las diferentes pistas de vídeo y audio se agrupan internamente con la finalidad de emular un archivo de dos flujos.

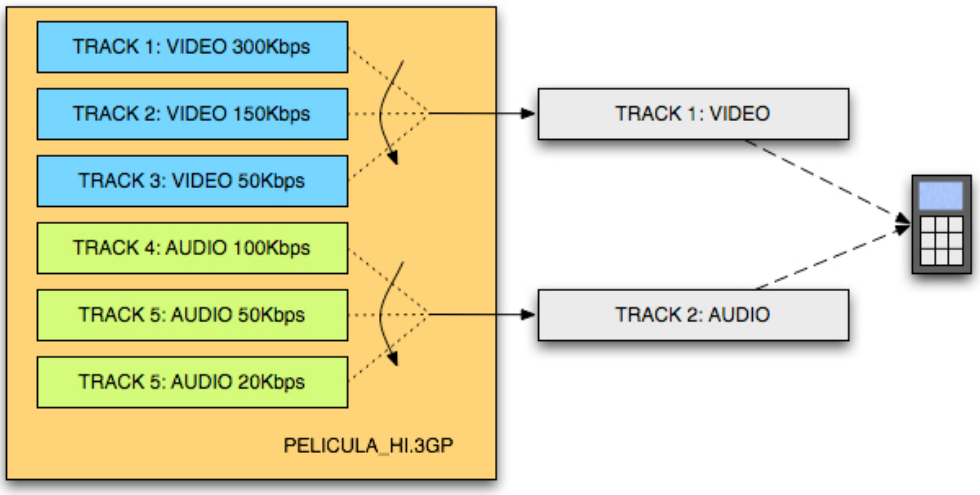

Figura 20: Esquema de gestión de pistas en la transmisión a terminales móviles

A continuación se detallan los pasos a seguir para obtener un sistema transparente para el cliente según evoluciona el flujo de trabajo típico del servidor.

6.2.6. DESCRIBE: Ordenado , control y almacenamiento de pistas

Como se comentó en apartados anteriores, los mensajes de tipo DESCRIBE incluyen la información relativa al archivo al que se quiere acceder; además informan de una serie de atributos relacionados con la adaptación de contenidos tales como pistas alternativas, ancho de banda etcétera. Del mismo modo, se comentó que los terminales actuales no reaccionan a ninguna de dichas descripciones: No reconocen grupos de pistas ni realizan una selección racional de parejas de estas, ya sea en función del ancho de banda disponible o de las características del terminal.

En los mensajes DESCRIBE se incluyen todas las pistas disponibles en un archivo y, aunque los terminales móviles suelen escoger una pareja de pistas[51][52], algunos clientes[53] realizan SETUP de todas las pistas disponibles iniciando una reproducción paralela de éstas. Para evitar este escenario, el servidor escogerá únicamente una pareja de pistas disponible y la anunciará al cliente mediante el mensaje DESCRIBE, al mismo tiempo almacenará el resto en grupos ordenados según ancho de banda para facilitar el intercambio entre éstos, para ello es preciso realizar una serie de modificaciones en las siguientes funciones y estructuras:

- *DoDescribe* en *QTSSFileModule*: Es la función que se ejecuta cuando el cliente solicita un DESCRIBE de un archivo. En esta función, entre otras cosas, se realiza un análisis del archivo para, por una parte, almacenarlo en la sesión de archivo[54] y por otra generar el SDP para enviar al cliente. Inicialmente esta información es la misma y se pueden encontrar funciones como *GetSDPFile* de *QTRTPFile* y variables como *sdpLen* que procesaban el SDP del archivo y almacenaban su longitud respectivamente.

- *GetSDPFile* en *QTRTPFile*: Como se ha comentado con anterioridad, es la función encargada de analizar el archivo para generar un SDP que cumpla con el estándar correspondiente. Para ello, entre otros, va haciendo una lectura recursiva de las diferentes pistas del archivo y añade su información secuencialmente en el *buffer* del SDP.

- *RTPTrackListEntry* en *QTRTPFile.h:* Es la estructura utilizada para identificar unívocamente a cada una de las pistas implicadas en una sesión RTP. La información que contiene puede apreciarse en la Tabla 7[55]. Para llevar a cabo la adaptación se han agregado cuatro campos que permiten identificar el ancho de banda máximo consumido por la pista, el tipo de carga que transporta[56] y el identificador de las pistas inmediatamente anteriores y posteriores en cuanto a nivel de calidad, es decir, indica la pista del mismo tipo cuyo nivel de calidad es inmediatamente superior o inferior, si este valor es cero se considera que la pista es la de mayor y/o menor ancho de banda disponible.

Una vez realizadas las funciones se obtiene un SDP que es almacenado en la sesión y enviado al cliente al mismo tiempo. Es importante notar que el SDP almacenado en la sesión será referencia constante durante la misma, es decir, únicamente podrán accederse a las pistas y contenidos de éste durante la reproducción del archivo. Toda la información que quede almacenada en este momento

[51] Una de audio y otra de vídeo.

[52] Generalmente la primera disponible.

[53] VLC, QuickTime, etcétera.

[54] FileSession según el manual de QTSS.

[55] En verde los campos añadidos para la adaptación.

[56] Discrimina entre audio y vídeo.

será la única conocida por el servidor, por ello es importante respetar al máximo toda la información original almacenada en el archivo.

Tipo	Nombre	Tipo	Nombre
UInt32	TrackID	UInt32	CurSampleNumber
QTHintTrack	*HintTrack	UInt32	ConsecutivePFramesSent
QTHintTrack_HintTrackControlBlock	*HTCB	UInt32	TargetPercentage
Bool16	IsTrackActive	UInt32	SampleToSeekTo
Bool16	IsPacketAvailable	UInt32	LastSyncSampleNumber
UInt32	QualityLevel	UInt32	NextSyncSampleNumber
void	*Cookie1	UInt16	NumPacketsInThisSample
UInt32	Cookie2	UInt16	CurPacketNumber
UInt32	SSRC	Float64	CurPacketTime
UInt16	FileSequenceNumberRandomOffset	Char	CurPacket[max_length]
UInt16	BaseSequenceNumberRandomOffset	UInt32	CurPacketLength
UInt16	LastSequenceNumber	UInt32	Group
SInt32	SequenceNumberAdditive	UInt16	Payload
UInt32	FileTimestampRandomOffset	UInt32	NextQTrackID
UInt32	BaseTimestampRandomOffset	UInt32	PrevQTrackID
RTPTrackListEntry	*NextTrack		

Tabla 7: Elementos de la estructura de una pista RTP

En este momento se presenta la problemática de, por una parte, respetar la información obtenida de la pista para almacenarla en la sesión y, por otra, presentar al cliente únicamente una pareja de pistas utilizando un cierto criterio. Además sería interesante ir realizando un pequeño análisis de las pistas disponibles para su posterior uso en los algoritmos de adaptación. Para todo ello se decide diseñar una nueva función para obtener el SDP:

- *MyGetSDPFile* en *QTRTPFile*: Al igual que *GetSDPFile*, esta función tiene como finalidad realizar un análisis de las pistas disponibles para generar un SDP. La principal diferencia con la anterior es que realiza un análisis de las pistas del archivo para generar SDP ordenado. Además cuenta con dos parámetros de longitud en lugar de uno para determinar, el tamaño total del SDP y el tamaño hasta la segunda pista incluida. Estos dos tamaños permiten entregar al cliente únicamente una parte del SDP, mientras que internamente se almacena la información completa. Para ordenar las pistas, durante el análisis de éstas, se almacena el ancho de banda y su *payload* gracias a un parseo de su *hint*. Puede verse la extracción de datos en el cuadro siguiente:

```
temptrackSDP = curEntry->HintTrack->GetSDPFile(&temptrackSDPLength);
curEntry->Group=atoi(strstr(temptrackSDP,"b=AS:")+5);
curEntry->Payload = strstr(temptrackSDP,"video")?1:0;
```

Una vez clasificadas todas las pistas con su carga y ancho de banda se procede a la ordenación de las mismas a través de las funciones *minBWTrack* y *minBWTrack* de *QTRTPFile*. Estas funciones asocian a cada pista su anterior y consecutiva en términos de ancho de banda siempre que compartan *payload*. A continuación se muestra un ejemplo de una de ellas y como, para todas las pistas, busca la inmediatamente anterior del mismo tipo.

```
UInt32 QTRTPFile::minBWTrack(Uint32 trackPayload, Uint32 maxBW)
{
    RTPTrackListEntry*  curEntry;
    UInt32 trackID=0;
    UInt32 tempBW = 0;

    for (curEntry = fFirstTrack; curEntry != NULL; curEntry = curEntry->NextTrack)
    {
          if (curEntry->Payload==trackPayload && curEntry->Group<maxBW &&
         curEntry->Group>tempBW)
                    {
                    trackID=curEntry->TrackID;
                    tempBW=curEntry->Group;
                    }
     }
     return trackID;
}
```

Una vez identificadas todas las pistas con su tipo, ancho de banda y pistas adyacentes se procede a la construcción del SDP que se enviará al cliente. Para ello se añaden a un buffer aquellas pistas de mayor ancho de banda[57] y se cuantifica su longitud, posteriormente se hará lo propio con el resto. En la respuesta para el cliente deberá contabilizarse únicamente hasta la longitud de la pareja de pistas mientras que para el almacenado se contabilizará la longitud total, de este modo el cliente percibe únicamente la pareja que el servidor le proporciona impidiéndole hacer SETUP de otras pistas diferentes.

En el siguiente cuadro se muestra como queda un SDP generado utilizando las funciones presentadas. Las líneas coloreadas indican los límites establecidos por cada una de las longitudes.

[57] Una por tipo.

```
a=x-copyright: MP4/3GP File hinted with GPAC 0.4.4 (C)2000-2005 - http://gpac.sourceforge.net
a=range:npt=0-1369.79000
m=video 0 RTP/AVP 96
b=AS:341
a=rtpmap:96 MP4V-ES/90000
a=control:trackID=65536
a=fmtp:96 profile-level-id=1;
config=000001b001000001b589130000010000000120000c48d89d4c505841214630000001b24c
61766331642e35312e33382e30
a=framesize:96 176-144
m=audio 0 RTP/AVP 97
b=AS:114
a=rtpmap:97 mpeg4-generic/22050/2
a=control:trackID=65537
a=fmtp:97 profile-level-id=40; config=1390; streamType=5; mode=AAC-hbr; objectType=64;
sizeLength=13; indexLength=3; indexDeltaLength=3
m=video 0 RTP/AVP 98
                    ----------longitud del SDP para el cliente----------
b=AS:253
a=rtpmap:98 MP4V-ES/90000
a=control:trackID=65538
a=fmtp:98 profile-level-id=1;
config=000001b001000001b589130000010000000120000c48d89d4c505841214630000001b24c
61766331642e35312e33382e30
a=framesize:98 176-144
m=audio 0 RTP/AVP 99
b=AS:114
a=rtpmap:99 mpeg4-generic/22050/2
a=control:trackID=65539
a=fmtp:99 profile-level-id=40; config=1390; streamType=5; mode=AAC-hbr; objectType=64;
sizeLength=13; indexLength=3; indexDeltaLength=3
m=video 0 RTP/AVP 100
b=AS:96
a=rtpmap:100 MP4V-ES/90000
a=control:trackID=65540
a=fmtp:100 profile-level-id=1;
config=000001b001000001b589130000010000000120000c48d89d4c505841214630000001b24c
61766331642e35312e33382e30
a=framesize:100 176-144
m=audio 0 RTP/AVP 101
b=AS:114
a=rtpmap:101 mpeg4-generic/22050/2
a=control:trackID=65541
a=fmtp:101 profile-level-id=40; config=1390; streamType=5; mode=AAC-hbr; objectType=64;
sizeLength=13; indexLength=3; indexDeltaLength=3
                    ----------longitud del SDP para el servidor----------
```

Un dato importante a tener en cuenta es el hecho de que cada pista es identificada unívocamente mediante un tipo de *payload* concreto (ver Tabla 4 y puntos adyacentes), es decir, ante

el cambio de pistas por parte del servidor el cliente deberá percibir un mismo campo PT en los diferentes paquetes para que dicho cambio sea transparente. Dado que en todas las pruebas realizadas se utiliza un rango dinámico, los valores suelen estar comprendidos entre 96 y 96+n siendo n el número de pistas. Típicamente la estructura utilizada es asignar un identificador secuencial a las pistas según estas son añadidas, normalmente de forma alterna audio-vídeo.

Para evitar la necesidad de llevar un control de tipos de *payload* utilizados, activos y existentes, se ha tomado la decisión de marcar todos los paquetes pertenecientes a las pistas de vídeo con PT = 96 y a las de audio con PT = 97, este sistema es equivalente a almacenar el PT real de las pistas anunciadas inicialmente y variarlo en el resto, pero restándole complejidad.

Para que el anuncio inicial contenga los valores deseados se realiza un análisis de las pistas de mayor ancho de banda y se modifican todos los campos relacionados con PT[58] que aparecen en el SDP por los valores deseados. Puede apreciarse esta modificación en el siguiente cuadro donde los valores deseados varían en función del *payload* de la pista y donde en caso de existir alguno de los campos mencionados anteriormente se realiza la modificación:

```
const char * newInfo =(curEntry->Payload == 1)?"96":"97";

char * myPtr = NULL;
if(myPtr = strstr(pSDPFile,"RTP/AVP ")) ::memcpy(myPtr+8, newInfo, 2);
if(myPtr = strstr(pSDPFile,"rtpmap:"))   ::memcpy(myPtr+7, newInfo, 2);
if(myPtr = strstr(pSDPFile,"fmtp:"))     ::memcpy(myPtr+5, newInfo, 2);
```

De este modo el mensaje SDP enviado al cliente tiene el siguiente aspecto:

[58] RTP/AVP, rtpmap y fmtp.

```
v=0
o=StreamingServer 3443763354 1233654121000 IN IP4 192.168.1.100
s=/st.mp4
u=http:///
e=admin@
c=IN IP4 0.0.0.0
b=AS:1032
t=0 0
a=control:*
a=x-copyright: MP4/3GP File hinted with GPAC 0.4.4 (C)2000-2005 - http://gpac.sourceforge.net
a=range:npt=0-1369.79000
m=video 0 RTP/AVP 96
b=AS:341
a=3GPP-Adaptation-Support:1
a=rtpmap:96 MP4V-ES/90000
a=control:trackID=65536
a=fmtp:96 profile-level-id=1;
config=000001b001000001b589130000010000000012000c48d89d4c50584121463000001b24c6
1766331642e35312e33382e30
a=framesize:96 176-144
m=audio 0 RTP/AVP 97
b=AS:114
a=3GPP-Adaptation-Support:1
a=rtpmap:97 mpeg4-generic/22050/2
a=control:trackID=65537
a=fmtp:97 profile-level-id=40; config=1390; streamType=5; mode=AAC-hbr; objectType=64;
sizeLength=13; indexLength=3; indexDeltaLength=3
```

A partir de este momento el cliente recibe mediante el SDP las pistas que resultan interesantes desde el punto de vista del servidor. En este proyecto se presenta la posibilidad de que la pareja de pistas entregadas sean aquellas con mayor ancho de banda; cabe la posibilidad de entregar otras pistas tanto de manera fija como siguiendo algún criterio adicional, no obstante, el algoritmo de adaptación que será presentado posteriormente posibilita el hecho de que se realice un cambio de flujo relativamente rápido adaptándose a las circunstancias de la red. Desde este punto el cliente lanzará una serie de mensajes SETUP hacia el servidor.

6.2.7. SETUP: Sistema de adición de pistas alternativas

Una vez se ha enviado la información pertinente al cliente mediante el SDP, éste lanza tantos SETUP como pistas anuncie el servidor, generalmente sin tener en cuenta ningún tipo de criterio adicional. Tal y como se ha presentado en el punto 6.2.6, el cliente recibirá una única pareja de flujos audio vídeo que procederá a solicitar. De cara al cliente estas son las únicas pistas disponibles y, durante todo el proceso, será las que considerará que ejecuta. Por ello deben respetarse todas las variables relativas a dichas pistas, desde el tipo de carga hasta números de secuencia, pasando por marcas temporales, codificación, etcétera. Durante la función *DoSetup* de *QTSSFileModule* se ejecuta una función que añade las pistas solicitadas a la sesión; para realizar esta adición es necesario que previamente se hallen la pista deseada en el SDP almacenado según la función de DESCRIBE; las

modificaciones llevadas a cabo en la función comentada tienen como finalidad realizar la adición de todas las pistas relacionadas con la solicitada sin que el cliente tenga que lanzar las peticiones correspondientes. Posteriormente debe enviarse una respuesta satisfactoria para el cliente.

Para llevar a cabo estas tareas se identifica por una parte la pista solicitada[59] y su tipo de *payload*:

```
UInt32 originalTrackID = ::strtol(theDigitStr, NULL, 10);

for (UInt32 x = 0; x < theFile->fSDPSource.GetNumStreams(); x++)
  {
  SourceInfo::StreamInfo* theStreamInfo = theFile->fSDPSource.GetStreamInfo(x);
  if (theStreamInfo->fTrackID == originalTrackID)
        {
            trackType = theStreamInfo->fPayloadType;
            break;
        }
    }
```

Posteriormente se realizan las funciones propias del servidor para la pista solicitada pero para todas aquellas que compartan *payload* con la original. Esto permite, por ejemplo, añadir todas las pistas de vídeo cuando se solicita la pista de vídeo original. Esta adición se realiza utilizando los mismos parámetros que la pista original.

```
for (UInt32 x = 0; x < theFile->fSDPSource.GetNumStreams(); x++)
    {
        SourceInfo::StreamInfo* theStreamInfo = theFile->fSDPSource.GetStreamInfo(x);
        if (theStreamInfo->fPayloadType!=trackType) continue;
     …
    }
```

Dentro de las funciones típicas se utiliza una nueva función llamada *DeActivateTrackID* en *QTRTPFile*, la cual marca la pista como inactiva con lo que la función correspondiente no enviará paquetes pertenecientes a dicha pista. Las pistas permanecerán inactivas hasta que alguna función indique lo contrario. Detalle de la llamada a la función *DeActivateTrackID*.

```
QTRTPFile::ErrorCode qtfileErr = theFile->fFile.AddTrack(theTrackID, false);
if (theTrackID != originalTrackID)
qtfileErr = theFile->fFile.DeActivateTrackID(theTrackID);
```

Y ahora un la definición de la propia función:

[59] Conocida como originalTrackID.

```
QTRTPFile::ErrorCode QTRTPFile::DeActivateTrackID(UInt32 TrackID)
{
 RTPTrackListEntry *trackEntry = NULL;
 if( !this->FindTrackEntry(TrackID, &trackEntry))  return fErr =  errTrackIDNotFound;
 trackEntry->IsTrackActive = false;
 return errNoError;
}
```

Es importante destacar que durante el proceso de adición de pistas a todas ellas se les atribuyen las mismas variables iniciales tales como SSRC, puertos de comunicación, valores iniciales en contadores, etcétera. Una vez añadidas todas las pistas y desactivadas las alternativas sólo se debe responder el mensaje RTSP al cliente. Para lograrlo se aplica la condición siguiente:

```
if (theTrackID==originalTrackID)
{
(void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssLastModifiedHeader,
theFile->fFile.GetQTFile()->GetModDateStr(), DateBuffer::kDateBufferLen);

(void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssCacheControlHeader,
kCacheControlHeader.Ptr, kCacheControlHeader.Len);

theErr = QTSS_SendStandardRTSPResponse(inParamBlock->inRTSPRequest, newStream, 0);

Assert(theErr == QTSS_NoErr);
}
```

A la hora de añadir las pistas hay un parámetro importante a tener en cuenta: El número de niveles de calidad. Este nivel de calidad máximo se utiliza en el algoritmo de adaptación clásico para definir los niveles y en el nuevo algoritmo para indicar el máximo índice de calidad como se presentará más adelante. En la nueva implementación se ha optado por externalizar esta variable como parámetro para permitir su fácil modificación. El sistema a utilizar es el mismo que con el resto de parámetros explicados con anterioridad.

Este parámetro debe ser añadido a cada una de las pistas existentes. Para ello se utiliza la función *SetTrackQualityLevel* que se encuentra en el archivo *QTRTPFile*. Originalmente esta función cuenta con dos parámetros: la estructura de pista RTP a la que se desea asignar el nivel de calidad y el nuevo nivel. Inicialmente se asigna un nivel entre el máximo de 6 existentes en los que se define el porcentaje de tramas a descartar.

En esta nueva versión el nivel de calidad tiene un rango mucho más amplio como se mostrará posteriormente. La función agrega un tercer parámetro indicando el máximo nivel de calidad disponible ya que el nuevo nivel de calidad a asignar dependerá de este máximo como se explica en la sección del nuevo algoritmo.

6.2.8. Generación de respuestas RTSP

Las peticiones RTSP están ligadas a la sesión del usuario; esta sesión incluye todas las pistas RTP agregadas con anterioridad. Desde el punto de vista del usuario, dicha sesión únicamente contiene las dos pistas seleccionadas inicialmente, mientras que de cara al servidor se incluyen además las alternativas a éstas. Anteriormente se presentó cómo las respuestas RTSP a solicitudes PLAY incluyen la información relativa a los flujos implicados en la reproducción. Por defecto el servidor tiene definidas unas funciones que incluyen en dichas respuestas la información de todas las pistas existentes; ya que esto supone un problema para el cliente, es necesario hacer una modificación para discriminar las pistas sobre las que se realiza la respuesta. Por otra parte el hecho de incluir únicamente las pistas que se agregaron inicialmente no es una solución válida ya que el cliente podría lanzar alguna petición RTSP mientras se encuentra activa otra pareja.

Las respuestas RTSP por parte del servidor se realizan llamando a la función *QTSS_SendStandardRTSPResponse* la cual se encarga de discriminar si se trata de un PLAY, PAUSE, TEARDOWN, etcétera. Para el correcto envío de respuestas RTSP se ha realizado una modificación en ducha función, la cual pasa de tener tres atributos fijos a un número indeterminado[60]. La idea es permitir que los dos últimos parámetros se utilicen como identificadores de pistas activas, útiles para enviar respuestas a mensajes de PLAY. Se ha decidido pasar de tres a indeterminado un lugar de cinco para permitir la compatibilidad con todas las llamadas que sufre la función por parte de los diferentes módulos.

Esta función modificada viene definida en el archivo *QTSSCallbacks* por los siguientes parámetros:

```
QTSS_Error QTSSCallbacks::QTSS_SendStandardRTSPResponse(QTSS_RTSPRequestObject
inRTSPRequest, QTSS_Object inRTPInfo, UInt32 inFlags,…)
```

A partir del parámetro *Request* puede extraerse el tipo de petición generada por el cliente según el cual se enviarán diferentes respuestas. Este es el único caso para el que se pasarán parámetros adicionales. Es posible gestionar los parámetros variables gracias a la librería *stdarg.h* que permite definir un inicio y un puntero para acceder a los mismos. Se puede apreciar un ejemplo práctico en la respuesta al caso PLAY de *SendStandardRTSPResponse* incluido a continuación:

```
switch (((RTSPRequestInterface*)inRTSPRequest)->GetMethod())
   {
      case qtssPlayMethod:
      va_list  param_pt;
      va_start(param_pt, inFlags);
      ((RTPSession*)inRTPInfo)->SendPlayResponse((RTSPRequestInterface*)inRTSPRequest, inFlags,
      va_arg(param_pt,UInt32), va_arg(param_pt,UInt32));
      return QTSS_NoErr;
      ….
     }
```

[60] Generalmente se utilizarán tres o cinco.

Para gestionar las variables múltiples la librería permite la utilización de diferentes funciones. Inicialmente debe definirse un índice, en este caso llamado *param_pt* que actúa como puntero. A continuación la función *va_start* define el nombre del parámetro a partir del cual accederá el puntero; en este ejemplo se apunta a la última variable fija llamada *inFlags*. Desde ese momento se va accediendo a los siguientes parámetros de forma secuencial utilizando la función *va_arg* que tiene como primer parámetro el puntero utilizado y como segundo el tipo de datos que se quieren extraer. En este ejemplo se extraen los ID de pista que se pasan cuando se llama a la función tras un PLAY del cliente.

Para llevar a cabo esta modificación de la función, además del fichero *QTSSCallbacks.cpp* y *.h* es preciso modificar *QTSS_Private.cpp* donde se lanza una petición a la función de callbacks. Para permitir el uso de un número de variables indefinidas se repite el proceso presentado con anterioridad.

```
QTSS_Error QTSS_SendStandardRTSPResponse(QTSS_RTSPRequestObject inRTSPRequest, QTSS_Object
inRTPInfo, UInt32 inFlags,...)
{
    va_list param_pt;
    va_start(param_pt, inFlags);
    return (sCallbacks->addr [kSendStandardRTSPCallback]) (inRTSPRequest, inRTPInfo,
inFlags,va_arg(param_pt, UInt32),va_arg(param_pt, UInt32));
}
```

Como se comentó con anterioridad, la modificación únicamente afecta al caso de PLAY en el que se ejecuta la función modificada de SendPlayResponse, la cual realiza una comprobación sobre los números de pista facilitados para escribir la respuesta del cliente:

```
for (UInt32 x = 0; x < valueCount; x++)
    {
        this->GetValuePtr(qtssCliSesStreamObjects, x, (void**)&theStream, &theLen);
        Assert(theStream != NULL);
        Assert(theLen == sizeof(RTPStream*));
        if (*theStream != NULL)
        {
                QTSS_GetValuePtr(*theStream, qtssRTPStrTrackID, 0, (void**)&theTrackID, &theLen);
                if (*theTrackID==activeTrack1 || *theTrackID==activeTrack2)
                {
                        if (y++ == 2)
                        lastValue = true;
                        (*theStream)->AppendRTPInfo(theHeader, request, inFlags, lastValue);
                        theHeader = qtssSameAsLastHeader;
                }
        }
    }
```

Con estas modificaciones, el cliente recibe respuestas correctas para todas las solicitudes RTSP que realice. En todos los casos se utilizan únicamente los tres primeros parámetros ya que las respuestas particulares a solicitudes diferentes al PLAY responden con información relativa a la sesión

y no a dichas pistas; la única salvedad es el caso comentado en el que, de este modo, se indica el número de pista, número de secuencia, marca temporal, etcétera utilizados por cada uno de los elementos activos. El aspecto es similar al siguiente:

```
RTSP/1.0 200 OK
Server: DSS/6.0.3 (Build/526.3; Platform/Linux; Release/Darwin Streaming Server;
State/Development; )
Cseq: 17
Session: 353322952731 1269409
Range: npt=8.40840-1370.16833
RTP-Info:
url=rtsp://192.168.0.100/prueba.3gp/trackID=65536;seq=64243;rtptime=1152163183,
url=rtsp://192.168.0.100/prueba.3gp/trackID=65537;seq=61665;rtptime=732693889
```

En este caso el cliente percibe la recepción de las pistas 65536 y 65537 con su información pertinente aunque realmente se le esté enviando otra pista. Para conocer cuáles son las pistas que se encuentran activas en cada momento, desde *QTSSFileModule* se invoca una nueva función llamada *getActiveTracks*, definida en *QTRTPFile* que retorna los dos valores deseados. Éstos serán utilizados como parámetros al llamar a *SendStandardRTSPResponse*.

```
UInt32 activeTracks[2] = {0,0};

(*theFile)->fFile.getActiveTracks(activeTracks);
(void)QTSS_SendStandardRTSPResponse(inParamBlock->inRTSPRequest, inParamBlock-
>inClientSession, qtssPlayRespWriteTrackInfo,activeTracks[0], activeTracks[1]);
```

6.2.9. Nuevo algoritmo de adaptación

Después de enviar las respuestas pertinentes al cliente tras las solicitudes SETUP y PLAY se inicia la reproducción normal del archivo. A partir de este momento se realiza un análisis de los informes RTCP (RR) y de los mensajes NADU enviados por el cliente tal y como sucedía en el algoritmo original. El nuevo sistema utiliza las mismas herramientas de análisis que el anterior realizando comparaciones similares para evaluar el estado de la red. También comparte el sistema de gestión del tiempo de transmisión en los paquetes para ayudar a ajustar el *buffer* en el cliente. La principal diferencia radica en el sistema de calidades, que pasa de estar basado en un número fijo de calidades que se modifican inmediatamente según los informes del cliente, a tener tantos niveles como pistas alternativas posea el archivo y gestionarse con una variable que denominaremos Índice de Calidad o IC.

El índice de calidad es una variable numérica con un rango y valor inicial determinado; además cuenta con dos límites, uno superior y otro inferior, que indican el nivel a partir del cual se realiza un cambio de calidad. La variación de este índice es típicamente con un incremento secuencial y un decremento multiplicativo debido a que se considera que la reducción de calidad debe ser prioritaria en caso de pérdidas en la red y el incremento más moderado para dar tiempo a los clientes a estabilizarse. También se pretende evitar el continuo intercambio entre niveles de calidad ya que

supone incrementar la probabilidad de pérdidas en la red. Puede apreciarse un ejemplo de variación del IC con sus márgenes máximo y mínimo en la Figura 21.

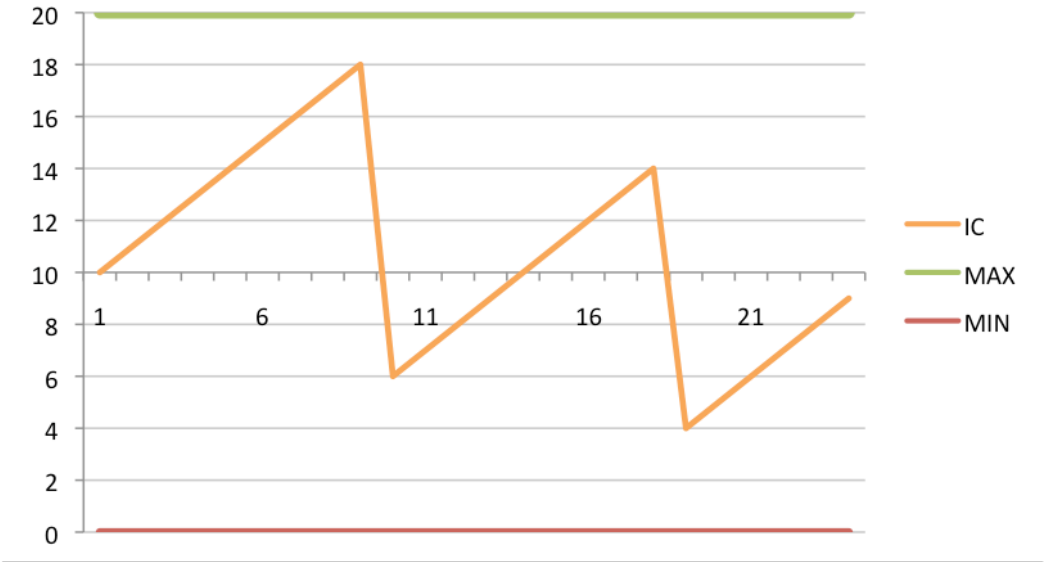

Figura 21: Variación del índice de calidad en el tiempo

En las pruebas realizadas se ha utilizado típicamente un rango para el IC entre 0 y 50 con un valor inicial de 20. El incremento del IC tiene lugar, como en el algoritmo anterior, siempre que se cumplan dos requisitos básicos: Por una parte haber recibido dos informes sin pérdidas de paquetes y por otra encontrarse en una situación favorable en términos de utilización de buffer y retardo de reproducción. Las funciones de decremento son, sin embargo, más agresivas.

Tal y como ocurre en el algoritmo anterior, este indicador de calidad es inversamente proporcional a la calidad real del enlace, de este modo cuanto mayor sea el IC peor son las circunstancias de la red y viceversa. Para trabajar con IC se definen, además del rango en el que se moverá éste, un límite superior e inferior para realizar los cambios de calidad. Típicamente se escoge para nivel inferior[61] el mínimo valor del rango (0) y para nivel superior y decremento de calidad un valor igual a 3/4 del máximo, es decir, en un rango de 0 a 50 los valores mínimo y máximo serían 0 y 37 respectivamente.

Todos estos valores se pueden extraer a partir del nivel máximo de IC. Para llevar a cabo un análisis del rendimiento con diferentes parámetros y facilitar su modificación, se ha incluído dicho máximo como parámetro externo en el archivo de configuración (ver sección 5.5.5). A partir de éste se define un mínimo fijo 0, un nivel inicial $0.4 \cdot IC_{MAX}$ típicamente igual a 20 y un límite máximo de $¾ \cdot IC_{MAX}$. En apartados posteriores se muestran algunas pruebas comparando el rendimiendo con diferentes parámetros de IC.

Al igual que sucede en el algoritmo original el decremento de la calidad es siempre el resultado dividir entre dos el nivel anterior. En este caso la función retorna la diferencia entre el valor máximo y

[61] Y por lo tanto incremento de calidad.

la mitad de la distancia entre el máximo y el actual. Por ejemplo, en un rango 0-50 con un IC actual IC_A la función devolvería un nuevo valor IC_N donde:

$$IC_N = 50 - (50\text{-}IC_A)/2$$

Este rango está pensado para propiciar el decremento de calidad cuando dos informes negativos son recibidos de forma consecutiva. Con un valor inicial típico de IC_A=10, tras un informe negativo tenemos que IC_N=30 y tras el segundo IC_{N2}=40, mayor que el límite 37 y por lo tanto se produciría el decremento de calidad.

Una vez realizado un cambio de flujo este IC vuelve a su valor inicial y se continúa con la evaluación de la información de cliente, gracias a este sistema y a las funciones de variación del IC se impide que el cliente sufra constantes variaciones en la calidad del flujo, al menos en diferentes sentidos ya que es típico que una vez que se producen pérdidas en el canal, la información recibida por el cliente sea suficientemente negativa como para implicar el descenso de más de un nivel de calidad. Puede apreciarse un ejemplo real de evolución del parámetro IC y los niveles de calidad en la Figura 22 donde se establece un IC inicial de 20, un rango 0-50 y un límite para decremento de 37. Inicialmente la transmisión es correcta hasta que en el instante t=10 un informe negativo implica una variación agresiva del IC. En este ejemplo se representa un escenario típico donde los informes negativos se suceden debido a una serie de pérdidas en la red, por ello durante un instante (t=10 – 16) no sólo no mejora el IC debido a la protección a la que se le somete (2 Informes sin pérdidas antes de incrementar su valor) si no que este se vuelve a degradar agresivamente dando lugar a un segundo cambio de nivel de calidad.

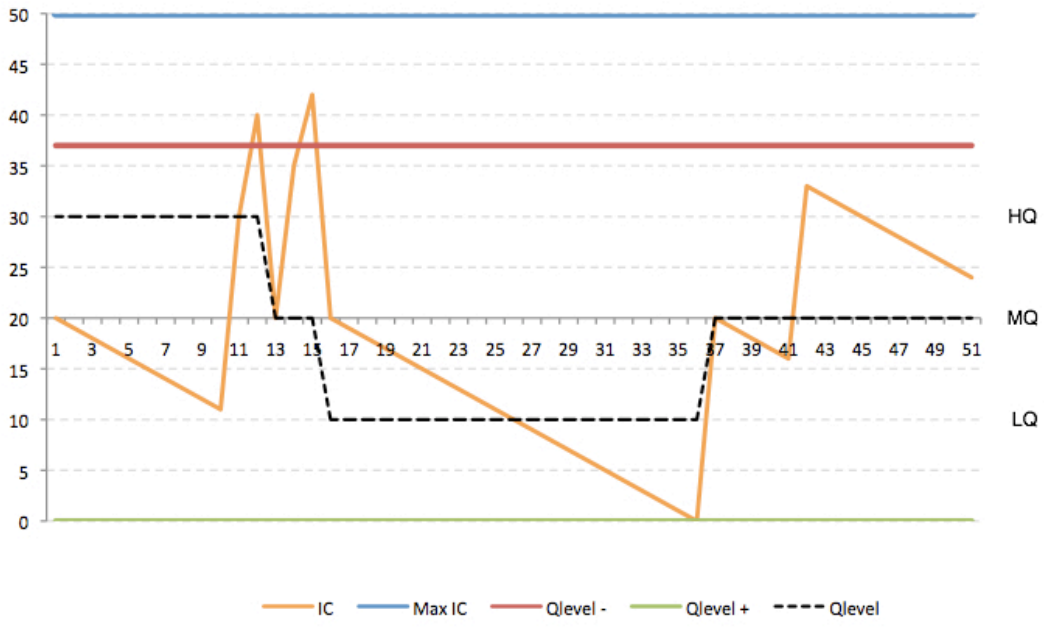

Figura 22: Variación del índice de calidad en el tiempo

A partir de ese momento se suceden los informes positivos que concluyen en un aumento de la calidad (t=37) y posteriormente se combinan los informes positivos con negativos dando lugar a variaciones del IC que no llega a volver a variar el nivel general. Es típico encontrarse con situaciones en las que el cliente llega a un nivel adecuado (digamos el segundo de tres) y va combinando

informes negativos con positivos; gracias al sistema del IC se mantiene en este nivel durante el periodo de reproducción en lugar de fluctuar. Por otra parte suelen darse casos en los que la red es apta para clientes en el segundo nivel pero no soportan el nivel máximo; en ese caso suele ocurrir que éstos, una vez en el nivel intermedio, van realizando paulatinamente intentos de incrementar la calidad la cual vuelve al estado anterior rápidamente debido a nuevas pérdidas en la red.

Cuando el nivel IC correspondiente es alcanzado, el servidor realiza de manera interna y totalmente transparente para el cliente el cambio de pistas, momento a partir del cual éste recibe un flujo de una calidad diferente a la anterior en el mismo instante en el que se encontraba. Este cambio puede realizarse a nivel de vídeo, audio o ambos y para ello existe una función fundamental llamada *SwapTracks*. Cabe destacar que existe la posibilidad de trabajar con dos niveles de calidad, uno general de la sesión y otro referente a las pistas, este algoritmo utiliza la segunda opción ya que se considera que puede decidirse mantener un nivel de calidad para un flujo y variar otro, típicamente decrementar la calidad del vídeo y mantener el audio intacto.

La comprobación de niveles de calidad se realiza antes de la transmisión de cada paquete, dentro de la función *SetTrackQualityLevel* utilizada originalmente para discriminar el número de tramas que debían desecharse de cada flujo. En este caso el nivel de calidad *inNewQualityLevel* se refiere al IC y se adhiere un tercer parámetro que indica el número máximo de éste. Este nuevo parámetro es necesario para asignar el valor inicial del IC y considerar si supera los niveles para el cambio de pista.

```
void QTRTPFile::SetTrackQualityLevel(RTPTrackListEntry* inEntry, UInt32 inNewQualityLevel, UInt32
inMaxQualityLevel)
{

  if (inNewQualityLevel != inEntry->QualityLevel)
    {
    UInt32 threshold = 3*(inMaxQualityLevel/4);
    if (inNewQualityLevel>threshold && inEntry->QualityLevel<inNewQualityLevel && inEntry-
>QualityLevel!=0)
        {
        inEntry->QualityLevel = inMaxQualityLevel*0.4;
        this->SwapTracks(false);
        return;
        }
    else if (inNewQualityLevel == 0 && inEntry->QualityLevel>inNewQualityLevel && inEntry-
>QualityLevel!=10)
        {
        inEntry->QualityLevel = 0;
        this->SwapTracks(true);
        return;
        }

     inEntry->QualityLevel = inNewQualityLevel;
     }
}
```

6.2.10. Evaluación de los parámetros del algoritmo.

Una de las principales características del nuevo algoritmo de adaptación es la diferencia de respuesta respecto al original. En este caso se considera que una respuesta inmediata resulta perjudicial para la red debido a que no permite salir del estado de saturación y desemboca en una constante fluctuación de calidades. Por otro lado, una respuesta demasiado lenta supone el desaprovechamiento de ancho de banda disponible y una peor calidad de experiencia para el usuario.

El decremento de calidad está ligado a los informes negativos por parte del cliente, mientras que el incremento responde a los positivos. Como se ha visto anteriormente, tal y como se define el nuevo algoritmo se necesitan generalmente dos informes negativos para decrementar la calidad. Los informes positivos deben ser, por el contrario, mucho mayores en número además de consecutivos para lograr el incremento. Existe por lo tanto entre un incremento rápido de calidad y evitar desestabilizar el sistema. Este compromiso se reduce en una elección del valor máximo del IC. Un valor de IC pequeño tiene como resultado una mayor fluctuación de calidades mientras un IC grande mantiene a los clientes más tiempo sin incrementar la tasa del flujo.

En los experimentos llevados a cabo se ha observado que el factor determinante para la variación del IC no es el estado del buffer ni el retardo sufrido por el cliente. Se da con mucha más frecuencia la recepción de informes que alertan de pérdidas de paquetes o RTT altos. Esto se debe a que las variaciones en los bufferes de clientes responden principalmente a estas pérdidas y por lo tanto se notifican posteriormente. Además, en una red de estas características resulta frecuente la pérdida de paquetes consecutivos cuando ésta llega a un nivel de saturación.

Dadas estas características se ha decidido realizar una serie de modificaciones en la lógica de decisión del algoritmo original. Mientras aquellas condiciones relacionadas con los ajustes temporales se han mantenido intactas, se ha realizado una modificación importante relacionada con las pérdidas de las que informa el cliente. Básicamente se trata de incrementar el periodo de tiempo que el cliente necesita para recuperarse tras un informe negativo. En particular se aumenta de 2 a 4 el número de informes RTCP sin pérdidas necesarios para iniciar el incremento de calidad. Por otra parte se considera que a partir de 10 informes consecutivos sin pérdidas la red es suficientemente estable y se inicia un proceso contínuo de variación del IC hasta alcanzar el incremento. Esta variación permite una recuperación más rápida por parte del cliente. Gracias a estas incorporaciones el valor del IC resulta menos determinante aunque en las pruebas realizadas se observa que el mejor compromiso se encuentra con los ejemplos presentados con anterioridad (rango 0-50). Además se agrega una funcionalidad adicional; a todos aquellos clientes que se encuentren en un nivel de IC 0[62] se le reasigna este valor al inicial. Con esta modificación se consigue que, tras un periodo de bonanza, no todos los clientes se encuentren con el IC a 0 y por lo tanto los cambios de calidad se realicen en diferentes instantes de tiempo. La consecuencia inmediata de esta diferenciación temporal es evitar que todos realicen los cambios simultáneamente, ahorrándoselo a alguno de los clientes.

[62] Mejor caso posible.

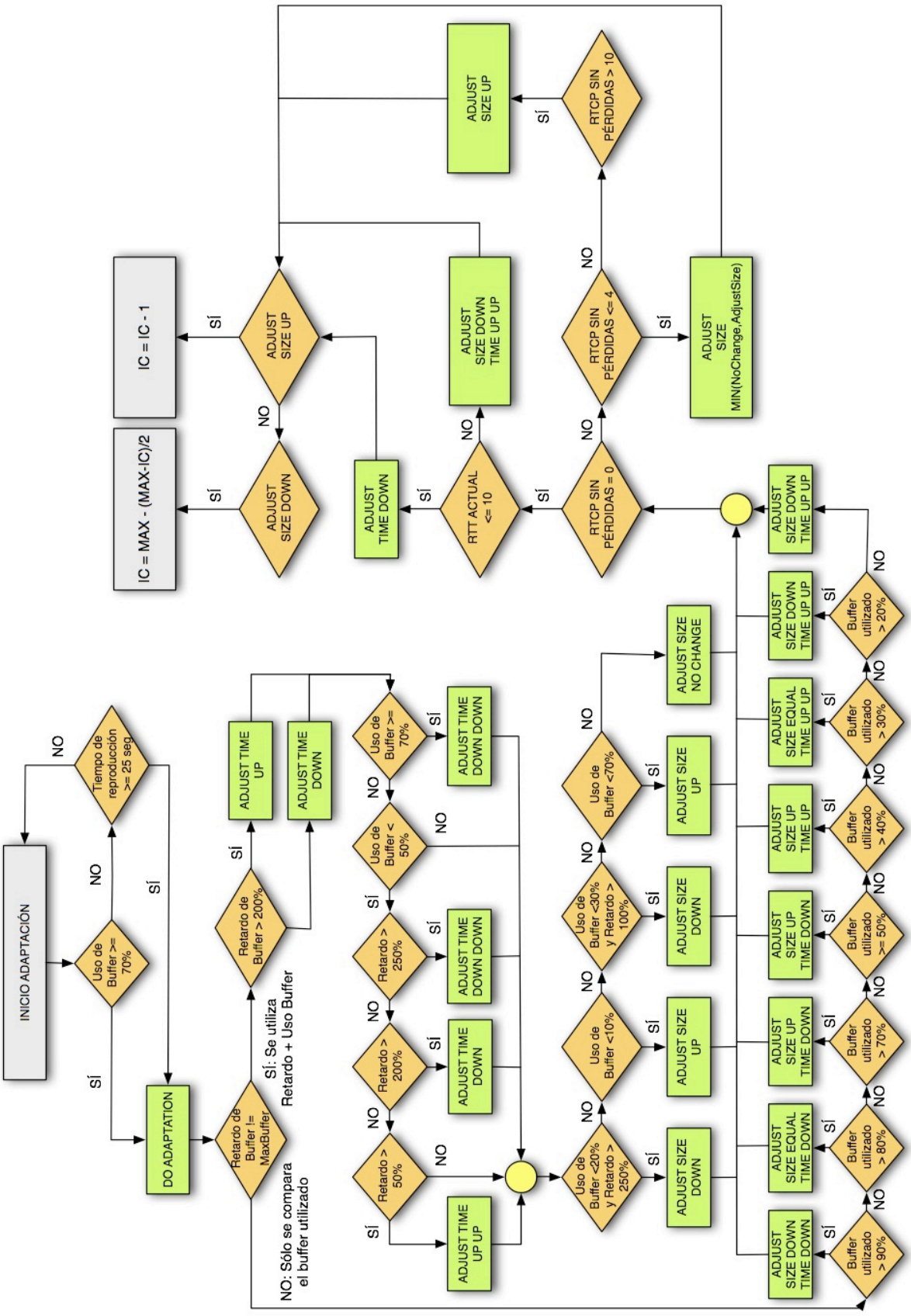

Figura 23: Lógica del nuevo algoritmo de adaptación.

6.2.11. Intercambio entre pistas: La función *SwapTracks*

Para realizar el intercambio de pistas se ha definido una función denominada *SwapTracks*, la cual funciona con un parámetro que indica si el cambio tiene que ser hacia una pista alternativa de mayor o menor tasa; su finalidad es detectar las pistas activas para cada sesión, desactivarlas y activar las alternas que correspondan dotando a éstas de todos los valores necesarios para una transmisión transparente hacia el cliente. Consta de cuatro partes fundamentales:

- Identificación de pistas activas: Imprescindible para realizar el intercambio, se basa en hacer un barrido de las pistas pertenecientes a la sesión, identificar las activas, y posteriormente las alternas, que serán aquellas ordenadas inmediatamente antes o después de la actual en términos de ancho de banda según convenga; todo ello utilizando las nuevas variables presentadas en el punto 6.2.6.

- Asignación de variables: Para evitar errores a la hora de transmitir paquetes tales como datos inexistentes o números de secuencia y/o de muestra a nivel interno erróneos se debe asignar a la nueva pista todos estos valores a partir de aquellos existentes en la pista activa. Ello se consigue mediante simples asignaciones de la antigua estructura de datos a la nueva.

- Activación y desactivación de pistas: Una vez transferidos los valores es importante que únicamente queden marcadas como activas aquellas pistas de las que se desea transmitir paquetes, de lo contrario la función pertinente en el servidor lanzará paquetes de todas ellas.

- Sincronización: Para que los datos transmitidos de la nueva pista sean aquellos pertenecientes al instante de tiempo deseado[63] y la reproducción quede lo más suave posible, debe realizarse un ejercicio de sincronía de flujos; para ello es posible utilizar la función existente *Seek* la cual desplaza un flujo a un instante de tiempo determinado[64]. Dicha función se llama pasando como parámetro temporal el denominado *CurPacketTime* de la pista antigua que indica el instante de transmisión respecto al archivo del último paquete transmitido.

Estos cuatro pasos deben ser llevados a cabo para tantas pistas como se desee intercambiar. En el siguiente cuadro se muestra el código de la función:

[63] Instante en el que se realiza el cambio.

[64] También existe una función que realiza un desplazamiento hacia un número de paquete concreto.

```
QTRTPFile::ErrorCode QTRTPFile::SwapTracks(Bool16 increaseQ)
{
    UInt32 OldTrackID, NewTrackID;
    RTPTrackListEntry *oldTrackEntry = NULL;
    RTPTrackListEntry *newTrackEntry = NULL;
    RTPTrackListEntry *curEntry = NULL;
    UInt16 updated[2] = {0,0};

    for ( curEntry = fFirstTrack;curEntry != NULL;curEntry = curEntry->NextTrack)
        {
        if (IsActiveTrackID(curEntry->TrackID) && updated[curEntry->Payload]==0)
                {
                OldTrackID = curEntry->TrackID;
                NewTrackID = increaseQ==true?curEntry->NextQTrackID:curEntry->PrevQTrackID;
                if (NewTrackID==0 || OldTrackID == NewTrackID)
                continue;

                if( !this->FindTrackEntry(OldTrackID, &oldTrackEntry) )
                  return fErr =  errTrackIDNotFound;

                if( !this->FindTrackEntry(NewTrackID, &newTrackEntry) )
                 return fErr =  errTrackIDNotFound;

                ::memcpy((char *)newTrackEntry->CurPacket + 2, (char *)oldTrackEntry->CurPacket +
2, 2);
                //
                // Read the sequence number right out of the packet.
                for(int i=0; i<QTRTPFILE_MAX_PACKET_LENGTH; i++)
                        {
                        newTrackEntry->CurPacket[i]=oldTrackEntry->CurPacket[i];
                        }

                newTrackEntry->QualityLevel = oldTrackEntry->QualityLevel;
                newTrackEntry->Cookie1  = oldTrackEntry->Cookie1;
                newTrackEntry->LastSequenceNumber = oldTrackEntry->LastSequenceNumber;
                newTrackEntry->SequenceNumberAdditive = oldTrackEntry->SequenceNumberAdditive;
                newTrackEntry->CurPacketNumber = oldTrackEntry->CurPacketNumber;
                newTrackEntry->CurSampleNumber = (oldTrackEntry->CurSampleNumber);
                newTrackEntry->SSRC = oldTrackEntry->SSRC;

                oldTrackEntry->IsTrackActive = false;
                oldTrackEntry->IsPacketAvailable = false;
                newTrackEntry->IsTrackActive = true;
                newTrackEntry->IsPacketAvailable = true;
                Seek(oldTrackEntry->CurPacketTime, 0);

                return errNoError;
                }
        }
return errNoError;
}
```

6.3. Pruebas en el sistema

Este apartado incluye algunas de las pruebas realizadas sobre el servidor modificado con el fin de evaluar su bondad y ajuste a los requisitos iniciales. En él se describen el escenario general de pruebas utilizado, aquellas pruebas relativas a la adaptación de contenido en función de las capacidades de los terminales y finalmente una comparativa entre el comportamiento del sistema, de la red y calidad del usuario entre los tres sistemas de adaptación presentados con anterioridad: Sin adaptación, con el algoritmo del servidor por defecto y con la implementación realizada que permite el intercambio de pistas.

6.3.1. Escenario

Las pruebas realizadas tienen como escenario una red mallada compuesta por los dispositivos *InfoPoints* de *Futurlink*. En cada uno de ellos se encuentra instalada la última versión del servidor, un servidor web *Apache* y un contenedor de *servlets Tomcat;* además, uno de ellos tiene una base de datos de perfiles de usuario sobre un servidor *postgres* en la que se almacenará la información de los terminales que vayan accediendo a la red. Puede apreciarse la estructura de los dispositivos en la Figura 24.

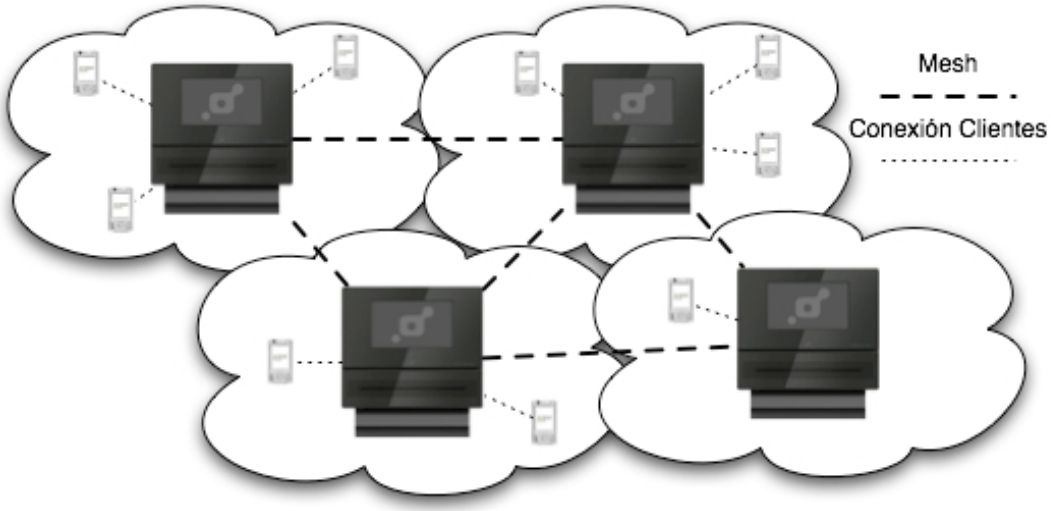

Figura 24: Estructura de la red

Cada uno de los dispositivos tiene un punto de acceso WLAN al que se asocia un número determinado de clientes. Pese a que los puntos de acceso son independientes, estos clientes tienen acceso al contenido de las diferentes máquinas gracias a la red mallada. Una vez que un cliente está asociado a un punto de acceso y accede a un recuro web, el portal le redirecciona al software detector que, tras analizar su perfil, lo introduce en la base de datos correspondiente. A partir de ese momento el cliente está identificado en la red.

Además del sistema de red mallada, existe un sistema de gestión de movilidad que permite, mediante una serie de túneles, que un cliente que realiza un traspaso entre dos puntos de acceso no pierda conectividad en el proceso, permitiendo la transmisión ininterrumpida de vídeo.

Para la realización de las pruebas se han utilizado diferentes terminales, en concreto tres modelos Nokia (N80, N91 y N95) y dos modelos HTC con Windows Mobile 5 y 6 respectivamente. Las características de los terminales pueden ser consultadas en las páginas de los fabricantes, pero es importante destacar que los tres modelos Nokia sí son compatibles con 3GPP Rel6 mientras los HTC no. Además de los terminales móviles se han conectado algunos dispositivos como portátiles tanto vía inalámbrica como cableada con el fin de realizar una mejor evaluación de las características de la red mediante análisis de paquetes y permitir realizar algunas capturas de pantalla. Aunque típicamente el ancho de banda disponible en las redes inalámbricas es de 54Mbps, en algunos casos se ha limitado este valor con el fin de lograr una saturación de la red con menos terminales.

6.3.2. Adaptación de contenido a las características del terminal

Para llevar a cabo la evaluación del sistema, se han tomado diversos terminales y se ha implementado una lógica de decisión que permita discriminar las características de los mismos como se muestra en la Figura 13. En el servidor de vídeo se han almacenado tantos vídeos como posibles resultados del algoritmo. Las pruebas realizadas se basan en comprobar que el servidor otorga un vídeo distinto a cada cliente antes y después de registrarse y entre sí.

Para comprobar el vídeo entregado por defecto, se conectan diversos terminales que solicitan un mismo recurso de vídeo sin pasar por el proceso de detección, verificando que a todos ellos se les entrega el vídeo correspondiente a las características básicas (lo.3gp). Posteriormente se realiza un proceso de detección para cada uno de ellos y se repite la operación. El resultado es que a cada terminal se le entrega un vídeo diferente al inicial y al del resto de terminales demostrado que tanto el sistema detector como el módulo del servidor de vídeo interactúan correctamente con la base de datos y como éste último interpreta bien las características de cada uno. Igualmente se demuestra que el proceso es completamente transparente para el cliente ya que en todo momento se realiza la misma petición por su parte la cual, pese a ser modificada internamente por el servidor, no supone ningún problema para la correcta ejecución del terminal.

6.3.3. Adaptación de tasa durante la transmisión

En este apartado se pretende lograr un escenario en el que los usuarios conectados a un punto de acceso saturan la capacidad de la red inalámbrica, produciéndose así pérdidas de paquetes y colisiones. Una vez que se logra saturar la red, se pretende evaluar la capacidad del sistema para adaptar la tasa a los usuarios de forma que no se supere el ancho de banda disponible en el punto de acceso. También pretende evaluarse la calidad de experiencia del usuario desde un punto de vista práctico comparando la calidad de imagen y fluidez en cada uno de los escenarios.

En este caso particular, para lograr la saturación de la red, se ha configurado un punto de acceso WLAN con servidor DSS a 1Mbps y se han conectado hasta tres terminales móviles simultáneamente, dos de ellos compatibles con 3GP-Rel6 y uno no. Todos los terminales acceden a

un mismo archivo de vídeo de múltiples pistas codificadas a unos 300Kbps y 150Kbps de promedio respectivamente.

Para realizar la evaluación del ancho de banda consumido en casa escenario se coloca un analizador de protocolos (*Wireshark*[65]) en el servidor de vídeo local antes de la salida inalámbrica que informa del ancho de banda lanzado para cada cliente. Debe tenerse en cuenta que los paquetes aparecidos en dicho analizador no corresponden con los que finalmente son lanzados a la red debido al sistema de *buffering* de la misma; se considerará que generalmente, sólo aquellos que se encuentren dentro del 80% de la tasa máxima definida (1Mbps) serán efectivamente transmitidos debido a la eficiencia de este tipo de redes. Se debe considerar, por lo tanto, que toda aquella información que supera los 800bps no encaja dada la capacidad de la red. Finalmente se incluyen algunas capturas tomadas desde uno de los clientes, útiles para evaluar el comportamiento en la pérdida de paquetes en cada uno de los escenarios.

6.3.4. Sin algoritmo de adaptación

En el primer escenario cada uno de los clientes va conectándose secuencialmente. Como puede apreciarse en la Figura 25 el acho de banda es repartido equitativamente, esto se debe a la naturaleza del protocolo UDP utilizado para la transmisión de RTP el cual transmite de forma equiprobable todos los paquetes que recibe en cola, existen estudios al respecto como el realizado en [24]. El resultado es una serie de clientes a los que el servidor encola todos los paquetes pertenecientes a cada sesión por igual; dado que la suma de todos los flujos supera con creces el límite de la red, estos paquetes son descartados o colisionan con lo que los clientes perciben una pérdida de paquetes más o menos aleatoria. A partir del instante de conexión del tercer cliente (t=140s), la tasa entregada total supera ampliamente el megabit por segundo iniciándose el proceso de pérdida de paquetes para todos los clientes.

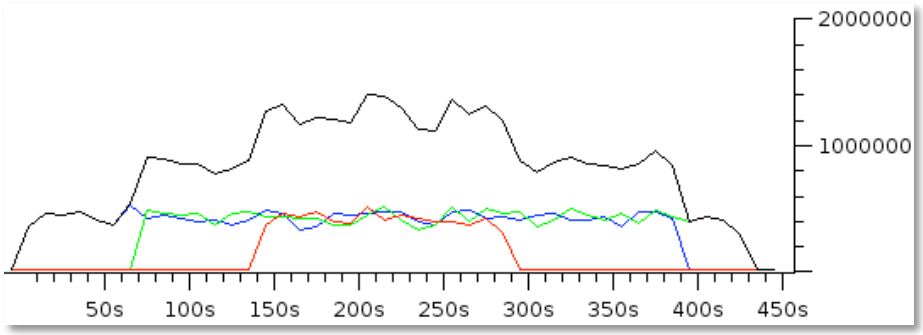

Figura 25: Ancho de banda utilizado sin algoritmo de adaptación

Desde el punto de vista de la red, un sistema sin adaptación supone un perjuicio igual para todos los usuarios de la red cuando la tasa máxima es alcanzada ya sea por un gran número de clientes, clientes que se desplazan a zonas de menor cobertura, etcétera. Desde el punto de vista del usuario, este tipo de saturación implica dos problemas fundamentales: Por una parte la posibilidad de desconexión por *buffer underrun* al no transmitirse una tasa mínima para mantener el *buffer* del

[65] www.wireshark.org

terminal con datos. Por otra parte la pérdida de paquetes aleatorios y, por lo tanto, tramas aleatorias que pueden ser del tipo I, P o B. Para un cliente, la pérdida de una trama I supone mucho mayor perjuicio que la de otro tipo de trama ya que otras tramas dependen de ella, afectando gravemente a la calidad de reproducción como puede apreciarse en la Figura 26.

Figura 26: Calidad de imagen en un escenario sin adaptación

Se demuestra por lo tanto, que en un escenario tan delicado como una red inalámbrica con múltiples clientes de streaming es imprescindible contar con un sistema de adaptación de tasa para evitar tanto desconexiones de clientes como graves degradaciones en la calidad de reproducción de estos.

6.3.5. Con algoritmo de adaptación por defecto

En esta ocasión se reproduce el escenario anterior pero en esta ocasión activando el sistema de adaptación por descarte de tramas que incluye por defecto DSS 6.0.3. En la Figura 27 puede apreciarse como el sistema efectivamente realiza una corrección del ancho de banda entregado a cada cliente en el instante en el que la tasa de transmisión soportada es alcanzada. En particular en el instante que un segundo cliente accede a la red (t=50s), al encontrarse cerca del límite, se realiza una corrección en uno de los clientes (línea verde) mientras que el otro mantiene la calidad. El hecho de que la adaptación se realice en un cliente u otro depende del estado de *buffer*, retardo e informes enviados por cada uno de ellos.

En el gráfico puede apreciarse como se agrega un tercer cliente sin capacidad de adaptación (línea roja) el cual no varía su tasa durante su estancia en la red. Puede verse también como existen diferentes niveles de calidad que fluctúan con cierta rapidez (t=160-210s) debido a que al incrementarse la calidad rápidamente se topa con el límite de la red y vuelven a producirse problemas. No es hasta que el tercer cliente abandona la red (t=230s) cuando los otros dos terminales inician un proceso de recuperación con ciertos altibajos hasta llegar de nuevo a los niveles de mayor calidad.

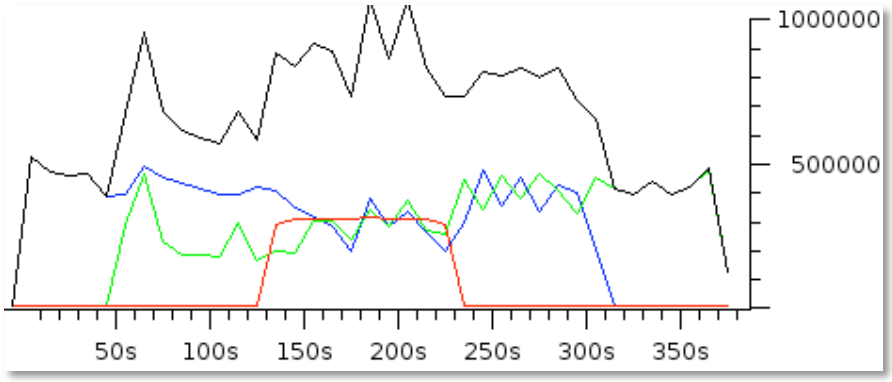

Figura 27: Ancho de banda utilizado con el algoritmo de adaptación por defecto

Una de las principales ventajas del sistema desde el punto de vista de ancho de banda, es la agresividad con la que se realizan tanto los incrementos como los decrementos de calidad. Pese a que el segundo caso está más justificado, el primero únicamente conduce a nuevas pérdidas en la red que pueden perjudicar al usuario. Debe tenerse en cuenta que es preferible una pérdida controlada de una trama B que una incontrolada de una I.

Desde el punto de vista de cliente, la calidad de experiencia de este escenario, pese a ser mejor que el anterior, no termina de ser satisfactoria. Pese a que la calidad de imagen durante todo el periodo es excelente[66] existen problemas de reproducción. Durante todo el periodo que los vídeos no se están reproduciendo con el máximo nivel de calidad el usuario aprecia una evidente pérdida de fluidez, especialmente molesta en los niveles más bajos. En una red saturada, un cliente puede encontrarse reproduciendo todo un vídeo completo a la mínima calidad, resultando en una tasa de marcos por segundo muy inferior a la del vídeo. Como demuestran algunos artículos [18] existe un compromiso entre calidad de imagen y fluidez en el que generalmente es preferible perder la primera.

6.3.6. Con algoritmo de adaptación por intercambio de pistas

En esta ocasión se repite el escenario anterior con el nuevo sistema de adaptación por intercambio de pistas activado. Con el fin de realizar una comparativa clara entre este sistema y el anterior se ha optado por desactivar el sistema de descarte de tramas aunque es posible combinar ambos. De nuevo el cliente que no soporta adaptación viene definido por la línea roja.

[66] El vídeo se transmite con la codificación inicial y únicamente tramas I lo que desemboca en imágenes de buena calidad a una tasa muy baja.

En la Figura 28 puede apreciarse la evolución del ancho de banda consumido en la red durante la transmisión de los vídeos. Inicialmente los terminales con soporte de adaptación acceden a la red y realizan un ajuste a la baja de la calidad de transmisión, esto se debe a la pérdida de algunos paquetes debido a que en un momento determinado la tasa se acerca peligrosamente al límite de transmisión de la red. Posteriormente cada uno de ellos, al comprobar la estabilidad de la red recupera el nivel de calidad alto hasta la llegada del tercer terminal (t=110s). Durante ese periodo, los dispositivos compatibles con 3GP-Rel6 se mantienen en el nivel de calidad bajo (150Kbps aprox.) mientras el incompatible mantiene la transmisión a máxima calidad. Puede observarse como los terminales verde y azul intentan periódicamente un incremento de calidad debido a la bonanza de la red, no obstante inmediatamente vuelven al nivel inferior dado a la agresividad del algoritmo y la proximidad a la zona crítica de capacidad.

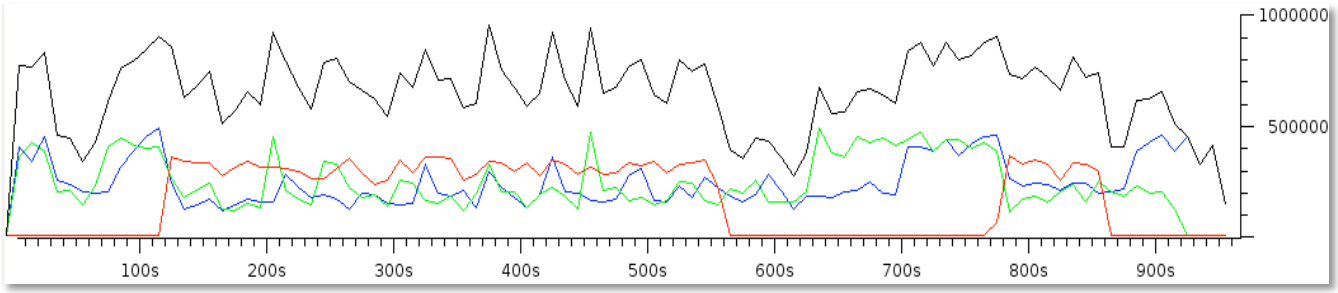

Figura 28: Ancho de banda utilizado con el algoritmo de intercambio de pistas

Una vez el cliente abandona la red (t=550s) los clientes compatibles aumentan progresivamente el nivel; el tiempo que tarda cada uno en recuperar el nivel depende de las características del mismo como capacidad de *buffer* y los paquetes que se pierdan en cada caso. Mientras que en este caso es el terminal verde el primero que se recupera, la segunda vez que el terminal sin adaptación abandona la red (t=850s) es el azul quien inicia el incremento antes.

Este algoritmo tiene dos puntos especialmente relevantes: Por una parte en ningún momento se supera la barrera de transmisión fijada (1Mbps), además ambos clientes compatibles reaccionan inmediatamente a los problemas ocurridos en la red reduciendo su tasa para acomodar tanto a los recién llegados como al resto de clientes existentes. El otro punto positivo es la velocidad con la que realiza el cambio, fluído y totalmente transparente para el usuario salvo por la diferencia de calidad que puede apreciarse en la Figura 29 donde las imágenes de la izquierda pertenecen al nivel alto de calidad (300Kbps) y las de la derecha al bajo (150Kbps).

El hecho de que el algoritmo tome un tiempo antes de realizar un incremento de calidad evita un escenario con pérdidas innecesarias mejorando así la calidad de experiencia del usuario debido a que es preferible aguantar unos instantes más a baja calidad antes de realizar una fluctuación rápida que implique pérdidas.

Figura 29: Comparativa entre los diferentes niveles de calidad

Dada la velocidad y fluidez con la que efectúa el cambio esta solución resulta mucho menos agresiva de cara al cliente que el descarte de tramas realizado por el algoritmo original. Desde el punto de vista de cliente es mucho más notable un vídeo codificado en alta calidad con una muy baja tasa de marcos que uno codificado a una calidad menor con mejor fluidez.

7. Conclusiones

Este documento ha introducido la problemática relativa a la transmisión de vídeo a terminales móviles en un entorno inalámbrico 802.11 tanto a nivel de uso de la red como de experiencia de usuario. Se ha incluido un pequeño estudio del estado del arte en cuanto a adaptación de tasa para la transmisión de vídeo *streaming* en redes inalámbricas. En este estudio se concluye la necesidad de una implementación de un sistema de adaptación efectivo, abierto y compatible con el mayor número posible de terminales.

Se han presentado el marco PSS para la transmisión de contenidos en redes inalámbricas de paquetes y sus diferentes versiones por parte del grupo 3GPP. En particular se han mostrado las principales características de la *Relsease* 6 incluyendo compatibilidad con archivos de múltiples pistas y las herramientas útiles para evaluar la calidad de experiencia y recepción por parte del cliente. Además se ha adjuntado un estudio sobre los perfiles de usuario, útiles para adaptar el contenido transmitido a las características de los terminales. También se ha añadido una descripción de los diferentes protocolos implicados en la transmisión de multimedia con requisitos de tiempo real. Dentro del abanico de protocolos destacan los mensajes especiales RTCP utilizados para notificar el retardo de la reproducción y el grado de utilización del *buffer*; estas herramientas resultan claves para poder realizar una adaptación de tasa efectiva. Finalmente se ha presentado la estructura típica que sigue una comunicación entre un cliente y un servidor de *streaming* de vídeo.

De los estudios realizados se ha llegado a una conclusión fundamental. Es necesario y factible la implementación de un sistema de *streaming* de vídeo capaz de adaptar contenidos a las características de diferentes terminales y de adaptar la tasa de transmisión según las circunstancias del cliente y la red.

Para seleccionar una plataforma de trabajo, se ha hecho una evaluación entre los principales servidores de *streaming* que han cumplido los requisitos para el proyecto. Tras describir sus principales características, pros y contras, se ha tomado la decisión de realizar las implementaciones del proyecto sobre Darwin Streaming Server ya que se muestra como el más apropiado. Una vez seleccionado el servidor se han presentado sus principales posibilidades incluyendo configuraciones, características, sistemas de *relay*, listas de reproducciones, transmisiones multidifusión, etcétera.

Además de presentar esas características, se ha explicado el protocolo para el desarrollo de módulos en DSS; se comentan las estructuras que deben tener, las principales funciones, archivos y estructuras implicadas y el sistema de roles utilizado. Del mismo modo se ha descrito la manera para integrar dichos módulos en la ejecución del servidor.

Una vez presentado el servidor, sus posibilidades y características, se han incluido un capítulo con las implementaciones realizadas sobre él. En primer lugar se ha implementado un módulo capaz de interactuar con una base de datos que contiene perfiles de usuario. Gracias a esta interacción el módulo es capaz de conocer las capacidades del cliente y, ante una misma petición, distribuir un contenido diferente dependiendo de sus características. Se han mostrado un ejemplo de lógica de decisión basado en los terminales disponibles y un sistema de modificación de peticiones transparente para el usuario.

El segundo gran bloque de implementaciones permite la adaptación de tasa en tiempo real durante la reproducción de contenidos. Tras un estudio de las posibilidades del servidor y las soluciones y estándares disponibles, se han definido una serie de modificaciones que hacen posible

que el servidor trabaje con archivos de múltiples pistas, entregue al cliente aquellas que considere más oportunas según el estado de la red y vaya modificándolas según varíen estas condiciones. Se ha presentado el concepto de gestión de múltiples pistas alternativas para un archivo de forma transparente para el cliente; también cómo realizar toda la interacción como si se tratara de un archivo clásico don dos únicas pistas. Finalmente se presenta un algoritmo de adaptación basado en la evaluación de la calidad de transmisión a partir de los mensajes del cliente. Gracias a este algoritmo ha sido posible trabajar con archivos de múltiples parejas de pistas y entregar cada una de ellas en el instante adecuado.

Finalmente se muestra una serie de pruebas realizadas que han demostrado la utilidad de las variaciones introducidas en el servidor. En primer lugar el sistema de interacción con la base de datos de usuarios permite asignar contenido adaptado a las características del terminal en cada momento. Esta característica incrementa su calidad de experiencia y permite un mayor abanico de contenidos para la distribución. Se ha demostrado que esta característica es fácilmente implementable en el servidor de una forma sencilla y resulta en todo caso transparente para el usuario respetando todos los protocolos implicados en la comunicación.

En segundo lugar, las pruebas realizadas en los escenarios de adaptación de tasa también han demostrado un correcto funcionamiento con un importante número de terminales compatibles. Por una parte se ha comprobado que la adaptación de tasa permite autorregular el uso del ancho de banda en una red inalámbrica 802.11 de una manera automática, adaptando el tráfico de la misma a un nivel inferior al máximo soportado. Esta regulación se efectúa de manera natural gracias a la información proporcionada por los propios clientes siguiendo los protocolos definidos. Por otro lado, en caso de necesidad, los intercambios entre diferentes niveles de calidad se realizan de una manera rápida y transparente casi imperceptible para el usuario. Esta fluidez unida al hecho de que no se efectúa ningún descarte de tramas desemboca en una mejor experiencia para el cliente. Finalmente el algoritmo ha demostrado comportarse correctamente permitiendo la permanencia de un mayor número de usuarios en la red. El principal inconveniente que se puede extraer es la todavía no existencia de un protocolo para informar al cliente del cambio de pista, obligando a una modificación significativa del servidor de *streaming*. Esta definición de protocolos, o una solución análoga que permita trabajar con menores modificaciones en el servidor, junto con un estudio exhaustivo de los parámetros para su integración con diferentes escenarios, son las principales líneas de investigación futuras a considerar.

Bibliografía

[1] 3GPP TR 26.937. "Transparent end-to-end packet switched streaming service (PSS): RTP usage model"; March 2004.

[2] A. Furuskar, S. Amzur, F. Muller and H. Olofsson, "EDGE, Enhanced Data Rates for GSM and TDMA/136 Evolution", IEEE Personal Communications 6(3) (June 1999) 56-66.

[3] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson. "RTP: A Transport Protocl for Real-Time Applications". RFC 3550, IETF, July 2003.

[4] H. Schulzrinne, A. Rao, and R. Lanphier. "Real Time Streaming Protocol (RTSP)". RFC 2326, IETF, April 1998.

[5] M. Handley, V. Jacobson. "SDP: Session Description Protocol". RFC 2327, IETF, April 1998.

[6] R. Fielding, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. "Hypertext Transfer Protocol (HTTP/1.1)". RFC 2616, IETF, June 1999.

[7] 3GPP TS 26.234. "Transparent end-to-end Packet-switched Streaming Service (PSS); Protocols and codecs (Release 6)". April 2006

[8] 3G TS 26.071 v 3.0.1 (1999-08), "3rd Generation Partnership Project; Technical Specification Group Service and System Aspects; mandatory speech codec speech processing functions AMR speech codec; General Descrip tion (3G TS 26.071 v 3.0.1)"

[9] ISO/IEC. "Information technology - coding of audio-visual objects - part 14: Mp4 file format". ISO/IEC 14496--14: 2003, Apr. 2004.

[10] International Telecommunication Union, "Video coding for low bit rate Communications", ITU-T Recommendation H.263 version 2, January 1998.

[11] ITU-T Recommendation H.264 & ISO/IEC 14496-10 "AVC, Advanced Video Coding for Generic Audiovisual Services", v3: 2005.

[12] ETSI. "Narrowband coding of speech at around 16 kbps using Adaptive Multi-Rate Narrowband (AMR-NB)", 1998.

[13] ISO/IEC IS 13818-7, "Information Technology--Generic Coding of Moving Pictures and Associated Audio, Part 7: Advanced Audio Coding, AAC," 1997.

[14] ISO/IEC 14496-12:2003. "Information technology -- Coding of audio-visual objects Part 12: ISO Base Media File Format".

[15] UAProf Specification -- http://www.wapforum.org/what/technical/ SPEC-UAProf-19991110.pdf

[16] 3GPP TS 23.140: "Multimedia messaging service (MMS); Functional description; Stage 2".

[17] Apple Computer Inc. "QuickTime Streaming Server Modules Programming Guide". http://mages.apple.com/quicktime/pdf/QTSS_Modules.pdf 2002-2005

[18] O. Verscheure, P. Frossard and M. Hamdi, "MPEG-2 video Services over Packet Networks: Joint Effect of Encoding Rate and Data Loss on User-Oriented QoS". 8th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV' 98. Cambridge, (United Kingdom), 1998.

[19] X. Sanchez-Loro, J. Casademont, J. L. Ferrer, J. Paradells. "A Proxy-based Solution for Device Capabilities Detection". Internet and Multimedia Systems and Applications – EuroIMSA 2007. Chamonix (France) 2007.

[20] T. Schierl, T. Wiegand, M. Kampmann. "3GPP compliant adaptive wireless video streaming using H.264/AVC," Image Processing, 2005. ICIP 2005. IEEE International Conference on , vol.3, no., pp. III-696-9, 11-14 Sept. 2005

[21] T. Schierl and T. Wiegand: "H.264/AVC Rate Adaptation For Internet Streaming," Packet Video Workshop, December 2004

[22] N. Cranley, L. Murphy, P. Perry. "Perceptual quality adaptation (PQA) algorithm for 3GP and multitracked MPEG-4 content over wireless IP networks," Personal, Indoor and Mobile Radio Communications, 2004. PIMRC 2004. 15th IEEE International Symposium on , vol.3, no., pp. 2107-2112 Vol.3, 5-8 Sept. 2004

[23] R. Malik, A. Munjal, S. Chakraverty. "Adaptive Forward Error Correction (AFEC) based Streaming using RTSP and RTP," Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on , vol., no., pp. 60-60, 19-25 Feb. 2006

[24] L. Cicco, S. Mascolo, V. Palmisano. "An Experimental Investigation of the End-to-End QoS of the Apple Darwin Streaming Server. ". Int. Conf. on Wired/Wireless communications, WWIC 2008. Tampere (Finland), 2008.

[25] P. Frojdh, U. Horn, M. Kampmann, A. Nohlgren, M. Westerlund, "Adaptive streaming within the 3GPP packet-switched streaming service," IEEE Network, vol.20, no.2, pp.34-40, 2006.

[26] T. Yala; L. Yuanxiang; G. Xiaoming, "A Novel Method of Dynamic Rate Adaptation for Streaming Media Server," Wireless Communications, Networking and Mobile Computing, 2006. WiCOM 2006.International Conference on , vol., no., pp.1-4, 22-24 Sept. 2006

Anexo I: Artículo redactado durante la elaboración del PFM.

Sometido al Fourth Workshop on multiMedia Applications over Wireless Networks (MediaWiN 2009) IEEE Symposium on Computers and Communications (ISCC 2009) Túnez.

Rate adaptation in 3GPP video streaming using track switching over a multihop WLAN

Eliseo Catalán Jordi Casademont Xavier Sánchez-Loro Josep Lluís Ferrer

Technical University of Catalonia

ecatalan@gmail.com jordi.casademont@upc.edu xsanchez@entel.upc.edu jlferrer@entel.upc.edu

Abstract

In nowadays communication technologies, one of the main challenges to overcome is how to translate theoretical algorithms, designed protocols, existing hardware products from different manufacturers and the already knowhow into innovative applications that can produce a benefit for the user and for the commercializing firm. Following this topic, the present paper introduces an application that combines different emerging technologies: wireless mesh networks, multihop routing and adaptation of streaming delivery with the objective to create a tool for effective marketing campaigns.

1. Introduction

At present time, many people carry in their pockets sophisticated communication tools that are able to do much more than their usual role: talk by phone. Many of our mobile phones, apart from their cellular 3G interface, are equipped with other network technologies as IEEE 802.11 and Bluetooth. The objective of the application presented in this paper is to make use of these facilities implementing a coordinated network of local information points that would give spaces such as airports, tourist areas, shopping malls, open environments, and so on extra information capabilities due to the higher available bandwidth of these technologies. Once the user enters under the coverage area of the network it will be able to access personalized and adapted video content. For instance, if the situation is a shopping mall, users get multimedia content of the different shops on the mall.

Another aspect to consider is the maximization of user's quality of experience (QoE) making use of an efficient distribution system, able to adapt multimedia data to the capabilities of user's devices and network conditions. In video transmission, QoE can be basically mapped to the quality and smoothness of the video play. Adapting the transmitted content to network circumstances leads to a better experience for the user, especially when working over WLAN.

WLAN are especially difficult to plan because available bandwidth varies a lot depending on the number of users or propagation distances. They also provide no guarantee in terms of QoS (Quality of Service), i.e. bandwidth, delay or jitter. This variability leads to a low QoE when trying to distribute fixed bitrate streaming video over multicellular WLAN with mobile users. Nevertheless, nowadays mobile devices are able to provide information related to buffer usage or existing delay using latest 3GPP Releases [1] and loses using RTCP messages. The server can use this feedback to adapt the content transmitted to the client.

Latest Darwin Streaming Server release provides 3GPP-Rel6 compatibility and performs some rate adaptation by removing specific frames. This adaptation solves buffering problems in client side and helps to reduce network traffic and packet losses, but does not avoid QoE problems, as the video is not fluently played due to frame drops.

As a single 3GP file supports the inclusion of multiple tracks containing the same video but with different bitrates, we have developed a track swapping system, which allows the server to switch between tracks in a total transparent way for the client. The goal is to adapt active connections bitrate according to network conditions to benefit the whole network. One of the main challenges is to make this track-switching system compatible with as many commercial clients as possible using existing an open source streaming server. Additionally, it is necessary to develop a module that provides information about user's device capabilities.

This document is structured in four major parts; firstly it introduces system's architecture, secondly the involved technology, thirdly it describes the implementation, then practical results and finally conclusions.

2. System architecture

The developed application is based in a competitive platform (Wilico by Futurlink [2]) that provides the minimal hardware to achieve the expected requirements. This platform is an off-the-shelf Linux computer with IEEE 802.11 and Bluetooth interfaces designed to work as an Access Point with high capabilities and distribute multimedia content. In the present application, Wilico nodes are programmed with software that enables them to set up a multihop wireless network able to relay data between them using the AODV protocol, they can also manage client mobility redirecting the data to the node where the client is associated. There are two different networks: the access network between user's devices and Wilico nodes, and the backbone network to connect relay nodes. Both networks use 802.11 interfaces with different channels (Figure 1).

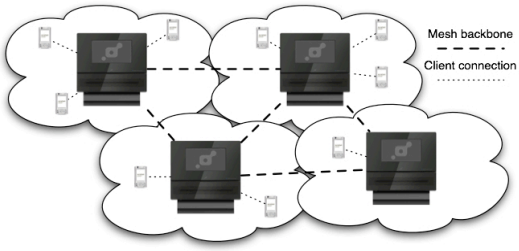

Figure 1: Multihop network.

Another point is the fact that when a user enters for the first time into the system, there are some tools to discover many of the capabilities and characteristics of his device [3]. The detector module is set in one of the Wilico nodes and using the multihop network interrogates new clients. It obtains the maximum possible information about their hardware and software characteristics as well as to completely characterize the used browser in order to create a consistent, valid and parsable profile. Detector module executes different detection mechanisms depending on the user's device. The module makes all possible tests on the device and then it matches the results. This matching gives preference to active tests (Applet, JavaScript) in front to passive ones (HTTP analysis, CC/PP-UAProf negotiation).

The final required module is the streaming server. This one can be installed in any of the Wilico nodes and stream video content to all clients using the backbone network. Nevertheless, in order to increase system performance, we have installed one streaming server in each Wilico node that only serves its associated clients.

3. Involved Technology

3.1. PSS: Packet-Switched Stream Service

Proposed by 3GPP, PSS specification scope is to provide a framework for streaming applications over 3G data networks. Different PSS versions have been released so far, the Release-6 is especially relevant because it includes tools to perform rate adaptation in mobile transmissions. Although this framework was intended to work in cellular telephony scenarios, the protocols and solutions implemented include any kind of network such as WLAN.

3.2. 3GP Multimedia Container

3GPP also proposed a multimedia container format to provide a standard container for different available mobile devices. 3GP video files are very similar to MPEG-4 part 14. Multiple video coding schemes can be found inside this container, typically MPEG-4 part 2, H.263 and H.264 for video and AMR NB/WB, AAC for audio. Those codifications were chosen considering the typical narrow bandwidth that is available in packet-switched networks over cellular networks.

3.3. ISO Boxes

3GP files are based in ISO Boxes, like MP4. These boxes contain not only audio and video tracks, but also additional control information. They are ordered hierarchically and divided in two major groups as shown in Figure 2.

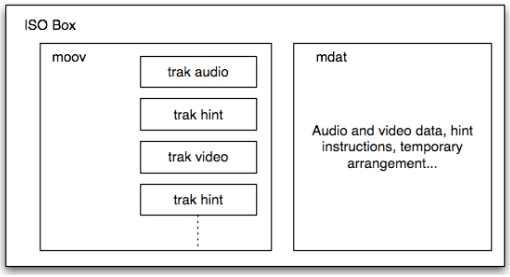

Figure 2: Example of ISO Boxes.

The first group, called *moov,* is related to the movie and stores as many *trak* boxes as tracks available in the file, these tracks have many fields used to define its characteristics and properties. Hint tracks are a special kind of *trak* that contains useful information about a video or audio track; the streaming servers use this information to properly package data to the clients. In hinted media there are as many hint tracks as media tracks. In addition of this *moov* group, there is the *mdat* group that contains media information.

Since 3GP 6th Release, one of the fields included in *traks* is the alternate group, that indicates which tracks could be chosen instead of the others.

3.4. RTP/RTCP

RTP (Real-Time Transmission Protocol) is used to transmit media with real-time requirements providing functionalities as timestamps or payload identifiers. Transmitted flows sources are identified using a field called SSRC (Synchronization source); there is also an identifier called CSRC (Contributing source) used to identify contributor sources in scenarios where an intermediate device as a mixer is defined. Each transmitted track has a RTP flow so, typically, in a two-tracked file, two RTP sessions will be started.

The most relevant fields of RTP packets are: Payload Type (PT), that will generally be in 96-127 range reserved for dynamic coding; Sequence Number (SQ), incremented sequentially in each session, usually starts with a random value; Timestamp, that indicates the sampling time of the first octet in the data, it keeps flow synchronization and it's useful to evaluate jitter.

Information provided by RTP packets is interpreted by the client and server who generate Receiver Report (RR) and Sender Report (SR) messages respectively; these messages are part of then RTCP (Real Time Control Protocol). RTCP messages include percentage of losses, cumulative losses, latest received sequence number, jitter, last sender/receiver report and delay since last report. In these reports additional fields can be added as, for instance, application specific messages known as APP. The PT field marked with the value 204 identifies these APP messages and then the NAME field identifies the kind of specific data carried.

The APP reports used in 3GPP are known as NADU, which stands for Next Application Data Units and are identified when this name is set to PSS0. Much of the information included in these packets is referent to ADU, which vary depending on the coding of the carried payload.

NADU packets include different fields as SSRC of the reported source; NSN and NUN indicating the RTP sequence number and the number of the next ADU to decode respectively. Free buffer size indicates the available data room in 64 bytes blocks, so a limited size can be announced. Finally the DELAY is useful to calculate the time between the stored data and its playing time.

RTP is transmitted over UDP, as this transport protocol has not delivery guarantee neither network congestion mechanisms, all server sessions are allowed to go into the network with equal probability. The result is a high packet loss when aggregate throughput goes over network capacity [5].

3.5. RTSP: Real Time Streaming Protocol

RTSP is a fundamental part of the protocol stack in PSS transmission as it is used to establish and control sessions, providing a request/reply system similar to HTTP. RTSP uses a well-known TCP port in order to start message exchange; these messages are typically the following:

- OPTIONS is sent by the client in order to obtain all server supported RTSP commands.
- DESCRIBE is sent by the client in order to obtain a description of a video file.
- PLAY and PAUSE are used to start, seek and temporary stop the data flow.
- TEARDOWN is used to finish the session.
- SETUP is used to ask the server for a track;

When a track is setup, clients expect the server to send information with some session parameters: time, sequence number, SSRC and timestamp fields. Some clients have shown more tolerance to malformed packets such as wrong sequence numbers, but in all cases respecting the PT field is mandatory.

3.6. SDP: Session Description Protocol

SDP is the protocol used to describe the file characteristics such as contained tracks, coding, bitrate, length, alternate groups, 3GP adaptation support, ... The client can interpret this information in order to choose its more suitable tracks. In practical situations most terminals ignore this information choosing the first pair of tracks or them all.

3.7. DSS: Darwin Streaming Server

The presented solution is developed over DSS 6.0.3. This server was chosen because is open source, provides a good developing framework and tools, and is 3GPP Rel6 compatible. DSS 6.0.3 introduces a rate-adaptation mechanism using RR and NADU packets based on dropping video frames.

4. Implementation

This point presents the main modifications carried out in the DSS in order to work with most commercial devices and to improve the adaptation algorithm.

4.1. 3GP file generation

The first step of the implementation is the generation of a proper 3GP/MP4 file, with multiple tracks with the same content but coded at different bitrate. Having the same video coded at different bitrates enables the client to continue playing after track-switching. In order to work with DSS, this file must be hinted in advance.

4.2. Describe

Typical client-server dialog (Figure 3) starts when clients ask for a resource, typically a 3GP/MP4 file.

By default, the original DSS sends information of all tracks inside the SDP Session Description message. Nevertheless, many times this is not the best option because clients choose the first track by default, independently of system capacity, device capabilities or track bitrate. Other clients (VLC or QuickTime to

the date releases) ask for all available tracks, downloading simultaneously all of them, as they are not 3GP release 6 compatible.

Our system improves this situation making use of some known information: main capacities of user devices, acquired during the association phase, and network load. The sever has been provided with an algorithm that using this known information selects the most suitable tracks of a requested file for each specific situation. After that, it sends a SDP Session Description message with only the optimum tracks. This solution forces the client to select the tracks that the server algorithm considers most favourable.

4.3. Setup: Adding alternative tracks

When the client receives the SDP Session Description message, it sends one Setup request for each track (usually one for video and one for audio). Then, the server establishes one client session that contains not only requested tracks but also alternative tracks containing the same information but coded at different bitrate. Alternative tracks are those included in the 3GP file that have not been announced in the SDP Session Description message. Alternative tracks may be used for dynamic adaptation if network conditions change during the transmission. All alternative tracks share the parameter values of the original track as SSRC, port, first RTP sequence number, ... These alternative tracks remain inactive so the server does not send packets belonging to them. The response message to the client only includes information related to the active tracks as it won't accept any other information.

Tracks (active and alternative) belonging to a client session are ordered inside the server according to their bitrate. This management is done in basis of some new attributes added to each track.

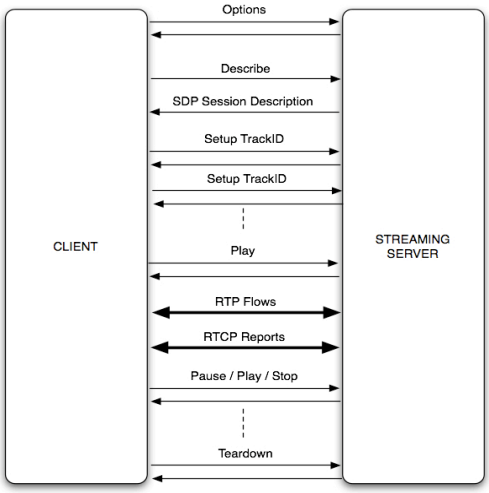

Figure 3: Client-Server Dialog.

4.4. DSS existing algorithm: frame dropping

Latest Darwin Streaming Server 6.0.3 release provides a rate adaptation algorithm based on the transmission of a single high bitrate coded video track that, in case of congestion, is forced to drop a certain percentage of frames in order to decrease the outgoing bitrate [4].

DSS defines 6 different quality levels that have associated different percentages of frame dropping. For instance, the highest quality level drops no frames, a medium level drops 70% of B frames, and the lowest level drops all B and P frames. In order to change the active quality level, DSS applies an algorithm that works with reported losses, RTT values, buffer usage ratio and delay when available. All this information is obtained from NADU and RR packets.

Initially, the algorithm allows a certain amount of time for the client to buffer the video. During this phase the outgoing bitrate is higher than codification bitrate to shorten buffering time, usually the double. After that it starts the adaptation phase.

Another problem is client's buffer management. The buffer can get full (overflow) or run out of data (under-run). Therefore, packet transmission is delayed or advanced according to client buffer feedback. This is controlled by the variable called Transmission Time.

This solution helps dealing with buffering problems such as overflow and under-run on client sides, also it benefits the rest of the network elements modifying the overall transmission rate; the main disadvantage is the poor quality of the displayed video due to frame dropping. The video becomes discontinuous too often and QoE is acceptable but not good.

4.5. Modified algorithm: track switching

The main advantage of the developed adaptation function is that instead of dropping frames of a video flow coded at a high bitrate, we transparently switch tracks coded at lower bitrate. In terms of video quality and smoothness this second situation is much better than the first and results in a higher quality of experience as shown in [6].

As it has been said, DSS algorithm uses 6 quality levels that are mapped to different percentage of frame dropping. Instead, the proposed adaptation procedure works with pre-coded alternative tracks. Each video file is coded several times at different bitrates. Therefore, each quality level is mapped to a different track of the same video with a specific bitrate.

In order to change the quality level, and consequently of video track, we have defined a Quality Indicator (QI). When the QI overcomes a certain threshold, it changes to a higher quality level. When decreases under another threshold it switches to a lower quality level. To prevent repeated bouncing quality level changes in the boundaries of a threshold, QI is modified accordingly to additive increments

when RTCP RR messages report no loses and NADU messages report stable buffers and no delays, and multiplicative decrements when one of the previous conditions is not fulfilled.

The algorithm also controls the Transmission Time to accommodate client's buffers.

4.6. Swapping function

The main problem to perform track switching to mobile terminals like phones or PDA is doing it in a completely transparent way. As introduced before, RTP clients only expect to receive information about the initially described tracks, all RTSP responses and RTP packets must be consistent with that information. To achieve this consistence, a swap function has been designed. The goal of the swap function is to assign the proper values to each track when the switching is performed; it must keep tracks time correct, adjust RTP sequence numbers, Payload Type and Timestamp and reflect that in all RTP packets, and finally manage track activation and deactivation.

5. Practical Experiments

Practical experiments carried out include the comparison of the system performance using: no adaptation at all, default adaptation system based in frame dropping, and track switching adaptation. In order to test a scenario which requires many adaptation situations, we have forced the 802.11 network to work at 1 Mbps. Clients are two 3GPP-Rel6 compatible terminals (Nokia N95 and N80) and one non-compatible. The idea is to compare bandwidth distribution and video quality in each case; smoothness in video playing is also taken into account. The adaptable files include two video tracks (about 300 Kbps and 150 Kbps in average) and one audio track, so only video tracks will be switched if needed; the requested 3GP file is the same for all the terminals.

Following figures show the outgoing of Darwin Streaming Server. Blue and green lines are 3GPP compatible terminals and red line the non-compatible one. These lines represent outgoing data flows generated by the streaming server application belonging to sessions that are established and finished to test the system behaviour. Later, they are forwarded to the 1 Mbps IEEE 802.11 network interface of the streaming server Wilico node. Therefore, if the addition of all of them (black line) is higher than 1 Mbps there will be loses and many delays in the network interface. In fact, transmission problems begin earlier than 1 Mbps due to the 802.11 efficiency.

5.1. Non-adaptation scenario

Non-adaptation scenario presented in Figure 4 shows a fair bandwidth distribution among all clients. All new connections are accommodated in the server without decreasing the bitrate of already established ones. This leads that when the third connection is established (t = 150s) the aggregate output is much higher than 1 Mbps. The consequence is a very poor received video quality: bad image quality, frequent pauses to enter in a buffering period and also disconnections. Main problems are delays and loss of referenced frames that generate a chain of errors in the video play.

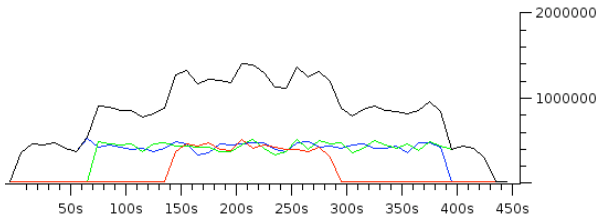

Figure 4: Server output bitrate (bps) - without adaptation.

5.2. Default adaptation scenario.

This scenario (Figure 5) provides adjustments in active session bitrates at server output. When green client establishes a connection (t = 50s), blue client reduces its bitrate. The same happens at t = 130s when red client enters. Note that the aggregate throughput never goes up to 1 Mbps. Server response to small condition variation is very fast, which can lead to a bouncing effect. The main advantage of this system is the fast adaptation to available network capacity and bandwidth equally distribution among clients.

In terms of video quality it has a relatively good behaviour because avoids disconnections due to buffering problems and reduces decoding errors (malformed frames or black pixel groups) due to referenced I frames losses. Nevertheless, QoE suffers from a very noticeable frame dropping and the experience is not very satisfactory.

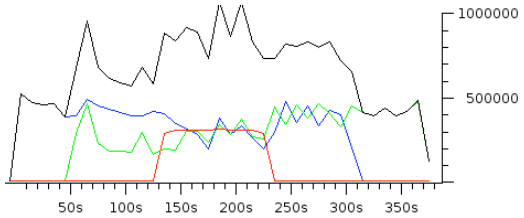

Figure 5: Server output bitrate (bps) - with default algorithm.

5.3. Track-switching scenario

In this scenario track switching and Transmission Time variation are performed. Figure 6 shows a situation similar to Figure 5 where new connections are established and the system adjusts the global outgoing throughput under 1 Mbps. Therefore rate adaptation works properly, like the DSS original adaptation

algorithm. The system avoids bouncing changes, note that quality level decreasing is almost immediate when problems are found (new connection enters t = 770s) but opposite procedure takes a longer time (one client leaves t = 540s).

The main advantage of this method is its higher level of QoE. The transaction from one quality level to another is smooth on the client player, so it is not a harmful situation at all and no forced drops are performed. In situations where bandwidth restrictions apply, it is better to have all frames of a lower quality video, than a higher quality video with many lost frames [6].

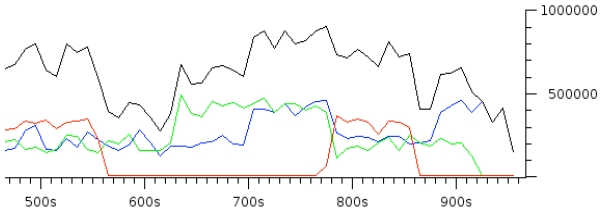

Figure 6: Server output bitrate (bps) - with track-switching algorithm.

6. Conclusion

This paper presents an innovative application that makes use of the newest communication technologies at different protocol architecture levels: new hardware platform (Wilico nodes), multihop IEEE 802.11 mesh network, dynamic AODV routing protocol, application to discover client devices capabilities and finally a modified open source streaming server to adapt video delivery to both device capabilities and network conditions.

Experimental results have shown that the system is nearly ready for commercial usage, and that the new track-switching stream adaptation algorithm is a proper solution to congested wireless networks that gives better QoE than the one provided with the to the date newest release of Darwin Streaming Server (6.0.3). The system has proved to work fine with different 3GPP compatible terminals.

7. Acknowledgements

This work has been supported in part by UPC Càtedra Red.es, FEDER and the Spanish Government through projects TEC2006-04504 and PROFIT Infopoints-NET. We also acknowledge the unconditional support from Futurlink.

8. References

[1] 3GPP TS 26.234, “Transparent End-To-End Packet-Switched Streaming Ser-vice (PSS): Protocols and Codecs (Release 6)”.

[2] Futurlink: www.futurlink.com.

[3] X. Sanchez-Loro, J. Casademont, J. L. Ferrer, J. Paradells. “A Proxy-based Solution for Device Capabilities Detection”. Internet and Multimedia Systems and Applications – EuroIMSA 2007. Chamonix (France) 2007.

[4] L. Cicco, S. Mascolo, V. Palmisano. “An Experimental Investigation of the End-to-End QoS of the Apple Darwin Streaming Server. ”. Int. Conf. on Wired/Wireless communications, WWIC 2008. Tampere (Finland), 2008.

[5] P. Frojdh, U. Horn, M. Kampmann, A. Nohlgren, M. Westerlund, "Adaptive streaming within the 3GPP packet-switched streaming service," IEEE Network, vol.20, no.2, pp.34-40, 2006.

[6] O. Verscheure, P. Frossard and M. Hamdi, “MPEG-2 video Services over Packet Networks: Joint Effect of Encoding Rate and Data Loss on User-Oriented QoS”. 8th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV’ 98. Cambridge, (United Kingdom), 1998.

Anexo II: Relación de archivos y funciones creadas o modificadas

- APIModules:
    - QTSSFileModule/QTSSFileModule.cpp:
        - DoDescribe: MyGetSDPFile.
        - DoSetup.
        - DoPlay.
        - SendPackets.
    - SelectorDB / SelectorDB.cpp:
        - Módulo detector.

- ApiStubLib:
    - QTSS_Private.cpp:
        - SendStandardRTSPResponse.
    - QTSS.h:
        - SendStandardRTSPResponse.

- QTFileLib:
    - QTRTPFile.cpp/h:
        - MyGetSDPFile en.cpp.
        - getActiveTracks.
        - minBWTrack.
        - maxBWTrack.
        - ActivateTrackID.
        - DeActivateTrackID.
        - IsActiveTrackID.
        - SwapTracks.
        - SetTrackQualityLevel.

- RTPStream3GPP.cpp:
        - UpdateTimeAndQuality.

- Server.tproj:

- QTSSCallbacks.cpp/h:
  - QTSS_SendStandardRTSPResponse.
- QTSSServer.cpp:
  - LoadCompiledInModules.
- RTPSession.cpp/h:
  - SendPlayResponse.

Anexo III: Principales átomos de las ISO Boxes

Code	Abstract
ID32	ID3 version 2 container
albm	Album title and track number (user-data)
auth	Media author name (user-data)
bpcc	Bits per component
bxml	binary XML container
ccid	OMA DRM Content ID
cdef	type and ordering of the components within the codestream
clsf	Media classification (user-data)
cmap	mapping between a palette and codestream components
co64	64-bit chunk Offset
colr	specifies the colourspace of the image
cprt	copyright etc. (user-data)
crhd	reserved for ClockReferenceStream header
ctts	(composition) time to sample
cvru	OMA DRM Cover URI
dcfD	Marlin DCF Duration, user-data atom type
dinf	data information box, container
dref	data reference box, declares source(s) of media data in track
dscp	Media description (user-data)
edts	edit list container
elst	an edit list
free	free space
frma	original format box
ftyp	file type and compatibility
gnre	Media genre (user-data)
grpi	OMA DRM Group ID
hdlr	handler, declares the media (handler) type
hmhd	hint media header, overall information (hint track only)
icnu	OMA DRM Icon URI
ihdr	Image Header
iinf	item information
iloc	item location
imif	IPMP Information box
infu	OMA DRM Info URL
iods	Object Descriptor container box
iphd	reserved for IPMP Stream header
ipmc	IPMP Control Box
ipro	item protection
jp2c	JPEG 2000 contiguous codestream
jp2h	Header
jp2i	intellectual property information

kywd	Media keywords (user-data)
loci	Media location information (user-data)
lrcu	OMA DRM Lyrics URI
m7hd	reserved for MPEG7Stream header
mdat	media data container
mdhd	media header, overall information about the media
mdia	container for the media information in a track
mdri	Mutable DRM information
mehd	movie extends header box
meta	Metadata container
mfhd	movie fragment header
mfra	Movie fragment random access
mfro	Movie fragment random access offset
minf	media information container
mjhd	reserved for MPEG-J Stream header
moof	movie fragment
moov	container for all the meta-data
mvex	movie extends box
mvhd	movie header, overall declarations
nmhd	Null media header, overall information (some tracks only)
ochd	reserved for ObjectContentInfoStream header
odaf	OMA DRM Access Unit Format
odda	OMA DRM Content Object
odhd	reserved for ObjectDescriptorStream header
odhe	OMA DRM Discrete Media Headers
odrb	OMA DRM Rights Object
odrm	OMA DRM Container
odtt	OMA DRM Transaction Tracking
ohdr	OMA DRM Common Readers
padb	sample padding bits
pclr	palette which maps a single component in index space to a multiple- component image
pdin	Progressive download information
perf	Media performer name (user-data)
pitm	primary item referente
resc	grid resolution at which the image was captured
resd	default grid resolution at which the image should be displayed
rtng	Media rating (user-data)
sbgp	Sample to Group box
schi	scheme information box
schm	scheme type box
sdep	Sample dependency
sdhd	reserved for SceneDescriptionStream header
sdtp	Independent and Disposable Samples Box
sdvp	SD Profile Box
sgpd	Sample group definition box
sinf	protection scheme information box

skip	free space
smhd	sound media header, overall information (sound track only)
stbl	sample table box, container for the time/space map
stco	chunk offset, partial data-offset information
stdp	sample degradation priority
stsc	sample-to-chunk, partial data-offset information
stsd	sample descriptions (codec types, initialization etc.)
stsh	shadow sync sample table
stss	sync sample table (random access points)
stsz	sample sizes (framing)
stts	(decoding) time-to-sample
stz2	compact sample sizes (framing)
subs	Sub-sample information
tfhd	track fragment header
tfra	Track fragment radom access
titl	Media title (user-data)
tkhd	track header, overall information about the track
traf	track fragment
trak	container for an individual track or stream
tref	track reference container
trex	track extends defaults
trun	track fragment run
tsel	Track selection (user-data)
udta	user-data
uinf	a tool by which a vendor may provide access to additional information associated with a UUID
ulst	a list of UUID's
uuid	user-extension box
vmhd	video media header, overall information (video track only)
yrrc	Year when media was recorded (user-data)

Anexo IV: Ejemplo de archivo de configuración

```
<?xml version ="1.0"?>
<!-- The Document Type Definition (DTD) for the file -->
<!DOCTYPE CONFIGURATION [
<!ELEMENT CONFIGURATION (SERVER, MODULE*)>
<!ELEMENT SERVER (PREF|LIST-PREF|OBJECT|LIST-OBJECT)*>
<!ELEMENT MODULE (PREF|LIST-PREF|OBJECT|LIST-OBJECT)*>
<!ATTLIST MODULE
      NAME CDATA #REQUIRED>
<!ELEMENT PREF (#PCDATA)>
<!ATTLIST PREF
      NAME CDATA #REQUIRED
      TYPE (UInt8|SInt8|UInt16|SInt16|UInt32|SInt32|UInt64|SInt64|Float32|Float64|Bool16|Bool8|char)
"char">
<!ELEMENT LIST-PREF (VALUE*)>
<!ELEMENT VALUE (#PCDATA)>
<!ATTLIST LIST-PREF
      NAME CDATA #REQUIRED
      TYPE
(UInt8|SInt8|UInt16|SInt16|UInt32|SInt32|UInt64|SInt64|Float32|Float64|Bool16|Bool8|char) "char">
<!ELEMENT OBJECT (PREF|LIST-PREF|OBJECT|LIST-OBJECT)*>
<!ATTLIST OBJECT
      NAME CDATA #REQUIRED>
<!ELEMENT LIST-OBJECT (OBJECT-VALUE*)>
<!ELEMENT OBJECT-VALUE (PREF|LIST-PREF|OBJECT|LIST-OBJECT)*>
<!ATTLIST LIST-OBJECT
      NAME CDATA #REQUIRED>
]>
<CONFIGURATION>
      <SERVER>
            <PREF NAME="rtsp_timeout" TYPE="UInt32" >0</PREF>
            <PREF NAME="real_rtsp_timeout" TYPE="UInt32" >180</PREF>
            <PREF NAME="rtp_timeout" TYPE="UInt32" >120</PREF>
            <PREF NAME="maximum_connections" TYPE="SInt32" >1000</PREF>
            <PREF NAME="maximum_bandwidth" TYPE="SInt32" >102400</PREF>
            <PREF NAME="movie_folder" >/usr/local/movies/</PREF>
            <PREF NAME="bind_ip_addr" >0</PREF>
            <PREF NAME="break_on_assert" TYPE="Bool16" >false</PREF>
            <PREF NAME="auto_restart" TYPE="Bool16" >true</PREF>
            <PREF NAME="total_bytes_update" TYPE="UInt32" >1</PREF>
            <PREF NAME="average_bandwidth_update" TYPE="UInt32" >60</PREF>
            <PREF NAME="safe_play_duration" TYPE="UInt32" >600</PREF>
            <PREF NAME="module_folder" >/usr/local/sbin/StreamingServerModules/</PREF>
            <PREF NAME="error_logfile_name" >Error</PREF>
            <PREF NAME="error_logfile_dir" >/var/streaming/logs/</PREF>
            <PREF NAME="error_logfile_interval" TYPE="UInt32" >7</PREF>
            <PREF NAME="error_logfile_size" TYPE="UInt32" >256000</PREF>
            <PREF NAME="error_logfile_verbosity" TYPE="UInt32" >2</PREF>
            <PREF NAME="screen_logging" TYPE="Bool16" >true</PREF>
            <PREF NAME="error_logging" TYPE="Bool16" >true</PREF>
            <PREF NAME="drop_all_video_delay" TYPE="SInt32" >1750</PREF>
            <PREF NAME="start_thinning_delay" TYPE="SInt32" >0</PREF>
            <PREF NAME="large_window_size" TYPE="UInt32" >64</PREF>
            <PREF NAME="window_size_threshold" TYPE="UInt32" >200</PREF>
            <PREF NAME="min_tcp_buffer_size" TYPE="UInt32" >8192</PREF>
            <PREF NAME="max_tcp_buffer_size" TYPE="UInt32" >200000</PREF>
            <PREF NAME="tcp_seconds_to_buffer" TYPE="Float32" >.5</PREF>
            <PREF NAME="do_report_http_connection_ip_address" TYPE="Bool16" >false</PREF>
            <PREF NAME="default_authorization_realm" >Streaming Server</PREF>
            <PREF NAME="run_user_name" >qtss</PREF>
            <PREF NAME="run_group_name" >qtss</PREF>
            <PREF NAME="append_source_addr_in_transport" TYPE="Bool16" >false</PREF>
            <LIST-PREF NAME="rtsp_port" TYPE="UInt16" >
                  <VALUE>554</VALUE>
                  <VALUE>7070</VALUE>
                  <VALUE>8000</VALUE>
                  <VALUE>8001</VALUE>
            </LIST-PREF>
            <PREF NAME="max_retransmit_delay" TYPE="UInt32" >500</PREF>
            <PREF NAME="small_window_size" TYPE="UInt32" >24</PREF>
            <PREF NAME="ack_logging_enabled" TYPE="Bool16" >false</PREF>
            <PREF NAME="rtcp_poll_interval" TYPE="UInt32" >100</PREF>
            <PREF NAME="rtcp_rcv_buf_size" TYPE="UInt32" >768</PREF>
```

```
            <PREF NAME="send_interval" TYPE="UInt32" >50</PREF>
            <PREF NAME="thick_all_the_way_delay" TYPE="SInt32" >-2000</PREF>
            <PREF NAME="alt_transport_src_ipaddr" ></PREF>
            <PREF NAME="max_send_ahead_time" TYPE="UInt32" >25</PREF>
            <PREF NAME="reliable_udp_slow_start" TYPE="Bool16" >true</PREF>
            <PREF NAME="auto_delete_sdp_files" TYPE="Bool16" >false</PREF>
            <PREF NAME="authentication_scheme" >digest</PREF>
            <PREF NAME="sdp_file_delete_interval_seconds" TYPE="UInt32" >10</PREF>
            <PREF NAME="auto_start" TYPE="Bool16" >false</PREF>
            <PREF NAME="reliable_udp" TYPE="Bool16" >true</PREF>
            <PREF NAME="reliable_udp_dirs" >/</PREF>
            <PREF NAME="reliable_udp_printfs" TYPE="Bool16" >false</PREF>
            <PREF NAME="drop_all_packets_delay" TYPE="SInt32" >2500</PREF>
            <PREF NAME="thin_all_the_way_delay" TYPE="SInt32" >1500</PREF>
            <PREF NAME="always_thin_delay" TYPE="SInt32" >750</PREF>
            <PREF NAME="start_thicking_delay" TYPE="SInt32" >250</PREF>
            <PREF NAME="quality_check_interval" TYPE="UInt32" >1000</PREF>
            <PREF NAME="RTSP_error_message" TYPE="Bool16" >false</PREF>
            <PREF NAME="RTSP_debug_printfs" TYPE="Bool16" >false</PREF>
            <PREF NAME="enable_monitor_stats_file" TYPE="Bool16" >false</PREF>
            <PREF NAME="monitor_stats_file_interval_seconds" TYPE="UInt32" >10</PREF>
            <PREF NAME="monitor_stats_file_name" >server_status</PREF>
            <PREF NAME="enable_packet_header_printfs" TYPE="Bool16" >false</PREF>
            <PREF NAME="packet_header_printf_options" >rtp;rr;sr;app;ack;</PREF>
            <PREF NAME="overbuffer_rate" TYPE="Float32" >2.0</PREF>
            <PREF NAME="medium_window_size" TYPE="UInt32" >48</PREF>
            <PREF NAME="window_size_max_threshold" TYPE="UInt32" >1000</PREF>
            <PREF NAME="RTSP_server_info" TYPE="Bool16" >true</PREF>
            <PREF NAME="run_num_threads" TYPE="UInt32" >0</PREF>
            <PREF NAME="pid_file" >/var/run/DarwinStreamingServer.pid</PREF>
            <PREF NAME="force_logs_close_on_write" TYPE="Bool16" >false</PREF>
            <PREF NAME="disable_thinning" TYPE="Bool16" >false</PREF>
            <LIST-PREF NAME="player_requires_rtp_header_info" >
                  <VALUE>Nokia</VALUE>
                  <VALUE>Real</VALUE>
            </LIST-PREF>
            <LIST-PREF NAME="player_requires_bandwidth_adjustment" >
                  <VALUE>Nokia</VALUE>
                  <VALUE>Real</VALUE>
            </LIST-PREF>
            <LIST-PREF NAME="player_requires_no_pause_time_adjustment" >
                  <VALUE>Nokia</VALUE>
                  <VALUE>Real</VALUE>
                  <VALUE>PVPlayer</VALUE>
            </LIST-PREF>
            <PREF NAME="enable_3gpp_protocol" TYPE="Bool16" >true</PREF>
            <PREF NAME="enable_3gpp_protocol_rate_adaptation" TYPE="Bool16" >true</PREF>
            <PREF NAME="3gpp_protocol_rate_adaptation_report_frequency" TYPE="UInt16" >1</PREF>
            <PREF NAME="default_stream_quality" TYPE="UInt16" >0</PREF>
            <PREF NAME="player_requires_rtp_start_time_adjust" >Real</PREF>
            <PREF NAME="enable_3gpp_debug_printfs" TYPE="Bool16" >false</PREF>
            <PREF NAME="enable_udp_monitor_stream" TYPE="Bool16" >false</PREF>
            <PREF NAME="udp_monitor_video_port" TYPE="UInt16" >5002</PREF>
            <PREF NAME="udp_monitor_audio_port" TYPE="UInt16" >5004</PREF>
            <PREF NAME="udp_monitor_dest_ip" >127.0.0.1</PREF>
            <PREF NAME="udp_monitor_src_ip" >0.0.0.0</PREF>
            <PREF NAME="enable_allow_guest_default" TYPE="Bool16" >true</PREF>
            <PREF NAME="run_num_rtsp_threads" TYPE="UInt32" >1</PREF>
            <PREF NAME="player_requires_disable_3gpp_rate_adapt" ></PREF>
            <PREF NAME="player_requires_3gpp_target_time" ></PREF>
            <PREF NAME="3gpp_target_time_milliseconds" TYPE="UInt32" >3000</PREF>
            <PREF NAME="player_requires_disable_thinning" ></PREF>
      </SERVER>
      <MODULE NAME="QTSSErrorLogModule" ></MODULE>
      <MODULE NAME="QTSSHomeDirectoryModule" >
            <PREF NAME="enabled" TYPE="Bool16" >false</PREF>
            <PREF NAME="movies_directory" >/Sites/Streaming/</PREF>
            <PREF NAME="max_num_conns_per_home_directory" TYPE="UInt32" >0</PREF>
            <PREF NAME="max_bandwidth_kbps_per_home_directory" TYPE="UInt32" >0</PREF>
      </MODULE>
      <MODULE NAME="QTSSRefMovieModule" >
            <PREF NAME="refmovie_xfer_enabled" TYPE="Bool16" >true</PREF>
            <PREF NAME="refmovie_rtsp_port" TYPE="UInt16" >0</PREF>
      </MODULE>
      <MODULE NAME="Selector" >
            <PREF NAME="selector_enabled" TYPE="Bool16" >false</PREF>
            <PREF NAME="hi_res" >352x240</PREF>
            <PREF NAME="ConnString" >hostaddr=147.83.39.46 dbname=profilev2 user=root
password=futurlink</PREF>
```

```
</MODULE>
<MODULE NAME="QTSSFileModule" >
      <PREF NAME="flow_control_probe_interval" TYPE="UInt32" >10</PREF>
      <PREF NAME="max_allowed_speed" TYPE="Float32" >4.000000</PREF>
      <PREF NAME="enable_shared_file_buffers" TYPE="Bool16" >true</PREF>
      <PREF NAME="enable_private_file_buffers" TYPE="Bool16" >false</PREF>
      <PREF NAME="num_shared_buffer_increase_per_session" TYPE="UInt32" >8</PREF>
      <PREF NAME="shared_buffer_unit_k_size" TYPE="UInt32" >256</PREF>
      <PREF NAME="private_buffer_unit_k_size" TYPE="UInt32" >256</PREF>
      <PREF NAME="num_shared_buffer_units_per_buffer" TYPE="UInt32" >1</PREF>
      <PREF NAME="num_private_buffer_units_per_buffer" TYPE="UInt32" >1</PREF>
      <PREF NAME="max_shared_buffer_units_per_buffer" TYPE="UInt32" >8</PREF>
      <PREF NAME="max_private_buffer_units_per_buffer" TYPE="UInt32" >8</PREF>
      <PREF NAME="add_seconds_to_client_buffer_delay" TYPE="Float32" >0.000000</PREF>
      <PREF NAME="record_movie_file_sdp" TYPE="Bool16" >false</PREF>
      <PREF NAME="enable_movie_file_sdp" TYPE="Bool16" >false</PREF>
      <PREF NAME="enable_player_compatibility" TYPE="Bool16" >true</PREF>
      <PREF NAME="compatibility_adjust_sdp_media_bandwidth_percent" TYPE="UInt32" >50</PREF>
      <PREF NAME="compatibility_adjust_rtp_start_time_milli" TYPE="SInt64" >500</PREF>
      <PREF NAME="allow_invalid_hint_track_refs" TYPE="Bool16" >false</PREF>
      <PREF NAME="max_qlevels" TYPE="UInt32" >50</PREF>
      <PREF NAME="sdp_url" ></PREF>
      <PREF NAME="admin_email" ></PREF>
</MODULE>
<MODULE NAME="QTSSReflectorModule" >
      <PREF NAME="reflector_bucket_offset_delay_msec" TYPE="UInt32" >73</PREF>
      <PREF NAME="reflector_buffer_size_sec" TYPE="UInt32" >10</PREF>
      <PREF NAME="reflector_use_in_packet_receive_time" TYPE="Bool16" >false</PREF>
      <PREF NAME="reflector_in_packet_max_receive_sec" TYPE="UInt32" >60</PREF>
      <PREF NAME="reflector_rtp_info_offset_msec" TYPE="UInt32" >500</PREF>
      <PREF NAME="disable_rtp_play_info" TYPE="Bool16" >false</PREF>
      <PREF NAME="allow_non_sdp_urls" TYPE="Bool16" >true</PREF>
      <PREF NAME="enable_broadcast_announce" TYPE="Bool16" >true</PREF>
      <PREF NAME="enable_broadcast_push" TYPE="Bool16" >true</PREF>
      <PREF NAME="max_broadcast_announce_duration_secs" TYPE="UInt32" >0</PREF>
      <PREF NAME="allow_duplicate_broadcasts" TYPE="Bool16" >false</PREF>
      <PREF NAME="enforce_static_sdp_port_range" TYPE="Bool16" >false</PREF>
      <PREF NAME="minimum_static_sdp_port" TYPE="UInt16" >20000</PREF>
      <PREF NAME="maximum_static_sdp_port" TYPE="UInt16" >65535</PREF>
      <PREF NAME="kill_clients_when_broadcast_stops" TYPE="Bool16" >false</PREF>
      <PREF NAME="use_one_SSRC_per_stream" TYPE="Bool16" >true</PREF>
      <PREF NAME="timeout_stream_SSRC_secs" TYPE="UInt32" >30</PREF>
      <PREF NAME="timeout_broadcaster_session_secs" TYPE="UInt32" >20</PREF>
      <PREF NAME="authenticate_local_broadcast" TYPE="Bool16" >false</PREF>
      <PREF NAME="disable_overbuffering" TYPE="Bool16" >false</PREF>
      <PREF NAME="allow_broadcasts" TYPE="Bool16" >true</PREF>
      <PREF NAME="allow_announced_kill" TYPE="Bool16" >true</PREF>
      <PREF NAME="enable_play_response_range_header" TYPE="Bool16" >true</PREF>
      <PREF NAME="enable_player_compatibility" TYPE="Bool16" >true</PREF>
      <PREF NAME="compatibility_adjust_sdp_media_bandwidth_percent" TYPE="UInt32" >100</PREF>
      <PREF NAME="force_rtp_info_sequence_and_time" TYPE="Bool16" >false</PREF>
      <PREF NAME="BroadcasterGroup" >broadcaster</PREF>
      <PREF NAME="redirect_broadcast_keyword" ></PREF>
      <PREF NAME="redirect_broadcasts_dir" ></PREF>
      <PREF NAME="broadcast_dir_list" ></PREF>
      <PREF NAME="ip_allow_list" >127.0.0.*</PREF>
</MODULE>
<MODULE NAME="QTSSRelayModule" >
      <PREF NAME="relay_prefs_file" >/etc/streaming/relayconfig.xml</PREF>
      <PREF NAME="relay_stats_url" ></PREF>
</MODULE>
<MODULE NAME="QTSSAccessLogModule" >
      <PREF NAME="request_logging" TYPE="Bool16" >true</PREF>
      <PREF NAME="request_logfile_size" TYPE="UInt32" >10240000</PREF>
      <PREF NAME="request_logfile_interval" TYPE="UInt32" >7</PREF>
      <PREF NAME="request_logtime_in_gmt" TYPE="Bool16" >true</PREF>
      <PREF NAME="request_logfile_dir" >/var/streaming/logs/</PREF>
      <PREF NAME="request_logfile_name" >StreamingServer</PREF>
</MODULE>
<MODULE NAME="QTSSFlowControlModule" >
      <PREF NAME="loss_thin_tolerance" TYPE="UInt32" >30</PREF>
      <PREF NAME="num_losses_to_thin" TYPE="UInt32" >3</PREF>
      <PREF NAME="loss_thick_tolerance" TYPE="UInt32" >5</PREF>
      <PREF NAME="num_losses_to_thick" TYPE="UInt32" >6</PREF>
      <PREF NAME="num_worses_to_thin" TYPE="UInt32" >2</PREF>
      <PREF NAME="flow_control_udp_thinning_module_enabled" TYPE="Bool16" >true</PREF>
</MODULE>
<MODULE NAME="QTSSPosixFileSysModule" ></MODULE>
<MODULE NAME="QTSSAdminModule" >
```

```
            <PREF NAME="IPAccessList" >127.0.0.*</PREF>
            <PREF NAME="Authenticate" TYPE="Bool16" >true</PREF>
            <PREF NAME="LocalAccessOnly" TYPE="Bool16" >true</PREF>
            <PREF NAME="RequestTimeIntervalMilli" TYPE="UInt32" >50</PREF>
            <PREF NAME="enable_remote_admin" TYPE="Bool16" >true</PREF>
            <PREF NAME="AdministratorGroup" >admin</PREF>
      </MODULE>
      <MODULE NAME="QTSSMP3StreamingModule" >
            <PREF NAME="mp3_request_logfile_name" >mp3_access</PREF>
            <PREF NAME="mp3_request_logfile_dir" >/var/streaming/logs/</PREF>
            <PREF NAME="mp3_streaming_enabled" TYPE="Bool16" >true</PREF>
            <PREF NAME="mp3_broadcast_password" > </PREF>
            <PREF NAME="mp3_broadcast_buffer_size" TYPE="UInt32" >8192</PREF>
            <PREF NAME="mp3_max_flow_control_time" TYPE="SInt32" >10000</PREF>
            <PREF NAME="mp3_request_logging" TYPE="Bool16" >true</PREF>
            <PREF NAME="mp3_request_logfile_size" TYPE="UInt32" >10240000</PREF>
            <PREF NAME="mp3_request_logfile_interval" TYPE="UInt32" >7</PREF>
            <PREF NAME="mp3_request_logtime_in_gmt" TYPE="Bool16" >true</PREF>
      </MODULE>
      <MODULE NAME="QTSSAccessModule" >
            <PREF NAME="modAccess_enabled" TYPE="Bool16" >true</PREF>
            <PREF NAME="modAccess_usersfilepath" >/etc/streaming/qtusers</PREF>
            <PREF NAME="modAccess_groupsfilepath" >/etc/streaming/qtgroups</PREF>
            <PREF NAME="modAccess_qtaccessfilename" >qtaccess</PREF>
      </MODULE>
</CONFIGURATION>
```

Anexo V: Makefile

```
# Copyright (c) 1999 Apple Computer, Inc.  All rights reserved.

NAME = DarwinStreamingServer
C++ = $(CPLUS)
CC = $(CCOMP)
LINK = $(LINKER)
CCFLAGS += $(COMPILER_FLAGS)  -DDSS_USE_API_CALLBACKS -g -Wall -Wno-format-y2k $(INCLUDE_FLAG)
PlatformHeader.h
LIBS = $(CORE_LINK_LIBS) -lCommonUtilitiesLib -lQTFileLib -lpq -lpqxx

# OPTIMIZATION
CCFLAGS += -O3

# EACH DIRECTORY WITH HEADERS MUST BE APPENDED IN THIS MANNER TO THE CCFLAGS

CCFLAGS += -I.
CCFLAGS += -IQTFileLib
CCFLAGS += -IOSMemoryLib
CCFLAGS += -IRTSPClientLib
CCFLAGS += -IAPIModules
CCFLAGS += -IAPICommonCode
CCFLAGS += -IAPIModules/OSMemory_Modules
CCFLAGS += -IAPIModules/QTSSAccessLogModule
CCFLAGS += -IAPIModules/QTSSFileModule
CCFLAGS += -IAPIModules/QTSSFlowControlModule
CCFLAGS += -IAPIModules/QTSSReflectorModule
CCFLAGS += -IAPIModules/QTSSSvrControlModule
CCFLAGS += -IAPIModules/QTSSWebDebugModule
CCFLAGS += -IAPIModules/QTSSWebStatsModule
CCFLAGS += -IAPIModules/QTSSAuthorizeModule
CCFLAGS += -IAPIModules/QTSSPOSIXFileSysModule
CCFLAGS += -IAPIModules/QTSSAdminModule
CCFLAGS += -IAPIModules/QTSSMP3StreamingModule
CCFLAGS += -IAPIModules/QTSSRTPFileModule
CCFLAGS += -IAPIModules/QTSSAccessModule
CCFLAGS += -IAPIModules/QTSSHttpFileModule
CCFLAGS += -IQTFileTools/RTPFileGen.tproj
CCFLAGS += -IAPIStubLib
CCFLAGS += -ICommonUtilitiesLib
CCFLAGS += -IRTCPUtilitiesLib
CCFLAGS += -IHTTPUtilitiesLib
CCFLAGS += -IRTPMetaInfoLib
CCFLAGS += -IPrefsSourceLib
CCFLAGS += -IServer.tproj
CCFLAGS += -IAPIModules/SelectorDB

# EACH DIRECTORY WITH A STATIC LIBRARY MUST BE APPENDED IN THIS MANNER TO THE LINKOPTS

LINKOPTS = -LCommonUtilitiesLib
LINKOPTS += -LQTFileLib
C++FLAGS = $(CCFLAGS)

CFILES = CommonUtilitiesLib/daemon.c

CPPFILES =        Server.tproj/GenerateXMLPrefs.cpp \
                  Server.tproj/main.cpp \
                  Server.tproj/QTSSCallbacks.cpp \
                  Server.tproj/QTSSDataConverter.cpp \
                  Server.tproj/QTSSDictionary.cpp \
                  Server.tproj/QTSSErrorLogModule.cpp \
                  Server.tproj/QTSServer.cpp \
                  Server.tproj/QTSServerInterface.cpp \
                  Server.tproj/QTSServerPrefs.cpp \
                  Server.tproj/QTSSExpirationDate.cpp \
                  Server.tproj/QTSSFile.cpp \
                  Server.tproj/QTSSMessages.cpp \
                  Server.tproj/QTSSModule.cpp \
                  Server.tproj/QTSSPrefs.cpp \
                  Server.tproj/QTSSSocket.cpp \
                  Server.tproj/QTSSUserProfile.cpp \
                  Server.tproj/RTCPTask.cpp \
                  Server.tproj/RTPBandwidthTracker.cpp \
                  Server.tproj/RTPOverbufferWindow.cpp \
                  Server.tproj/RTPPacketResender.cpp \
```

```
                    Server.tproj/RTPSession3GPP.cpp \
                    Server.tproj/RTPSession.cpp \
                    Server.tproj/RTPSessionInterface.cpp \
                    Server.tproj/RTPStream3gpp.cpp \
                    Server.tproj/RTPStream.cpp \
                    Server.tproj/RTSPProtocol.cpp \
                    Server.tproj/RTSPRequest3GPP.cpp \
                    Server.tproj/RTSPRequest.cpp \
                    Server.tproj/RTSPRequestInterface.cpp \
                    Server.tproj/RTSPRequestStream.cpp \
                    Server.tproj/RTSPResponseStream.cpp \
                    Server.tproj/RTSPSession3GPP.cpp \
                    Server.tproj/RTSPSession.cpp \
                    Server.tproj/RTSPSessionInterface.cpp \
                    Server.tproj/RunServer.cpp \
                    PrefsSourceLib/FilePrefsSource.cpp \
                    PrefsSourceLib/XMLPrefsParser.cpp \
                    PrefsSourceLib/XMLParser.cpp \
                    OSMemoryLib/OSMemory.cpp \
                    RTSPClientLib/RTSPClient.cpp \
                    RTSPClientLib/ClientSocket.cpp \
                    HTTPUtilitiesLib/HTTPProtocol.cpp \
                    HTTPUtilitiesLib/HTTPRequest.cpp \
                    RTCPUtilitiesLib/RTCPAckPacket.cpp \
                    RTCPUtilitiesLib/RTCPAPPNADUPacket.cpp \
                    RTCPUtilitiesLib/RTCPAPPPacket.cpp \
                    RTCPUtilitiesLib/RTCPAPPQTSSPacket.cpp \
                    RTCPUtilitiesLib/RTCPPacket.cpp \
                    RTCPUtilitiesLib/RTCPSRPacket.cpp \
                    RTPMetaInfoLib/RTPMetaInfoPacket.cpp \
                    APIStubLib/QTSS_Private.cpp \
                    APICommonCode/QTSSModuleUtils.cpp\
                    APICommonCode/QTSSRollingLog.cpp \
                    APICommonCode/SDPSourceInfo.cpp \
                    APICommonCode/SourceInfo.cpp \
                    APICommonCode/QTAccessFile.cpp \
                    APICommonCode/QTSS3GPPModuleUtils.cpp \
                    SafeStdLib/InternalStdLib.cpp \
                    APIModules/QTSSAccessLogModule/QTSSAccessLogModule.cpp \
                    APIModules/QTSSFileModule/QTSSFileModule.cpp \
                    APIModules/QTSSFlowControlModule/QTSSFlowControlModule.cpp \
                    APIModules/QTSSReflectorModule/QTSSReflectorModule.cpp \
                    APIModules/QTSSReflectorModule/QTSSRelayModule.cpp \
                    APIModules/QTSSReflectorModule/ReflectorSession.cpp\
                    APIModules/QTSSReflectorModule/RelaySession.cpp\
                    APIModules/QTSSReflectorModule/ReflectorStream.cpp\
                    APIModules/QTSSReflectorModule/RCFSourceInfo.cpp \
                    APIModules/QTSSReflectorModule/RTSPSourceInfo.cpp \
                    APIModules/QTSSReflectorModule/RelayOutput.cpp \
                    APIModules/QTSSReflectorModule/RelaySDPSourceInfo.cpp \
                    APIModules/QTSSReflectorModule/RTPSessionOutput.cpp \
                    APIModules/QTSSReflectorModule/SequenceNumberMap.cpp \
                    APIModules/QTSSWebDebugModule/QTSSWebDebugModule.cpp \
                    APIModules/QTSSWebStatsModule/QTSSWebStatsModule.cpp \
                    APIModules/QTSSPOSIXFileSysModule/QTSSPosixFileSysModule.cpp \
                    APIModules/QTSSAdminModule/AdminElementNode.cpp \
                    APIModules/QTSSAdminModule/AdminQuery.cpp \
                    APIModules/QTSSAdminModule/QTSSAdminModule.cpp \
                    APIModules/QTSSMP3StreamingModule/QTSSMP3StreamingModule.cpp \
                    APIModules/QTSSRTPFileModule/QTSSRTPFileModule.cpp \
                    APIModules/QTSSRTPFileModule/RTPFileSession.cpp \
                    APIModules/QTSSAccessModule/QTSSAccessModule.cpp \
                    APIModules/QTSSHttpFileModule/QTSSHttpFileModule.cpp \
                    APIModules/QTSSAccessModule/AccessChecker.cpp \
                    APIModules/SelectorDB/Selector.cpp
# CCFLAGS += $(foreach dir,$(HDRS),-I$(dir))

LIBFILES =  QTFileLib/libQTFileLib.a \   CommonUtilitiesLib/libCommonUtilitiesLib.a
all: DarwinStreamingServer

DarwinStreamingServer: $(CFILES:.c=.o) $(CPPFILES:.cpp=.o)  $(LIBFILES)
      $(LINK) -o $@ $(CFILES:.c=.o) $(CPPFILES:.cpp=.o) $(COMPILER_FLAGS) $(LINKOPTS) $(LIBS)

install: DarwinStreamingServer
clean: rm -f $(CFILES:.c=.o) $(CPPFILES:.cpp=.o)

.SUFFIXES: .cpp .c .o
.cpp.o:     $(C++) -c -o $*.o $(DEFINES) $(C++FLAGS) $*.cpp
.c.o:       $(CC) -c -o $*.o $(DEFINES) $(CCFLAGS) $*.c
```

Anexo VI: Código fuente

```
//-------------------DoDescribe en QTSSFileModule.cpp-------------------//

QTSS_Error DoDescribe(QTSS_StandardRTSP_Params* inParamBlock)
{
    if (isSDP(inParamBlock))
    {
        StrPtrLen pathStr;
        (void)QTSS_LockObject(inParamBlock->inRTSPRequest);
        (void)QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0,
       (void**)&pathStr.Ptr, &pathStr.Len);
        QTSS_Error err = QTSSModuleUtils::SendErrorResponse(inParamBlock->inRTSPRequest,
       qtssClientNotFound, sNoSDPFileFoundErr, &pathStr);
        (void)QTSS_UnlockObject(inParamBlock->inRTSPRequest);
        return err;
    }

    //
    // Get the FileSession for this DESCRIBE, if any.
    UInt32 theLen = sizeof(FileSession*);
    FileSession*    theFile = NULL;
    QTSS_Error      theErr = QTSS_NoErr;
    Bool16          pathEndsWithSDP = false;
    static StrPtrLen sSDPSuffix(".sdp");
    SInt16 vectorIndex = 1;
    ResizeableStringFormatter theFullSDPBuffer(NULL,0);
    StrPtrLen bufferDelayStr;
    char tempBufferDelay[64];
    StrPtrLen theSDPData;

    (void)QTSS_GetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, (void*)&theFile,
     &theLen);
    // Generate the complete file path
    UInt32 thePathLen = 0;
    OSCharArrayDeleter thePath(QTSSModuleUtils::GetFullPath(inParamBlock->inRTSPRequest,
     qtssRTSPReqFilePath,&thePathLen, &sSDPSuffix));

    //first locate the target movie
    thePath.GetObject()[thePathLen - sSDPSuffix.Len] = '\0';
    //truncate the .sdp added in the GetFullPath call
    StrPtrLen   requestPath(thePath.GetObject(), ::strlen(thePath.GetObject()));
    if (requestPath.Len > sSDPSuffix.Len )
    {   StrPtrLen endOfPath(&requestPath.Ptr[requestPath.Len -  sSDPSuffix.Len], sSDPSuffix.Len);
        if (endOfPath.EqualIgnoreCase(sSDPSuffix)) // it is a .sdp
        {   pathEndsWithSDP = true;
        }
    }

    if ( theFile != NULL )
    {
        // There is already a file for this session. This can happen if there are multiple
       DESCRIBES, or a DESCRIBE has been issued with a Session ID, or some such thing.
        StrPtrLen   moviePath( theFile->fFile.GetMoviePath() );

       // Stop playing because the new file isn't ready yet to send packets.
       // Needs a Play request to get things going. SendPackets on the file is active if not
       paused.
       (void)QTSS_Pause(inParamBlock->inClientSession);
       (*theFile).fPaused = true;

        //
        // This describe is for a different file. Delete the old FileSession.
        if ( !requestPath.Equal( moviePath ) )
        {
            DeleteFileSession(theFile);
            theFile = NULL;

            // NULL out the attribute value, just in case.
            (void)QTSS_SetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, &theFile,
sizeof(theFile));
        }
    }

    if ( theFile == NULL )
    {
```

```
        theErr = CreateQTRTPFile(inParamBlock, thePath.GetObject(), &theFile);
        if (theErr != QTSS_NoErr)
            return theErr;

        // Store this newly created file object in the RTP session.
        theErr = QTSS_SetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, &theFile,
sizeof(theFile));
    }

    //replace the sacred character we have trodden on in order to truncate the path.
    thePath.GetObject()[thePathLen - sSDPSuffix.Len] = sSDPSuffix.Ptr[0];

    iovec theSDPVec[sNumSDPVectors];//1 for the RTSP header, 6 for the sdp header, 1 for the sdp
     body
    ::memset(&theSDPVec[0], 0, sizeof(theSDPVec));

    if (sEnableMovieFileSDP)
    {
        // Check to see if there is an sdp file, if so, return that file instead  of the built-in
     sdp. ReadEntireFile allocates memory but if all goes well theSDPData will be  managed by the
     File Session
        (void)QTSSModuleUtils::ReadEntireFile(thePath.GetObject(), &theSDPData);
    }
    OSCharArrayDeleter sdpDataDeleter(theSDPData.Ptr); // Just in case we fail we know to clean up.
     But we clear the deleter if we succeed.

    if (theSDPData.Len > 0)
    {
        SDPContainer fileSDPContainer;
        fileSDPContainer.SetSDPBuffer(&theSDPData);
        if (!fileSDPContainer.IsSDPBufferValid())
        {    return QTSSModuleUtils::SendErrorResponseWithMessage(inParamBlock->inRTSPRequest,
qtssUnsupportedMediaType, &sSDPNotValidMessage);
        }


        // Append the Last Modified header to be a good caching proxy citizen before sending the
       Describe
        (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssLastModifiedHeader,
                                    theFile->fFile.GetQTFile()->GetModDateStr(),
DateBuffer::kDateBufferLen);
        (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssCacheControlHeader,
                                    kCacheControlHeader.Ptr, kCacheControlHeader.Len);

        //Now that we have the file data, send an appropriate describe response to the client
        theSDPVec[1].iov_base = theSDPData.Ptr;
        theSDPVec[1].iov_len = theSDPData.Len;

        QTSSModuleUtils::SendDescribeResponse(inParamBlock->inRTSPRequest, inParamBlock-
>inClientSession,
                                              &theSDPVec[0], 3, theSDPData.Len);
    }
    else
    {
        // Before generating the SDP and sending it, check to see if there is an If-Modified-Since
     date. If there is, and the content hasn't been modified, then just return a 304 Not Modified
        QTSS_TimeVal* theTime = NULL;
        (void) QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqIfModSinceDate, 0,
(void**)&theTime, &theLen);
        if ((theLen == sizeof(QTSS_TimeVal)) && (*theTime > 0))
        {
            // There is an If-Modified-Since header. Check it vs. the content.
            if (*theTime == theFile->fFile.GetQTFile()->GetModDate())
            {
                theErr = QTSS_SetValue( inParamBlock->inRTSPRequest, qtssRTSPReqStatusCode, 0,
                                        &kNotModifiedStatus, sizeof(kNotModifiedStatus) );
                Assert(theErr == QTSS_NoErr);
                // Because we are using this call to generate a 304 Not Modified response, we do not
               need to pass in a RTP Stream
                theErr = QTSS_SendStandardRTSPResponse(inParamBlock->inRTSPRequest, inParamBlock-
>inClientSession, 0);
                Assert(theErr == QTSS_NoErr);
                return QTSS_NoErr;
            }
        }

        FILE* sdpFile = NULL;
        if (sRecordMovieFileSDP &&  !pathEndsWithSDP) // don't auto create sdp for an sdp file
       because it would look like a broadcast
```

```
        {
            sdpFile = ::fopen(thePath.GetObject(),"r"); // see if there already is a .sdp for the
               movie
            if (sdpFile != NULL) // one already exists don't mess with it
            {   ::fclose(sdpFile);
                sdpFile = NULL;
            }
            else
                sdpFile = ::fopen(thePath.GetObject(),"w"); // create the .sdp
        }

        UInt32 totalSDPLength = 0;

        //Get filename
         char* fileNameStr = NULL;
        (void)QTSS_GetValueAsString(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0,
&fileNameStr);
        QTSSCharArrayDeleter fileNameStrDeleter(fileNameStr);

        //Get IP addr
        StrPtrLen ipStr;
        (void)QTSS_GetValuePtr(inParamBlock->inRTSPSession, qtssRTSPSesLocalAddrStr, 0,
(void**)&ipStr.Ptr, &ipStr.Len);


//
// *** The order of sdp headers is specified and required by rfc 2327
//
// -------- version header

        theFullSDPBuffer.Put(sVersionHeader);
        theFullSDPBuffer.Put(sEOL);

// -------- owner header

        const SInt16 sLineSize = 256;
        char ownerLine[sLineSize]="";
        ownerLine[sLineSize - 1] = 0;

        char *ipCstr = ipStr.GetAsCString();
        OSCharArrayDeleter ipDeleter(ipCstr);

 // the first number is the NTP time used for the session identifier (this changes for each request)
 // the second number is the NTP date time of when the file was modified (this changes when the file
changes)
        qtss_sprintf(ownerLine, "o=StreamingServer %"_64BITARG_"d %"_64BITARG_"d IN IP4 %s",
(SInt64) OS::UnixTime_Secs() + 2208988800LU, (SInt64) theFile->fFile.GetQTFile()-
>GetModDate(),ipCstr);
        Assert(ownerLine[sLineSize - 1] == 0);

        StrPtrLen ownerStr(ownerLine);
        theFullSDPBuffer.Put(ownerStr);
        theFullSDPBuffer.Put(sEOL);

// -------- session header

        theFullSDPBuffer.Put(sSessionNameHeader);
        theFullSDPBuffer.Put(fileNameStr);
        theFullSDPBuffer.Put(sEOL);

// -------- uri header

        theFullSDPBuffer.Put(sURLHeader);
        theFullSDPBuffer.Put(sEOL);

// -------- email header

        theFullSDPBuffer.Put(sEmailHeader);
        theFullSDPBuffer.Put(sEOL);

// -------- connection information header

        theFullSDPBuffer.Put(sConnectionHeader);
        theFullSDPBuffer.Put(sEOL);

// -------- time header

      // t=0 0 is a permanent always available movie (doesn't ever change unless we change the code)
        theFullSDPBuffer.Put(sPermanentTimeHeader);
```

```
        theFullSDPBuffer.Put(sEOL);

// -------- control header

        theFullSDPBuffer.Put(sStaticControlHeader);
        theFullSDPBuffer.Put(sEOL);

// -------- add buffer delay

        if (sAddClientBufferDelaySecs > 0) // increase the client buffer delay by the preference
       amount.
        {
            Float32 bufferDelay = 3.0; // the client doesn't advertise it's default value so we
       guess.

            static StrPtrLen sBuffDelayStr("a=x-bufferdelay:");

            StrPtrLen delayStr;
            theSDPData.FindString(sBuffDelayStr, &delayStr);
            if (delayStr.Len > 0)
            {
                UInt32 offset = (delayStr.Ptr - theSDPData.Ptr) + delayStr.Len; // step past the
       string
                delayStr.Ptr = theSDPData.Ptr + offset;
                delayStr.Len = theSDPData.Len - offset;
                StringParser theBufferSecsParser(&delayStr);
                theBufferSecsParser.ConsumeWhitespace();
                bufferDelay = theBufferSecsParser.ConsumeFloat();
            }

            bufferDelay += sAddClientBufferDelaySecs;
            qtss_sprintf(tempBufferDelay, "a=x-bufferdelay:%.2f",bufferDelay);
            bufferDelayStr.Set(tempBufferDelay);
            theFullSDPBuffer.Put(bufferDelayStr);
            theFullSDPBuffer.Put(sEOL);
        }

// -------- Here we call MyGetSDPFile which includes two parameters instead of one
//       It returns the SDP data ordered by track bandwidth in pairs (audio video)
//       and includes two values, the total length and the length including only
//       the two first tracks.

     //now append content-determined sdp ( cached in QTRTPFile )
     int mysdpLen = 0, totalsdpLen = 0;
     theSDPData.Ptr = theFile->fFile.MyGetSDPFile(&mysdpLen,&totalsdpLen);
     DEBUG_PRINTF(("SDP %s\n", theSDPData.Ptr));
     theSDPData.Len = mysdpLen;

// ----------- Add the movie's sdp headers to our sdp headers

     theFullSDPBuffer.Put(theSDPData);
     StrPtrLen fullSDPBuffSPL(theFullSDPBuffer.GetBufPtr(),theFullSDPBuffer.GetBytesWritten());

     //We re-edit the length of the SDP data
     theSDPData.Len = totalsdpLen;
// ------------ Check the headers
        SDPContainer rawSDPContainer;
        rawSDPContainer.SetSDPBuffer( &fullSDPBuffSPL );
        if (!rawSDPContainer.IsSDPBufferValid())
        {    return QTSSModuleUtils::SendErrorResponseWithMessage(inParamBlock->inRTSPRequest,
qtssUnsupportedMediaType, &sSDPNotValidMessage);
        }

// ------------ reorder the sdp headers to make them proper.
        Float32 adjustMediaBandwidthPercent = 1.0;
        Bool16 adjustMediaBandwidth = false;
        if (sPlayerCompatibility )
            adjustMediaBandwidth = QTSSModuleUtils::HavePlayerProfile(sServerPrefs,
inParamBlock,QTSSModuleUtils::kAdjustBandwidth);

        if (adjustMediaBandwidth)
            adjustMediaBandwidthPercent = (Float32) sAdjustMediaBandwidthPercent / 100.0;

        ResizeableStringFormatter buffer;
        SDPContainer* insertMediaLines = QTSS3GPPModuleUtils::Get3GPPSDPFeatureListCopy(buffer);
       SDPLineSorter sortedSDP(&rawSDPContainer,adjustMediaBandwidthPercent,insertMediaLines);
       delete insertMediaLines;
       StrPtrLen *theSessionHeadersPtr = sortedSDP.GetSessionHeaders();
       StrPtrLen *theMediaHeadersPtr = sortedSDP.GetMediaHeaders();
```

```
// ----------- write out the sdp

     totalSDPLength += ::WriteSDPHeader(sdpFile, theSDPVec, &vectorIndex, theSessionHeadersPtr);
     totalSDPLength += ::WriteSDPHeader(sdpFile, theSDPVec, &vectorIndex, theMediaHeadersPtr);

// -------- done with SDP processing

        if (sdpFile !=NULL)
            ::fclose(sdpFile);

        Assert(theSDPData.Len > 0);
        Assert(theSDPVec[2].iov_base != NULL);
        //ok, we have a filled out iovec. Let's send the response!

        // Append the Last Modified header to be a good caching proxy citizen before sending the
       Describe
        (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssLastModifiedHeader,theFile-
>fFile.GetQTFile()->GetModDateStr(), DateBuffer::kDateBufferLen);
        (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssCacheControlHeader,
kCacheControlHeader.Ptr, kCacheControlHeader.Len);
        QTSSModuleUtils::SendDescribeResponse(inParamBlock->inRTSPRequest, inParamBlock-
>inClientSession,
                                                                        &theSDPVec[0], vectorIndex,
totalSDPLength);
    }

    Assert(theSDPData.Ptr != NULL);
    Assert(theSDPData.Len > 0);

   //now parse the movie media sdp data. We need to do this in order to extract payload information.
   //The SDP parser object will not take responsibility of the memory (one exception... see above)
   theFile->fSDPSource.Parse(theSDPData.Ptr, theSDPData.Len);
   sdpDataDeleter.ClearObject(); // don't delete theSDPData, theFile has it now.

    return QTSS_NoErr;
}

//-------------------DoSetup en QTSSFileModule.cpp-------------------//

QTSS_Error DoSetup(QTSS_StandardRTSP_Params* inParamBlock)
{
    UInt32     trackType=NULL;

    if (isSDP(inParamBlock))
    {
        StrPtrLen pathStr;
        (void)QTSS_LockObject(inParamBlock->inRTSPRequest);
        (void)QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0,
(void**)&pathStr.Ptr, &pathStr.Len);
        QTSS_Error err = QTSSModuleUtils::SendErrorResponse(inParamBlock->inRTSPRequest,
qtssClientNotFound, sNoSDPFileFoundErr, &pathStr);
        (void)QTSS_UnlockObject(inParamBlock->inRTSPRequest);
        return err;
    }

    //setup this track in the file object
    FileSession* theFile = NULL;
    UInt32 theLen = sizeof(FileSession*);
    QTSS_Error theErr = QTSS_GetValue(inParamBlock->inClientSession, sFileSessionAttr, 0,
(void*)&theFile, &theLen);
    if ((theErr != QTSS_NoErr) || (theLen != sizeof(FileSession*)))
    {
        char* theFullPath = NULL;
        //theErr = QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqLocalPath, 0,
(void**)&theFullPath, &theLen);
       theErr = QTSS_GetValueAsString(inParamBlock->inRTSPRequest, qtssRTSPReqLocalPath, 0,
&theFullPath);
        Assert(theErr == QTSS_NoErr);
        // This is possible, as clients are not required to send a DESCRIBE. If we haven't set
anything up yet, set everything up
        theErr = CreateQTRTPFile(inParamBlock, theFullPath, &theFile);
       QTSS_Delete(theFullPath);
        if (theErr != QTSS_NoErr)
            return theErr;

        int theSDPBodyLen = 0;
        char* theSDPData = theFile->fFile.GetSDPFile(&theSDPBodyLen);
```

```
        //now parse the sdp. We need to do this in order to extract payload information.
        //The SDP parser object will not take responsibility of the memory (one exception... see
above)
        theFile->fSDPSource.Parse(theSDPData, theSDPBodyLen);
        // Store this newly created file object in the RTP session.
        theErr = QTSS_SetValue(inParamBlock->inClientSession, sFileSessionAttr, 0, &theFile,
sizeof(theFile));
    }

    //unless there is a digit at the end of this path (representing trackID), don't even bother with
the request
    char* theDigitStr = NULL;
    (void)QTSS_GetValueAsString(inParamBlock->inRTSPRequest, qtssRTSPReqFileDigit, 0, &theDigitStr);
    QTSSCharArrayDeleter theDigitStrDeleter(theDigitStr);
     if (theDigitStr == NULL)
        return QTSSModuleUtils::SendErrorResponse(inParamBlock->inRTSPRequest,
                                                  qtssClientBadRequest,
sExpectedDigitFilenameErr);

    // Identify the requested track as original track
    UInt32 originalTrackID = ::strtol(theDigitStr, NULL, 10);

    // QTRTPFile::ErrorCode qtfileErr = theFile->fFile.AddTrack(theTrackID, false); //test for 3gpp
     monotonic wall clocktime and sequence
    DEBUG_PRINTF(("DO SETUP GetnumSTREAMS %u\n", theFile->fSDPSource.GetNumStreams()));

    // Identify the original payload type video/audio
    for (UInt32 x = 0; x < theFile->fSDPSource.GetNumStreams(); x++)
    {
        SourceInfo::StreamInfo* theStreamInfo = theFile->fSDPSource.GetStreamInfo(x);
        if (theStreamInfo->fTrackID == originalTrackID)
        {
            trackType = theStreamInfo->fPayloadType;
            break;
        }
    }

    //find the payload for this track ID (if applicable)
    StrPtrLen* thePayload = NULL;
    UInt32 thePayloadType = qtssUnknownPayloadType;
    Float32 bufferDelay = (Float32) 3.0; // FIXME need a constant defined for 3.0 value. It is used
multiple places

    //How many video/audio tracks?
    for (UInt32 x = 0; x < theFile->fSDPSource.GetNumStreams(); x++)
    {
        SourceInfo::StreamInfo* theStreamInfo = theFile->fSDPSource.GetStreamInfo(x);
     // If the payload is the same as the original track continue
        if (theStreamInfo->fPayloadType!=trackType)
     continue;
     thePayloadType = theStreamInfo->fPayloadType;
     UInt32 theTrackID = theStreamInfo->fTrackID;
     thePayload = &theStreamInfo->fPayloadName;
     bufferDelay = theStreamInfo->fBufferDelay;

     //Add tracks and deactivate non original ones
        QTRTPFile::ErrorCode qtfileErr = theFile->fFile.AddTrack(theTrackID, false);
     if (theTrackID != originalTrackID)
     qtfileErr = theFile->fFile.DeActivateTrackID(theTrackID);

    //if we get an error back, forward that error to the client

    if (qtfileErr == QTRTPFile::errTrackIDNotFound)
        return QTSSModuleUtils::SendErrorResponse(inParamBlock->inRTSPRequest,
                                                  qtssClientNotFound, sTrackDoesntExistErr);
    else if (qtfileErr != QTRTPFile::errNoError)
     return QTSSModuleUtils::SendErrorResponse(inParamBlock->inRTSPRequest,
                                                  qtssUnsupportedMediaType, sBadQTFileErr);

    // Before setting up this track, check to see if there is an If-Modified-Since date. If there
     is, and the content hasn't been modified, then just return a 304 Not Modified
    QTSS_TimeVal* theTime = NULL;
    (void) QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqIfModSinceDate, 0,
(void**)&theTime, &theLen);
    if ((theLen == sizeof(QTSS_TimeVal)) && (*theTime > 0))
    {
        // There is an If-Modified-Since header. Check it vs. the content.
        if (*theTime == theFile->fFile.GetQTFile()->GetModDate())
        {
```

```
            theErr = QTSS_SetValue( inParamBlock->inRTSPRequest, qtssRTSPReqStatusCode, 0,
                                    &kNotModifiedStatus, sizeof(kNotModifiedStatus) );
            Assert(theErr == QTSS_NoErr);
            // Because we are using this call to generate a 304 Not Modified response, we do not
               need to pass in a RTP Stream
            theErr = QTSS_SendStandardRTSPResponse(inParamBlock->inRTSPRequest, inParamBlock-
>inClientSession, 0);
            Assert(theErr == QTSS_NoErr);
            return QTSS_NoErr;
        }
    }

    //Create a new RTP stream
    QTSS_RTPStreamObject newStream = NULL;
    theErr = QTSS_AddRTPStream(inParamBlock->inClientSession, inParamBlock->inRTSPRequest,
&newStream, 0);
    if (theErr != QTSS_NoErr)
        return theErr;
    DEBUG_PRINTF(("Add Stream %u\n", theTrackID));
    // Set the payload type, payload name & timescale of this track
    SInt32 theTimescale = theFile->fFile.GetTrackTimeScale(theTrackID);
    UInt16* myVar;
    theErr = QTSS_SetValue(newStream, qtssRTPStrBufferDelayInSecs, 0, &bufferDelay,
sizeof(bufferDelay));
    Assert(theErr == QTSS_NoErr);
    theErr = QTSS_SetValue(newStream, qtssRTPStrPayloadName, 0, thePayload->Ptr, thePayload->Len);
    Assert(theErr == QTSS_NoErr);
    theErr = QTSS_SetValue(newStream, qtssRTPStrPayloadType, 0, &thePayloadType,
sizeof(thePayloadType));
    Assert(theErr == QTSS_NoErr);
    theErr = QTSS_SetValue(newStream, qtssRTPStrTimescale, 0, &theTimescale, sizeof(theTimescale));
    Assert(theErr == QTSS_NoErr);
    theErr = QTSS_SetValue(newStream, qtssRTPStrTrackID, 0, &theTrackID, sizeof(theTrackID));
    Assert(theErr == QTSS_NoErr);
    theErr = QTSS_GetValuePtr(newStream, qtssRTPStrSvrRTPPort, 0, (void**)&myVar, &theLen);

    // Now the number of quality levels comes as an external parameter.
    theErr = QTSS_SetValue(newStream, qtssRTPStrNumQualityLevels, 0, &sNumQualityLevels,
sizeof(sNumQualityLevels));
    Assert(theErr == QTSS_NoErr);

    // Get the SSRC of this track
    UInt32* theTrackSSRC = NULL;
    UInt32 theTrackSSRCSize = 0;

    (void)QTSS_GetValuePtr(newStream, qtssRTPStrSSRC, 0, (void**)&theTrackSSRC, &theTrackSSRCSize);

    // The RTP stream should ALWAYS have an SSRC assuming QTSS_AddStream succeeded.
    Assert((theTrackSSRC != NULL) && (theTrackSSRCSize == sizeof(UInt32)));

    //give the file some info it needs.

    theFile->fFile.SetTrackSSRC(theTrackID, *theTrackSSRC);
    theFile->fFile.SetTrackCookies(theTrackID, newStream, thePayloadType);

    StrPtrLen theHeader;
    theErr = QTSS_GetValuePtr(inParamBlock->inRTSPHeaders, qtssXRTPMetaInfoHeader, 0,
(void**)&theHeader.Ptr, &theHeader.Len);
    if (theErr == QTSS_NoErr)
    {
        // If there is an x-RTP-Meta-Info header in the request, mirror that header in the response.
       We will support any fields supported by the QTFileLib.
        RTPMetaInfoPacket::FieldID* theFields = NEW
        RTPMetaInfoPacket::FieldID[RTPMetaInfoPacket::kNumFields];
        ::memcpy(theFields, QTRTPFile::GetSupportedRTPMetaInfoFields(),
sizeof(RTPMetaInfoPacket::FieldID) * RTPMetaInfoPacket::kNumFields);

        // This function does the work of appending the response header based on the fields we
       support and the requested fields.
        theErr = QTSSModuleUtils::AppendRTPMetaInfoHeader(inParamBlock->inRTSPRequest, &theHeader,
theFields);

        // This returns QTSS_NoErr only if there are some valid, useful fields
        Bool16 isVideo = false;
        if (thePayloadType == qtssVideoPayloadType)
            isVideo = true;
        if (theErr == QTSS_NoErr)
            theFile->fFile.SetTrackRTPMetaInfo(theTrackID, theFields, isVideo);
    }
```

```
   // Our array has now been updated to reflect the fields requested by the client send the setup
response
     if (theTrackID==originalTrackID)
     {
     (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssLastModifiedHeader,
                                 theFile->fFile.GetQTFile()->GetModDateStr(),
DateBuffer::kDateBufferLen);
     (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssCacheControlHeader,
                                 kCacheControlHeader.Ptr, kCacheControlHeader.Len);
     theErr = QTSS_SendStandardRTSPResponse(inParamBlock->inRTSPRequest, newStream, 0);
     Assert(theErr == QTSS_NoErr);
     }
    }
    return QTSS_NoErr;
}

//-------------------DoPlay en QTSSFileModule.cpp-------------------//

QTSS_Error DoPlay(QTSS_StandardRTSP_Params* inParamBlock)
{
    QTRTPFile::ErrorCode qtFileErr = QTRTPFile::errNoError;

    if (isSDP(inParamBlock))
    {
        StrPtrLen pathStr;
        (void)QTSS_LockObject(inParamBlock->inRTSPRequest);
        (void)QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0,
(void**)&pathStr.Ptr, &pathStr.Len);
        QTSS_Error err = QTSSModuleUtils::SendErrorResponse(inParamBlock->inRTSPRequest,
qtssClientNotFound, sNoSDPFileFoundErr, &pathStr);
        (void)QTSS_UnlockObject(inParamBlock->inRTSPRequest);
        return err;
    }

    FileSession** theFile = NULL;
    UInt32 theLen = 0;
    QTSS_Error theErr = QTSS_GetValuePtr(inParamBlock->inClientSession, sFileSessionAttr, 0,
(void**)&theFile, &theLen);
    if ((theErr != QTSS_NoErr) || (theLen != sizeof(FileSession*)))
        return QTSS_RequestFailed;

    theErr = SetupCacheBuffers(inParamBlock, theFile);
    if (theErr != QTSS_NoErr)
        return theErr;

    //make sure to clear the next packet the server would have sent!
    (*theFile)->fPacketStruct.packetData = NULL;

    // Set the default quality before playing.
    QTRTPFile::RTPTrackListEntry* thePacketTrack;
    for (UInt32 x = 0; x < (*theFile)->fSDPSource.GetNumStreams(); x++)
    {
         SourceInfo::StreamInfo* theStreamInfo = (*theFile)->fSDPSource.GetStreamInfo(x);
         if (!(*theFile)->fFile.FindTrackEntry(theStreamInfo->fTrackID,&thePacketTrack))
            break;
         (*theFile)->fFile.SetTrackQualityLevel(thePacketTrack, sDefaultStreamingQuality, 0);
    }

    // How much are we going to tell the client to back up?
    Float32 theBackupTime = 0;

    char* thePacketRangeHeader = NULL;
    theErr = QTSS_GetValuePtr(inParamBlock->inRTSPHeaders, qtssXPacketRangeHeader, 0,
(void**)&thePacketRangeHeader, &theLen);
    if (theErr == QTSS_NoErr)
    {
        StrPtrLen theRangeHdrPtr(thePacketRangeHeader, theLen);
        StringParser theRangeParser(&theRangeHdrPtr);

        theRangeParser.ConsumeUntil(NULL, StringParser::sDigitMask);
        UInt64 theStartPN = theRangeParser.ConsumeInteger();

        theRangeParser.ConsumeUntil(NULL, StringParser::sDigitMask);
        (*theFile)->fStopPN = theRangeParser.ConsumeInteger();

        theRangeParser.ConsumeUntil(NULL, StringParser::sDigitMask);
        (*theFile)->fStopTrackID = theRangeParser.ConsumeInteger();
```

```
        qtFileErr = (*theFile)->fFile.SeekToPacketNumber((*theFile)->fStopTrackID, theStartPN);
        (*theFile)->fStartTime = (*theFile)->fFile.GetRequestedSeekTime();
    }
    else
    {
        Float64* theStartTimeP = NULL;
        Float64 currentTime = 0;
        theErr = QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqStartTime, 0,
(void**)&theStartTimeP, &theLen);
        if ((theErr != QTSS_NoErr) || (theLen != sizeof(Float64)))
        {   // No start time so just start at the last packet ready to send
            // This packet could be somewhere out in the middle of the file.
             currentTime =  (*theFile)->fFile.GetFirstPacketTransmitTime();
             theStartTimeP = &currentTime;
             (*theFile)->fStartTime = currentTime;
        }

        Float32* theMaxBackupTime = NULL;
        theErr = QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqPrebufferMaxTime, 0,
(void**)&theMaxBackupTime, &theLen);
        Assert(theMaxBackupTime != NULL);

        if (*theMaxBackupTime == -1)
        {
            //
            // If this is an old client (doesn't send the x-prebuffer header) or an mp4 client,
            // - don't back up to a key frame, and do not adjust the buffer time
         qtFileErr = (*theFile)->fFile.Seek(*theStartTimeP, 0);
            (*theFile)->fStartTime = *theStartTimeP;

            //
            // burst out -transmit time packets
            (*theFile)->fAllowNegativeTTs = false;
        }
        else
        {
            qtFileErr = (*theFile)->fFile.Seek(*theStartTimeP, *theMaxBackupTime);
            Float64 theFirstPacketTransmitTime = (*theFile)->fFile.GetFirstPacketTransmitTime();
            theBackupTime = (Float32) ( *theStartTimeP - theFirstPacketTransmitTime);

            //
            // For oddly authored movies, there are situations in which the packet
            // transmit time can be before the sample time. In that case, the backup
            // time may exceed the max backup time. In that case, just make the backup
            // time the max backup time.
            if (theBackupTime > *theMaxBackupTime)
                theBackupTime = *theMaxBackupTime;
            //
            // If client specifies that it can do extra buffering (new client), use the first
            // packet transmit time as the start time for this play burst. We don't need to
            // burst any packets because the client can do the extra buffering
               Bool16* overBufferEnabledPtr = NULL;
               theLen = 0;
               theErr = QTSS_GetValuePtr(inParamBlock->inClientSession, qtssCliSesOverBufferEnabled,
0, (void**)&overBufferEnabledPtr, &theLen);
               if ((theErr == QTSS_NoErr) && (theLen == sizeof(Bool16)) && *overBufferEnabledPtr)
                       (*theFile)->fStartTime = *theStartTimeP;
               else
                (*theFile)->fStartTime = *theStartTimeP - theBackupTime;


            (*theFile)->fAllowNegativeTTs = true;
        }
    }

    if (qtFileErr == QTRTPFile::errCallAgain)
    {
        //
        // If we are doing RTP-Meta-Info stuff, we might be asked to get called again here.
        // This is simply because seeking might be a long operation and we don't want to
        // monopolize the CPU, but there is no other reason to wait, so just set a timeout of 0
        theErr = QTSS_SetIdleTimer(1);
        Assert(theErr == QTSS_NoErr);
        return theErr;
    }
    else if (qtFileErr != QTRTPFile::errNoError)
        return QTSSModuleUtils::SendErrorResponse(  inParamBlock->inRTSPRequest,
                                                    qtssClientBadRequest,
sSeekToNonexistentTimeErr);
```

```
    //make sure to clear the next packet the server would have sent!
    (*theFile)->fPacketStruct.packetData = NULL;

    // Set the movie duration and size parameters
    Float64 movieDuration = (*theFile)->fFile.GetMovieDuration();
    (void)QTSS_SetValue(inParamBlock->inClientSession, qtssCliSesMovieDurationInSecs, 0,
&movieDuration, sizeof(movieDuration));

    UInt64 movieSize = (*theFile)->fFile.GetAddedTracksRTPBytes();
    (void)QTSS_SetValue(inParamBlock->inClientSession, qtssCliSesMovieSizeInBytes, 0, &movieSize,
sizeof(movieSize));

    UInt32 bitsPerSecond =  (*theFile)->fFile.GetBytesPerSecond() * 8;
    (void)QTSS_SetValue(inParamBlock->inClientSession, qtssCliSesMovieAverageBitRate, 0,
&bitsPerSecond, sizeof(bitsPerSecond));

    Bool16 adjustPauseTime = kAddPauseTimeToRTPTime; //keep rtp time stamps monotonically increasing
    if ( true == QTSSModuleUtils::HavePlayerProfile( sServerPrefs,
inParamBlock,QTSSModuleUtils::kDisablePauseAdjustedRTPTime) )
     adjustPauseTime = kDontAddPauseTimeToRTPTime;

     if (sPlayerCompatibility )  // don't change adjust setting if compatibility is off.
       (**theFile).fAdjustPauseTime = adjustPauseTime;

    if ( (**theFile).fLastPauseTime > 0 )
        (**theFile).fTotalPauseTime += OS::Milliseconds() - (**theFile).fLastPauseTime;

    // For the purposes of the speed header, check to make sure all tracks are over a reliable
transport
    Bool16 allTracksReliable = true;

    // Set the timestamp & sequence number parameters for each track.
    QTSS_RTPStreamObject* theRef = NULL;
    for (   UInt32 theStreamIndex = 0;
            QTSS_GetValuePtr(inParamBlock->inClientSession, qtssCliSesStreamObjects, theStreamIndex,
(void**)&theRef, &theLen) == QTSS_NoErr;
            theStreamIndex++)
    {
        UInt32* theTrackID = NULL;
        theErr = QTSS_GetValuePtr(*theRef, qtssRTPStrTrackID, 0, (void**)&theTrackID, &theLen);
        Assert(theErr == QTSS_NoErr);
        Assert(theTrackID != NULL);
        Assert(theLen == sizeof(UInt32));

        UInt16 theSeqNum = 0;
        UInt32 theTimestamp = (*theFile)->fFile.GetSeekTimestamp(*theTrackID); // this is the base
timestamp need to add in paused time.

     Assert(theRef != NULL);

        if ((**theFile).fAdjustPauseTime)
        {
            UInt32* theTimescale = NULL;
            QTSS_GetValuePtr(*theRef, qtssRTPStrTimescale, 0,  (void**)&theTimescale, &theLen);
            if (theLen != 0) // adjust the timestamps to reflect paused time else leave it alone we
can't calculate the timestamp without a timescale.
            {
                UInt32 pauseTimeStamp = CalculatePauseTimeStamp( *theTimescale,  (*theFile)-
>fTotalPauseTime, (UInt32) theTimestamp);
         if (pauseTimeStamp != theTimestamp)
                        theTimestamp = pauseTimeStamp;
            }
        }

     theSeqNum = (*theFile)->fFile.GetNextTrackSequenceNumber(*theTrackID);
        theErr = QTSS_SetValue(*theRef, qtssRTPStrFirstSeqNumber, 0, &theSeqNum, sizeof(theSeqNum));
        Assert(theErr == QTSS_NoErr);
        theErr = QTSS_SetValue(*theRef, qtssRTPStrFirstTimestamp, 0, &theTimestamp,
sizeof(theTimestamp));
        Assert(theErr == QTSS_NoErr);

        if (allTracksReliable)
        {
            QTSS_RTPTransportType theTransportType = qtssRTPTransportTypeUDP;
            theLen = sizeof(theTransportType);
            theErr = QTSS_GetValue(*theRef, qtssRTPStrTransportType, 0, &theTransportType, &theLen);
            Assert(theErr == QTSS_NoErr);
```

```
            if (theTransportType == qtssRTPTransportTypeUDP)
                allTracksReliable = false;
        }
    }

    //Tell the QTRTPFile whether repeat packets are wanted based on the transport we don't care if
it doesn't set (i.e. this is a meta info session)
     (void)  (*theFile)->fFile.SetDropRepeatPackets(allTracksReliable);// if alltracks are reliable
then drop repeat packets.

    //
    // This module supports the Speed header if the client wants the stream faster than normal.
    Float32 theSpeed = 1;
    theLen = sizeof(theSpeed);
    theErr = QTSS_GetValue(inParamBlock->inRTSPRequest, qtssRTSPReqSpeed, 0, &theSpeed, &theLen);
    Assert(theErr != QTSS_BadArgument);
    Assert(theErr != QTSS_NotEnoughSpace);

    if (theErr == QTSS_NoErr)
    {
        if (theSpeed > sMaxAllowedSpeed)
            theSpeed = sMaxAllowedSpeed;
        if ((theSpeed <= 0) || (!allTracksReliable))
            theSpeed = 1;
    }

    (*theFile)->fSpeed = theSpeed;

    if (theSpeed != 1)
    {
        //
        // If our speed is not 1, append the RTSP speed header in the response
        char speedBuf[32];
        qtss_sprintf(speedBuf, "%10.5f", theSpeed);
        StrPtrLen speedBufPtr(speedBuf);
        (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssSpeedHeader, speedBufPtr.Ptr,
speedBufPtr.Len);
    }

    //
    // Record the requested stop time, if there is one
    (*theFile)->fStopTime = -1;
    theLen = sizeof((*theFile)->fStopTime);
    theErr = QTSS_GetValue(inParamBlock->inRTSPRequest, qtssRTSPReqStopTime, 0, &(*theFile)-
>fStopTime, &theLen);

    //
    // Append x-Prebuffer header if provided & nonzero prebuffer needed
    if (theBackupTime > 0)
    {
        char prebufferBuf[32];
        qtss_sprintf(prebufferBuf, "time=%.5f", theBackupTime);
        StrPtrLen backupTimePtr(prebufferBuf);
        (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssXPreBufferHeader,
backupTimePtr.Ptr, backupTimePtr.Len);

    }

    // add the range header.
    {
        char rangeHeader[64];
        if (-1 == (*theFile)->fStopTime)
           (*theFile)->fStopTime = (*theFile)->fFile.GetMovieDuration();

        qtss_snprintf(rangeHeader,sizeof(rangeHeader) -1, "npt=%.5f-%.5f", (*theFile)->fStartTime,
(*theFile)->fStopTime);
        rangeHeader[sizeof(rangeHeader) -1] = 0;

        StrPtrLen rangeHeaderPtr(rangeHeader);
        (void)QTSS_AppendRTSPHeader(inParamBlock->inRTSPRequest, qtssRangeHeader,rangeHeaderPtr.Ptr,
rangeHeaderPtr.Len);

    }
    //Send response of the active tracks only
    UInt32 activeTracks[2] = {0,0};
    (*theFile)->fFile.getActiveTracks(activeTracks);
    (void)QTSS_SendStandardRTSPResponse(inParamBlock->inRTSPRequest, inParamBlock->inClientSession,
qtssPlayRespWriteTrackInfo,activeTracks[0], activeTracks[1]);
```

```
    SInt64 adjustRTPStreamStartTimeMilli = 0;
    if (sPlayerCompatibility && QTSSModuleUtils::HavePlayerProfile(sServerPrefs,
inParamBlock,QTSSModuleUtils::kDelayRTPStreamsUntilAfterRTSPResponse))
        adjustRTPStreamStartTimeMilli = sAdjustRTPStartTimeMilli;

    //Tell the server to start playing this movie. We do want it to send RTCP SRs, but we DON'T want
it to write the RTP header
    (*theFile)->fPaused = false;

    theErr = QTSS_Play(inParamBlock->inClientSession, inParamBlock->inRTSPRequest,
qtssPlayFlagsSendRTCP);
    if (theErr == QTSS_BadArgument)
       qtss_printf("QTSS_BadArgument\n");
    if (theErr == QTSS_RequestFailed)
       qtss_printf("QTSS_RequestFailed\n");
    if (theErr != QTSS_NoErr)
       return theErr;

    // Set the adjusted play time. SendPackets can get called between QTSS_Play and setting
fAdjustedPlayTime below.
    SInt64* thePlayTime = NULL;
    theErr = QTSS_GetValuePtr(inParamBlock->inClientSession, qtssCliSesPlayTimeInMsec, 0,
(void**)&thePlayTime, &theLen);
    Assert(theErr == QTSS_NoErr);
    Assert(thePlayTime != NULL);
    Assert(theLen == sizeof(SInt64));
    if (thePlayTime != NULL)
        (*theFile)->fAdjustedPlayTime = adjustRTPStreamStartTimeMilli + *thePlayTime -
((SInt64)((*theFile)->fStartTime * 1000) );

    return QTSS_NoErr;
}

QTSS_Error SendPackets(QTSS_RTPSendPackets_Params* inParams)
{
    static const UInt32 kQualityCheckIntervalInMsec = 250;  // v331=v107
    FileSession** theFile = NULL;
    UInt32 theLen = 0;
    QTSS_Error theErr = QTSS_GetValuePtr(inParams->inClientSession, sFileSessionAttr, 0,
(void**)&theFile, &theLen);
    Assert(theErr == QTSS_NoErr);
    Assert(theLen == sizeof(FileSession*));
    bool isBeginningOfWriteBurst = true;
    QTSS_Object theStream = NULL;


    if ( theFile == NULL || (*theFile)->fStartTime == -1 || (*theFile)->fPaused == true )
//something is wrong
    {
        Assert( theFile != NULL );
        Assert( (*theFile)->fStartTime != -1 );
        Assert( (*theFile)->fPaused != true );

        inParams->outNextPacketTime = qtssDontCallSendPacketsAgain;
        return QTSS_NoErr;
    }

    if ( (*theFile)->fAdjustedPlayTime == 0 ) // this is system milliseconds
    {
        Assert( (*theFile)->fAdjustedPlayTime != 0 );
        inParams->outNextPacketTime = kQualityCheckIntervalInMsec;
        return QTSS_NoErr;
    }

    QTRTPFile::RTPTrackListEntry* theLastPacketTrack = (*theFile)->fFile.GetLastPacketTrack();

    while (true)
    {
        if ((*theFile)->fPacketStruct.packetData == NULL)
        {
            Float64 theTransmitTime = (*theFile)->fFile.GetNextPacket((char**)&(*theFile)-
>fPacketStruct.packetData, &(*theFile)->fNextPacketLen);
            if ( QTRTPFile::errNoError != (*theFile)->fFile.Error() )
            {
                QTSS_CliSesTeardownReason reason = qtssCliSesTearDownUnsupportedMedia;
                (void) QTSS_SetValue(inParams->inClientSession, qtssCliTeardownReason, 0, &reason,
sizeof(reason));
                (void)QTSS_Teardown(inParams->inClientSession);
                return QTSS_RequestFailed;
```

```
            }
           theLastPacketTrack = (*theFile)->fFile.GetLastPacketTrack();

        if (!theLastPacketTrack->IsTrackActive)
               continue;

               if (theLastPacketTrack == NULL)
                      break;

            theStream = (QTSS_Object)theLastPacketTrack->Cookie1;
               Assert(theStream != NULL);
               if (theStream == NULL)
                      return 0;

            // Check to see if we should stop playing now

            if (((*theFile)->fStopTime != -1) && (theTransmitTime > (*theFile)->fStopTime))
            {
                // We should indeed stop playing
                (void)QTSS_Pause(inParams->inClientSession);
                inParams->outNextPacketTime = qtssDontCallSendPacketsAgain;

                (**theFile).fPaused = true;
                (**theFile).fLastPauseTime = OS::Milliseconds();

                return QTSS_NoErr;
            }
            if (((*theFile)->fStopTrackID != 0) && ((*theFile)->fStopTrackID == theLastPacketTrack-
>TrackID) && (theLastPacketTrack->HTCB->fCurrentPacketNumber > (*theFile)->fStopPN))
            {
                // We should indeed stop playing
                (void)QTSS_Pause(inParams->inClientSession);
                inParams->outNextPacketTime = qtssDontCallSendPacketsAgain;

                (**theFile).fPaused = true;
                (**theFile).fLastPauseTime = OS::Milliseconds();

                return QTSS_NoErr;
            }

            //
            // Find out what our play speed is. Send packets out at the specified rate and do so by
altering the transmit time of the packet based on the Speed rate.
            Float64 theOffsetFromStartTime = theTransmitTime - (*theFile)->fStartTime;
            theTransmitTime = (*theFile)->fStartTime + (theOffsetFromStartTime / (*theFile)-
>fSpeed);

            //
            // correct for first packet xmit times that are < 0
            if (( theTransmitTime < 0.0 ) && ( !(*theFile)->fAllowNegativeTTs ))
                theTransmitTime = 0.0;

            (*theFile)->fPacketStruct.packetTransmitTime = (*theFile)->fAdjustedPlayTime +
((SInt64)(theTransmitTime * 1000));
        }

        //We are done playing all streams!
        if ((*theFile)->fPacketStruct.packetData == NULL)
        {
            //TODO not quite good to the last drop -- we -really- should guarantee this, also
reflector a write of 0 len to QTSS_Write will flush any buffered data if we're
sending over tcp

               inParams->outNextPacketTime = qtssDontCallSendPacketsAgain;
            return QTSS_NoErr;
        }

        //we have a packet that needs to be sent now
        Assert(theLastPacketTrack != NULL);

     //If the stream is video, we need to make sure that QTRTPFile knows what quality level we're at

        if ( (!sDisableThinning) && (inParams->inCurrentTime > ((*theFile)->fLastQualityCheck +
kQualityCheckIntervalInMsec) ) )
        {
            QTSS_RTPPayloadType thePayloadType = (QTSS_RTPPayloadType)theLastPacketTrack->Cookie2;
            if (thePayloadType == qtssVideoPayloadType)
            {
                (*theFile)->fLastQualityCheck = inParams->inCurrentTime;
```

```
                        theStream = (QTSS_Object)theLastPacketTrack->Cookie1;
                        Assert(theStream != NULL);
                        if (theStream == NULL)
                                return 0;

                // Get the current quality level in the stream, and this stream's TrackID.
                UInt32* theQualityLevel = 0;
                theErr = QTSS_GetValuePtr(theStream, qtssRTPStrQualityLevel, 0,
(void**)&theQualityLevel, &theLen);
                Assert(theErr == QTSS_NoErr);
                Assert(theQualityLevel != NULL);
                Assert(theLen == sizeof(UInt32));

        //we check max quality level
        UInt32* maxQualityLevel = 0;
        theErr = QTSS_GetValuePtr(theStream, qtssRTPStrNumQualityLevels, 0,
(void**)&maxQualityLevel, &theLen);
        Assert(theErr == QTSS_NoErr);
                Assert(theQualityLevel != NULL);
                Assert(theLen == sizeof(UInt16));
                (*theFile)->fFile.SetTrackQualityLevel(theLastPacketTrack, *theQualityLevel,
*maxQualityLevel);
        (*theFile)->fAllowNegativeTTs = false;
            }
        }

     char * punter = (char *)(*theFile)->fPacketStruct.packetData;

     if (theLastPacketTrack->Payload == 1) punter[1]=((punter[1] & 0xffffff80) | 0x00000060);
     else punter[1]=((punter[1] & 0xffffff80) | 0x00000061);

        // Send the packet!
        QTSS_WriteFlags theFlags = qtssWriteFlagsIsRTP;
        if (isBeginningOfWriteBurst)
            theFlags |= qtssWriteFlagsWriteBurstBegin;

        theStream = (QTSS_Object)theLastPacketTrack->Cookie1;
        Assert(theStream != NULL);
        if (theStream == NULL)
               return 0;

        //adjust the timestamp so it reflects paused time.
        void* packetDataPtr =  (*theFile)->fPacketStruct.packetData;
        UInt32 currentTimeStamp = GetPacketTimeStamp(packetDataPtr);
        UInt32 pauseTimeStamp = SetPausetimeTimeStamp(*theFile, theStream, currentTimeStamp);
        UInt16 curSeqNum = GetPacketSequenceNumber(theStream);

        (void) QTSS_SetValue(theStream, sRTPStreamLastPacketSeqNumAttrID, 0, &curSeqNum,
sizeof(curSeqNum));

     theErr = QTSS_Write(theStream, &(*theFile)->fPacketStruct, (*theFile)->fNextPacketLen, NULL,
theFlags);

        isBeginningOfWriteBurst = false;

        if ( theErr == QTSS_WouldBlock )
        {
            if (currentTimeStamp != pauseTimeStamp) // reset the packet time stamp so we adjust it
again when we really do send it
                SetPacketTimeStamp(currentTimeStamp, packetDataPtr);

// In the case of a QTSS_WouldBlock error, the packetTransmitTime field of the packet struct will be
set to the time to wakeup, or -1 if not known. If the time to wakeup is not given by the server,
just give a fixed guess interval
            if ((*theFile)->fPacketStruct.suggestedWakeupTime == -1)
                inParams->outNextPacketTime = sFlowControlProbeInterval;    // for buffering, try me
again in # MSec
            else
            {
                Assert((*theFile)->fPacketStruct.suggestedWakeupTime > inParams->inCurrentTime);
                inParams->outNextPacketTime = (*theFile)->fPacketStruct.suggestedWakeupTime -
inParams->inCurrentTime;
            }

            return QTSS_NoErr;
        }
        else
        {
```

```
          (void) QTSS_SetValue(theStream, sRTPStreamLastSentPacketSeqNumAttrID, 0, &curSeqNum,
sizeof(curSeqNum));

          (*theFile)->fPacketStruct.packetData = NULL;
        }
    }
    return QTSS_NoErr;
}

//-------------------QTSS_SendStandardRTSPResponse en QTRTPFile.cpp-------------------//

QTSS_Error QTSS_SendStandardRTSPResponse(QTSS_RTSPRequestObject inRTSPRequest, QTSS_Object
inRTPInfo, UInt32 inFlags,...)
{
    va_list param_pt;
    va_start(param_pt, inFlags);
    return (sCallbacks->addr [kSendStandardRTSPCallback]) (inRTSPRequest, inRTPInfo,
inFlags,va_arg(param_pt, UInt32),va_arg(param_pt, UInt32));
}

//-------------------QTSS_SendStandardRTSPResponse en QTRTPFile.cpp-------------------//

QTSS_Error  QTSS_SendStandardRTSPResponse(QTSS_RTSPRequestObject inRTSPRequest, QTSS_Object
inRTPInfo, UInt32 inFlags,...);

//--Redefinición de la estructura RTPTrackListEntry y las principales funciones en QTRTPFile.--//

    struct RTPTrackListEntry {

        //
        // Track information
        UInt32          TrackID;
        QTHintTrack     *HintTrack;
        QTHintTrack_HintTrackControlBlock   *HTCB;
        Bool16          IsTrackActive, IsPacketAvailable;
        UInt32          QualityLevel;

        //
        // Server information
        void            *Cookie1;
        UInt32          Cookie2;
        UInt32          SSRC;
        UInt16          FileSequenceNumberRandomOffset, BaseSequenceNumberRandomOffset,
                        LastSequenceNumber;
        SInt32          SequenceNumberAdditive;
        UInt32          FileTimestampRandomOffset, BaseTimestampRandomOffset;

        //
        // Sample/Packet information
        UInt32          CurSampleNumber;
        UInt32          ConsecutivePFramesSent;
        UInt32          TargetPercentage;
        UInt32          SampleToSeekTo;
        UInt32          LastSyncSampleNumber;
        UInt32          NextSyncSampleNumber;
        UInt16          NumPacketsInThisSample, CurPacketNumber;

        Float64         CurPacketTime;
        char            CurPacket[QTRTPFILE_MAX_PACKET_LENGTH];
        UInt32          CurPacketLength;

        //
        // Quality swapping files
        UInt32         Group;
        UInt16         Payload;
        UInt32         NextQTrackID;
        UInt32         PrevQTrackID;

        //
        // List pointers
        RTPTrackListEntry   *NextTrack;
    };

         ErrorCode   ActivateTrackID(UInt32 TrackID);
         ErrorCode   DeActivateTrackID(UInt32 TrackID);
         Bool16       IsActiveTrackID(UInt32 TrackID);
         int   getActiveTracks(UInt32 activeTracks[2]);
         ErrorCode   SwapTracks(Bool16 increaseQ);
         char *      MyGetSDPFile(int * SDPFileLength, int * totalSDPFileLength);
```

```
        UInt32        maxBWTrack(UInt32 trackPayload, UInt32 maxBW);
        UInt32        minBWTrack(UInt32 trackPayload, UInt32 maxBW);
           void SetTrackQualityLevel(RTPTrackListEntry* inEntry, UInt32 inNewLevel, UInt32
inMaxLevel);

//-------------------MyGetSDPFile en QTRTPFile.cpp-------------------//

char * QTRTPFile::MyGetSDPFile(int * sdpFileLength, int * totalsdpFileLength)
{
    // Temporary vars
    RTPTrackListEntry*  curEntry;
    UInt32              tempAtomType;

    // General vars
    QTFile::AtomTOCEntry*   globalSDPTOCEntry;
    Bool16                  haveGlobalSDPAtom = false;
    char                    sdpRangeLine[256];
    char*                   pSDPFile=NULL;
    int                   newLen=0;

    //
    // Build our range header.
    qtss_sprintf(sdpRangeLine, "a=range:npt=0-%10.5f\r\n", this->GetMovieDuration());

    //
    // Figure out how long the SDP file is going to be.
    fSDPFileLength = ::strlen(sdpRangeLine);

   for ( curEntry = fFirstTrack;curEntry != NULL;curEntry = curEntry->NextTrack)
    {
        // Temporary vars
        int         trackSDPLength;

        //
        // Get the length of this track's SDP file.
        if( curEntry->HintTrack->GetSDPFileLength(&trackSDPLength) != QTTrack::errNoError)
            continue;

        //
        // Add it to our count.
        fSDPFileLength += trackSDPLength;
    }

    //
    // See if this movie has a global SDP atom.
    if( fFile->FindTOCEntry("moov:udta:hnti:rtp ", &globalSDPTOCEntry, NULL) )
    {
        //
        // Verify that this is an SDP atom.
        fFile->Read(globalSDPTOCEntry->AtomDataPos, (char *)&tempAtomType, 4);

        if ( ntohl(tempAtomType) == FOUR_CHARS_TO_INT('s', 'd', 'p', ' ') )
        {
            haveGlobalSDPAtom = true;
            fSDPFileLength += (UInt32) (globalSDPTOCEntry->AtomDataLength - 4);
         newLen += (UInt32) (globalSDPTOCEntry->AtomDataLength - 4);
        }
    }

    //
    // Build the SDP file.
    debug_printf("fSDPFileLength1 %u\n", fSDPFileLength);
    fSDPFile = pSDPFile = NEW char[fSDPFileLength + 1];
    if( fSDPFile == NULL )
    {   fErr = errInternalError;
        return NULL;
    }

    if( haveGlobalSDPAtom )
    {
        fFile->Read(globalSDPTOCEntry->AtomDataPos + 4, pSDPFile, (UInt32) (globalSDPTOCEntry-
>AtomDataLength - 4) );
        pSDPFile += globalSDPTOCEntry->AtomDataLength - 4;
    }

    ::memcpy(pSDPFile, sdpRangeLine, ::strlen(sdpRangeLine));
    pSDPFile += ::strlen(sdpRangeLine);
    newLen += ::strlen(sdpRangeLine);
```

```
      for ( curEntry = fFirstTrack;
            curEntry != NULL;
            curEntry = curEntry->NextTrack
          )
      {
          // Temporary vars
          char        *temptrackSDP;
          int         temptrackSDPLength;

          //
          // Get this track's SDP file and add it to our buffer.
          temptrackSDP = curEntry->HintTrack->GetSDPFile(&temptrackSDPLength);

     curEntry->Group=atoi(strstr(temptrackSDP,"b=AS:")+5);
     curEntry->Payload = strstr(temptrackSDP,"video")?1:0;
     debug_printf("Added track: %u, bandwidth: %u\n", curEntry->TrackID, curEntry->Group);

     delete [] temptrackSDP;
        }

 for (curEntry = fFirstTrack;curEntry != NULL;curEntry = curEntry->NextTrack)
 {
  curEntry->PrevQTrackID = minBWTrack(curEntry->Payload,curEntry->Group);
  debug_printf("Current track: %u - ", curEntry->TrackID);
  curEntry->NextQTrackID = maxBWTrack(curEntry->Payload,curEntry->Group);
 }

 // Add higher bitrate tracks only
 UInt32 tracksChosen[2]={0,0};
 for ( curEntry = fFirstTrack; curEntry != NULL; curEntry = curEntry->NextTrack)
 {
  if (curEntry->NextQTrackID == 0 && tracksChosen[curEntry->Payload]==0)
  {
            // Temporary vars
            char        *trackSDP;
            int         trackSDPLength = 0;

            //
            // Get this track's SDP file and add it to our buffer.
            trackSDP = curEntry->HintTrack->GetSDPFile(&trackSDPLength);
     if( trackSDP == NULL)
                 continue;
            ::memcpy(pSDPFile, trackSDP, trackSDPLength);
            delete [] trackSDP;//ARGH! GetSDPFile allocates the pointer that is being returned.

     const char * newInfo =(curEntry->Payload == 1)?"96":"97";

     char * myPtr = NULL;
     if(myPtr = strstr(pSDPFile,"RTP/AVP "))
     ::memcpy(myPtr+8, newInfo, 2);
     if(myPtr = strstr(pSDPFile,"rtpmap:"))
     ::memcpy(myPtr+7, newInfo, 2);
           if(myPtr = strstr(pSDPFile,"fmtp:"))
     ::memcpy(myPtr+5, newInfo, 2);

     pSDPFile += trackSDPLength;
     newLen += trackSDPLength;
     tracksChosen[curEntry->Payload]=curEntry->TrackID;

     debug_printf("Added track: %u - ", curEntry->TrackID);
  }
 }

  *totalsdpFileLength = newLen;

for ( curEntry = fFirstTrack; curEntry != NULL; curEntry = curEntry->NextTrack)
 {

  if (tracksChosen[curEntry->Payload]==curEntry->TrackID)
  continue;
  {
            // Temporary vars
            char        *trackSDP;
            int         trackSDPLength = 0;

            //
            // Get this track's SDP file and add it to our buffer.
            trackSDP = curEntry->HintTrack->GetSDPFile(&trackSDPLength);
```

```
        if( trackSDP == NULL)
                continue;
                ::memcpy(pSDPFile, trackSDP, trackSDPLength);
                delete [] trackSDP;//ARGH! GetSDPFile allocates the pointer that is being returned.
        pSDPFile += trackSDPLength;

        *totalsdpFileLength += trackSDPLength;
      }
     }

    // Return the (cached) SDP file.
     *sdpFileLength = newLen;
     fSDPFile[fSDPFileLength] = 0;
     return fSDPFile;
}

//-------------------getActiveTracks en QTRTPFile.cpp-------------------//

int QTRTPFile::getActiveTracks(UInt32 activeTracks[2])
{

    RTPTrackListEntry *curEntry = NULL;
    UInt32 i = 0;

for ( curEntry = fFirstTrack;curEntry != NULL;curEntry = curEntry->NextTrack)
      {
      if (IsActiveTrackID(curEntry->TrackID))
      activeTracks[i++]=curEntry->TrackID;
      }

    return 1;
}

//-------------------maxBWTrack en QTRTPFile.cpp-------------------//

UInt32 QTRTPFile::maxBWTrack(UInt32 trackPayload, UInt32 maxBW)
{
//
// Returns the inmediatly greater bw track over maxBW rate with the same payload type

    RTPTrackListEntry*  curEntry;
    UInt32 trackID = 0;
    UInt32 tempBW = 0xffffffff;

    for (curEntry = fFirstTrack; curEntry != NULL; curEntry = curEntry->NextTrack)
    {
     if (curEntry->Payload==trackPayload && curEntry->Group>maxBW && curEntry->Group<tempBW)
        {
        trackID=curEntry->TrackID;
        tempBW=curEntry->Group;
        }
     }

     debug_printf("Next Track: %u, bw: %u\n", trackID, tempBW);
     return trackID;
}

//-------------------minBWTrack en QTRTPFile.cpp-------------------//

UInt32 QTRTPFile::minBWTrack(UInt32 trackPayload, UInt32 maxBW)
{
//
// Returns the inmediatly smaller bw track over maxBW rate with the same payload type
    RTPTrackListEntry*  curEntry;
    UInt32 trackID=0;
    UInt32 tempBW = 0;

    for (curEntry = fFirstTrack; curEntry != NULL; curEntry = curEntry->NextTrack)
    {
     if (curEntry->Payload==trackPayload && curEntry->Group<maxBW && curEntry->Group>tempBW)
        {
        trackID=curEntry->TrackID;
        tempBW=curEntry->Group;
        }
    }
    debug_printf("Previous Track: %u, bw: %u - ", trackID, tempBW);

    return trackID;
}
```

```
//-------------------ActivateTrackID en QTRTPFile.cpp-------------------//

QTRTPFile::ErrorCode QTRTPFile::ActivateTrackID(UInt32 TrackID)
{
    RTPTrackListEntry *trackEntry = NULL;

    if( !this->FindTrackEntry(TrackID, &trackEntry) )
     return fErr =  errTrackIDNotFound;

    trackEntry->IsTrackActive = true;
    trackEntry->IsPacketAvailable = true;

    return errNoError;
}

//-------------------DeActivateTrackID en QTRTPFile.cpp-------------------//

QTRTPFile::ErrorCode QTRTPFile::DeActivateTrackID(UInt32 TrackID)
{
 RTPTrackListEntry *trackEntry = NULL;

 if( !this->FindTrackEntry(TrackID, &trackEntry) )
  return fErr =  errTrackIDNotFound;

 trackEntry->IsTrackActive = false;

 return errNoError;
}

//-------------------IsActiveTrackID en QTRTPFile.cpp-------------------//

Bool16 QTRTPFile::IsActiveTrackID(UInt32 TrackID)
{
    RTPTrackListEntry *trackEntry = NULL;

    if( !this->FindTrackEntry(TrackID, &trackEntry) )
     return false;

    return trackEntry->IsTrackActive;
}

//-------------------SwapTracks en QTRTPFile.cpp-------------------//

QTRTPFile::ErrorCode QTRTPFile::SwapTracks(Bool16 increaseQ)
{
    UInt32 OldTrackID, NewTrackID;
    RTPTrackListEntry *oldTrackEntry = NULL;
    RTPTrackListEntry *newTrackEntry = NULL;
    RTPTrackListEntry *curEntry = NULL;
    UInt16 updated[2] = {0,0};

    for ( curEntry = fFirstTrack;curEntry != NULL;curEntry = curEntry->NextTrack)
     {
     if (IsActiveTrackID(curEntry->TrackID) && updated[curEntry->Payload]==0)
        {
        OldTrackID = curEntry->TrackID;
        NewTrackID = increaseQ==true?curEntry->NextQTrackID:curEntry->PrevQTrackID;
        if (NewTrackID==0 || OldTrackID == NewTrackID)
        continue;

        if( !this->FindTrackEntry(OldTrackID, &oldTrackEntry) )
          return fErr =  errTrackIDNotFound;

        if( !this->FindTrackEntry(NewTrackID, &newTrackEntry) )
         return fErr =  errTrackIDNotFound;

        ::memcpy((char *)newTrackEntry->CurPacket + 2, (char *)oldTrackEntry->CurPacket + 2, 2);
        //
        // Read the sequence number right out of the packet.
        for(int i=0; i<QTRTPFILE_MAX_PACKET_LENGTH; i++)
                {
                newTrackEntry->CurPacket[i]=oldTrackEntry->CurPacket[i];
                }

        newTrackEntry->QualityLevel = oldTrackEntry->QualityLevel;
        newTrackEntry->Cookie1 = oldTrackEntry->Cookie1;
        newTrackEntry->LastSequenceNumber = oldTrackEntry->LastSequenceNumber;
        newTrackEntry->SequenceNumberAdditive = oldTrackEntry->SequenceNumberAdditive;
```

```
        newTrackEntry->CurPacketNumber = oldTrackEntry->CurPacketNumber;
        newTrackEntry->CurSampleNumber = (oldTrackEntry->CurSampleNumber);
        newTrackEntry->SSRC = oldTrackEntry->SSRC;
        oldTrackEntry->IsTrackActive = false;
        oldTrackEntry->IsPacketAvailable = false;
        newTrackEntry->IsTrackActive = true;
        newTrackEntry->IsPacketAvailable = true;

        debug_printf("\nSWAP: oldtrack %u - newtrack %u - curSampleNumber %u - %u, CurPacketTime %f
- %f, LastSeqNum %u - %u packetAVAI %u\n\n",                          OldTrackID, NewTrackID,
newTrackEntry->CurSampleNumber, oldTrackEntry->CurSampleNumber, newTrackEntry->CurPacketTime,
oldTrackEntry->CurPacketTime,newTrackEntry->LastSequenceNumber, oldTrackEntry->LastSequenceNumber,
newTrackEntry->IsPacketAvailable);

        Seek(oldTrackEntry->CurPacketTime, 0);

        return errNoError;
        }
     }
return errNoError;
}

//-------------------SetTrackQualityLevel en QTRTPFile.cpp-------------------//

void QTRTPFile::SetTrackQualityLevel(RTPTrackListEntry* inEntry, UInt32 inNewQualityLevel, UInt32
inMaxQualityLevel)
{

  if (inNewQualityLevel != inEntry->QualityLevel)
    {
     UInt32 threshold = 3*(inMaxQualityLevel/4);
     if (inNewQualityLevel>threshold && inEntry->QualityLevel<inNewQualityLevel && inEntry-
>QualityLevel!=0)
        {
        inEntry->QualityLevel = inMaxQualityLevel*0.4;
        this->SwapTracks(false);
        return;
        }
     else if (inNewQualityLevel == 0 && inEntry->QualityLevel>inNewQualityLevel && inEntry-
>QualityLevel!=10)
        {
        inEntry->QualityLevel = 0;
        this->SwapTracks(true);
        return;
        }

     inEntry->QualityLevel = inNewQualityLevel;
     }
}

//-------------------UpdateTimeAndQuality en RTPStream3GPP.cpp-------------------//

void RTPStream3GPP::UpdateTimeAndQuality(SInt64 curTime)
{
     fAdjustTime = kNoChange;
     fAdjustSize = kNoChange;

    if (!RateAdaptationEnabled())
        return;

    Bool16 limitTargetDelay = true;
    UInt32 maxTargetDelayMilli = 10000;// 5000;

    UInt32 adjustedFreeBuffer = fLastReportedFreeBufferSpace;
    UInt32 adjustTargetDelay = fTargetBufferingDelay;

    if (limitTargetDelay)
    {
        if (adjustTargetDelay > maxTargetDelayMilli) //make this constant a pref
            adjustTargetDelay = maxTargetDelayMilli;
    }

    if (fBufferSize > RTCPNaduPacket::kMaximumReportableFreeBufferSpace)
        fBufferSize = RTCPNaduPacket::kMaximumReportableFreeBufferSpace;

    UInt32 maxUsableBufferSizeBytes = fBufferSize;
    UInt32 extraBufferMultipleInSeconds = 2; // use up to 3 times the requested target delay in
bytes
    UInt32 maxUsableDelaySecs = extraBufferMultipleInSeconds * (adjustTargetDelay/1000);
```

```
    UInt32 movieByteRate = fMovieBitRate >> 3; // bits to bytes
    UInt32 bufferUsage = fBufferSize - fLastReportedFreeBufferSpace;
    UInt32 bufferDelay = fLastReportedBufferingDelay;

    if (bufferUsage > fBufferSize)  //there is more reported free buffer than the maximum -- not
good for our ratios but a good situation at the client i.e. no buffer overrun here.
    {
        bufferUsage = fBufferSize / 2; // Have to pick something so use 50%.
    }

    DEBUG_PRINTF(("reported buffer size = %"_U32BITARG_" reported Free Buffer=%"_U32BITARG_"
current calculated bufferUsage= %"_U32BITARG_"\n", fBufferSize,
fLastReportedFreeBufferSpace,bufferUsage));
     DEBUG_PRINTF(("Avg Movie BitRate = %lu original target delay=%lu adjusted TargetDelay =%ld
\n",fMovieBitRate,fTargetBufferingDelay, adjustTargetDelay));

     if (qtssVideoPayloadType == fRTPStream.GetPayLoadType() && fMovieBitRate > 0 &&
adjustTargetDelay > 0) //limit how much we use
     {
         maxUsableBufferSizeBytes = maxUsableDelaySecs * movieByteRate; //buffer time * bit rate for
movie is bigger than any single stream buffer

        if (maxUsableBufferSizeBytes > fBufferSize) // reported size is smaller than our buffer
target
        {     maxUsableBufferSizeBytes = fBufferSize;
              if (maxUsableBufferSizeBytes < movieByteRate) //hope for the best
                maxUsableBufferSizeBytes = movieByteRate;
              UInt32 newTargetDelay = (maxUsableBufferSizeBytes / movieByteRate) * 1000;
              if (newTargetDelay < adjustTargetDelay)
                    adjustTargetDelay = newTargetDelay;
        }

        if (adjustedFreeBuffer > maxUsableBufferSizeBytes)
            adjustedFreeBuffer = maxUsableBufferSizeBytes ;

        UInt32 freeBytes = fBufferSize - bufferUsage;
        if (freeBytes > fBufferSize)
            bufferUsage = maxUsableBufferSizeBytes / 2;

         DEBUG_PRINTF(("ADJUSTING buffer usage and target delay: maxUsableBufferSizeBytes =%lu
adjustedFreeBuffer=%lu bufferUsage=%lu adjusted TargetDelay=%lu\n",maxUsableBufferSizeBytes,
adjustedFreeBuffer, bufferUsage, adjustTargetDelay));
     }

    DEBUG_PRINTF(("Calculated maxUsableBufferSize =%"_U32BITARG_" reported
fBufferSize=%"_U32BITARG_"  reported buffer delay=%"_U32BITARG_" current calculated bufferUsage=
%"_U32BITARG_"\n", maxUsableBufferSizeBytes, fBufferSize,bufferDelay, bufferUsage));



    //bufferDelay should really be the network delay because if buffer delay were really large that
would be ok
    // it is supposed to be -1 if not supported or a real value. Some clients send 0 incorrectly.
    //if buffer delay is small that should mean the buffer is empty and a under-run failure
occurred.
    if (bufferDelay == 0)
        bufferDelay = kUInt32_Max;


     double bufferUsageRatio = static_cast<double>(bufferUsage) / maxUsableBufferSizeBytes;
     double bufferDelayRatio = static_cast<double>(bufferDelay) / adjustTargetDelay;
        DEBUG_PRINTF(("bufferUsageRatio =%f bufferDelayRatio=%f\n", bufferUsageRatio,
bufferDelayRatio));

     if(!fStartDoingAdaptation)
     {
     //This is used to prevent QTSS from thinning in the beginning of the stream, when the buffering
delay and usage are expected to be low Rate adaptation will start when EITHER of the two low
watermarks for thinning have passed, OR the media has been playing for the target buffering delay.
The ideal situation for the current code is 2x or more buffer size to target time. So target time
converted to bytes should be 50% or less the buffer size to avoid overrun this one is agressive and
works well with Nokia when all is good and there is extra bandwidth so it makes a good network look
good.
        if (bufferUsageRatio >= 0.7) //start active rate adapt when client is 70% full
            {fStartDoingAdaptation = true; /*qtss_printf("bufferusage\n");*/}
        else if (curTime - fRTPStream.GetSession().GetFirstPlayTime() >= 15000) // but don't wait
longer than 15 seconds
            {fStartDoingAdaptation = true;}
        else //neither criteria was met. //speed up while waiting for the buffer to fill.
```

```
        {   fAdjustTime = kAdjustUpUp;
        }

      if (fStartDoingAdaptation)
        {               fNumLargeRTT = 0;
                fNumSmallRTT = -3; //Delay the first rate increase

        }
     }

    if (fStartDoingAdaptation)
    {
        SInt32 currentQualityLevel = fRTPStream.GetQualityLevel();
     if (currentQualityLevel >(fRTPStream.GetNumQualityLevels()*0.75) || currentQualityLevel < 1)
        {
        fRTPStream.SetQualityLevel(20);
        }
 // new code works good for Nokia N93 on wifi and ok for slow links (needs some more comparison
testing against non rate adapt code and against build 520 or earlier)

        if (bufferDelay != kUInt32_Max) //not supported
        {
            DEBUG_PRINTF(("rate adapt is using delay ratio and buffer size\n\n"));

            //The buffering delay information is available.

            //should I speed up or slow down?  A Delay Ratio of 100% is a target not a minimum and
not a maximum.
            if (bufferDelayRatio < 2.0) //allow up to 200%
                fAdjustTime = kAdjustUpUp;
            else
                fAdjustTime = kAdjustDown;

            if (bufferUsageRatio >= 0.7) //if you are in danger of buffer-overflowing because the
buffer size is too small for the movie, also slow
        fAdjustTime = kAdjustDown;
            else if (bufferUsageRatio < 0.5 && bufferDelayRatio > 2.5) // stop pushing.
                fAdjustTime = kAdjustDownDown;
            else if (bufferUsageRatio < 0.5 && bufferDelayRatio > 2.0) // stop pushing.
                fAdjustTime = kAdjustDown;
            else if (bufferUsageRatio < 0.5 && bufferDelayRatio > 0.5) // try to push up hard.
                {fAdjustTime = kAdjustUpUp;
        fAdjustSize = kAdjustUp;
        }
            //should I thin or thicken?

            if (bufferUsageRatio  < 0.2 && bufferDelayRatio > 2.5) // avoid underflow since the
bandwidth is low.
            {  fAdjustSize = kAdjustDown;
        fAdjustTime = kAdjustUpUp;
            }
            else if (bufferUsageRatio <= 0.1 ) //try thickening
            {  fAdjustSize = kAdjustUpUp;
        fAdjustTime = kAdjustUpUp;
            }
            else if (bufferUsageRatio <= 0.3 && bufferDelayRatio < 1.0) //still in danger of
underflow
               fAdjustSize = kNoChange;
            else if (bufferUsageRatio < 0.7 ) //no longer in danger of underflow; ok to thick
               fAdjustSize = kAdjustUp;
            else
                        fAdjustSize = kNoChange;
        }
      else
        {
            DEBUG_PRINTF(("rate adapt is using only buffer size\n"));

            //The buffering delay is not available; we make thin/slow decisions based on just the
buffer usage alone
            if (bufferUsageRatio  > 0.9) //need to slow and thin to avoid overflow
            {
                fAdjustSize = kAdjustDown;
                fAdjustTime = kAdjustDown;
            }
         else if (bufferUsageRatio > 0.8) //need to slow and thin to avoid overflow
            {
                fAdjustSize = kNoChange;
                fAdjustTime = kAdjustDown;
            }
```

```
            else if (bufferUsageRatio > 0.7) //need to slow and thin to avoid overflow
            {
                fAdjustSize = kAdjustUp;
                fAdjustTime = kAdjustDown;
            }
            else if (bufferUsageRatio >= 0.5) //OK to start thickening
            {
                fAdjustSize = kAdjustUp;
                fAdjustTime = kAdjustDown;
            }
            else if (bufferUsageRatio > 0.4) //OK to start thickening
            {
                fAdjustSize = kAdjustUp;
                fAdjustTime = kAdjustUp;
            }
            else if (bufferUsageRatio > 0.3) //need to speed up to avoid underflow; not enough
bandwidth
            {
                fAdjustSize = kNoChange;
                fAdjustTime = kAdjustUpUp;
            }
            else if (bufferUsageRatio > 0.2) //need to speed up and thin to avoid underflow; not
enough bandwidth
            {
                fAdjustSize = kAdjustDown;
                fAdjustTime = kAdjustUpUp;
            }
            else //below 20%  //need to speed up and thin to avoid underflow; not enough bandwidth
            {
                fAdjustSize = kAdjustDown;
        fAdjustTime = kAdjustUpUp;
            }
         }
    }

     if(fNumRTCPWithoutPacketLoss == 0) //RTCP have reported packet loss --> thinning
     {
         if (fCurRTT <= 10)
         {
             DEBUG_PRINTF(("RTPStream3GPP::UpdateTimeAndQuality fast network packet loss slowing
down fNumRTCPWithoutPacketLoss=%"_S32BITARG_"\n",fNumRTCPWithoutPacketLoss));
            fAdjustTime = kAdjustDown; //slow down could be random packet loss.
         }
         else
         {
            DEBUG_PRINTF(("RTPStream3GPP::UpdateTimeAndQuality slow network packet loss decrease
quality fNumRTCPWithoutPacketLoss=%"_S32BITARG_"\n",fNumRTCPWithoutPacketLoss));
            fAdjustSize = kAdjustDown; //most likely out of bandwidth so reduce quality.
            fAdjustTime = kAdjustUpUp; //don't let the buffer drain while reducing quality.
         }
     }
     else if (fNumRTCPWithoutPacketLoss <= 4) //If I get packet loss, then do not increase the rate
for 2 RTCP cycles
        fAdjustSize = MIN(kNoChange, fAdjustSize);
     else if (fNumRTCPWithoutPacketLoss > 10)
        fAdjustSize = kAdjustUp;
     fNumRTCPWithoutPacketLoss++;

     //Set the quality based on the thinning value
    if (fAdjustSize == kAdjustUp)  // increase bit rate gradually
             fRTPStream.SetQualityLevel(fRTPStream.GetQualityLevel() - 1);
    else if (fAdjustSize == kAdjustUpUp)
        fRTPStream.SetQualityLevel(fRTPStream.GetQualityLevel() - fRTPStream.GetQualityLevel()/3 -
1);
    else if (fAdjustSize == kAdjustDown) // thin down aggressively
        fRTPStream.HalveQualityLevel();

    SInt32 levelTest = ( (fRTPStream.GetNumQualityLevels() - fRTPStream.GetQualityLevel()) /2 )+ 1;

    DEBUG_PRINTF(("RTPStream3GPP::UpdateTimeAndQuality  update threshold=%ld\n", levelTest));

     //Adjust the maximum quality if the router is getting congested(1 consecutive large RTT ratios)
     if (fNumLargeRTT >= 4)
     {
        fNumLargeRTT = 0;      //separate consecutive maximum quality lowering by at least 1 RTCP
cycles.
        fRTPStream.SetMaxQualityLevelLimit(fRTPStream.GetMaxQualityLevelLimit() + 1);
        DEBUG_PRINTF(("RTPStream3GPP::UpdateTimeAndQuality fNumLargeRTT(%"_S32BITARG_") >= 4 maximum
quality level decreased: %"_S32BITARG_"\n",fNumLargeRTT, fRTPStream.GetMaxQualityLevelLimit()));
```

```
     }
     else if (fNumSmallRTT >= levelTest) //Router is not congested(4 consecutive small RTT ratios);
can start thickening.
     {
        fNumSmallRTT = 0;  //separate consecutive maximum quality raising by at least x RTCP cycles.
        fRTPStream.SetMaxQualityLevelLimit(fRTPStream.GetMaxQualityLevelLimit() - 1);
        DEBUG_PRINTF(("RTPStream3GPP::UpdateTimeAndQuality fNumSmallRTT (%"_S32BITARG_") >=
levelTest(%"_S32BITARG_") maximum quality level increased: %"_S32BITARG_"\n",fNumSmallRTT,
levelTest, fRTPStream.GetMaxQualityLevelLimit()));
     }

     char *payload = "?";
     if (GetDebugPrintfs())
     {
        UInt8 payloadType = fRTPStream.GetPayLoadType();
        if (qtssVideoPayloadType == payloadType)
                payload="video";
        else if (qtssAudioPayloadType == payloadType)
                payload="audio";
     }

     if (bufferDelay != kUInt32_Max)
        DEBUG_PRINTF((
                "RTPStream3GPP::UpdateTimeAndQuality type=%s quality=%"_S32BITARG_",
qualitylimit=%"_S32BITARG_", fAdjustTime=%i bufferUsage=%"_U32BITARG_"(%.0f%%), "
                "bufferDelay=%"_U32BITARG_"(%.0f%%)\n",       payload,fRTPStream.GetQualityLevel(),
fRTPStream.GetMaxQualityLevelLimit(),fAdjustTime,bufferUsage,
     bufferUsageRatio * 100,bufferDelay, bufferDelayRatio * 100
        ));
     else
        DEBUG_PRINTF((
                "RTPStream3GPP::UpdateTimeAndQuality type=%s quality=%"_S32BITARG_",
qualitylimit=%"_S32BITARG_", fAdjustTime=%i bufferUsage=%"_U32BITARG_"(%.0f%%), "
                "bufferDelay=?\n",payload,fRTPStream.GetQualityLevel(),
fRTPStream.GetMaxQualityLevelLimit(),fAdjustTime,bufferUsage, bufferUsageRatio * 100
        ));

}

//-------------------SendPlayResponse en RTPSession.h-------------------//

void            SendPlayResponse(RTSPRequestInterface* request, UInt32 inFlags, UInt32 activeTrack1,
UInt32 activeTrack2);

//-------------------SendPlayResponse en RTPSession.cpp-------------------//

//SendPlayResponse sends information about active tracks only
void RTPSession::SendPlayResponse(RTSPRequestInterface* request, UInt32 inFlags, UInt32
activeTrack1, UInt32 activeTrack2)
{
    QTSS_RTSPHeader theHeader = qtssRTPInfoHeader;
    RTPStream** theStream = NULL;
    UInt32* theTrackID = NULL;
    UInt32 theLen = 0, y = 1;
    UInt32 valueCount = this->GetNumValues(qtssCliSesStreamObjects);
    Bool16 lastValue = false;
    for (UInt32 x = 0; x < valueCount; x++)
    {
        this->GetValuePtr(qtssCliSesStreamObjects, x, (void**)&theStream, &theLen);
        Assert(theStream != NULL);
        Assert(theLen == sizeof(RTPStream*));

        if (*theStream != NULL)
        {
         QTSS_GetValuePtr(*theStream, qtssRTPStrTrackID, 0, (void**)&theTrackID, &theLen);
         if (*theTrackID==activeTrack1 || *theTrackID==activeTrack2)
         {
        if (y++ == 2)
        lastValue = true;
                (*theStream)->AppendRTPInfo(theHeader, request, inFlags, lastValue);
                theHeader = qtssSameAsLastHeader;
         }
        }
    }
    request->SendHeader();
}

//-------------------SendStandardRTSPResponse en QTSSCallbacks.h-------------------//
```

```
     // New variable parameters function
        static QTSS_Error   QTSS_SendStandardRTSPResponse(QTSS_RTSPRequestObject inRTSPRequest,
QTSS_Object inRTPInfo, UInt32 inFlags,...);

//-------------------SendStandardRTSPResponse en QTSSCallbacks.cpp-------------------//

//New QTSS_SendStandardRTSPResponse including additional parameters to indicate active tracks and
keep compatibility with older implementations
QTSS_Error QTSSCallbacks::QTSS_SendStandardRTSPResponse(QTSS_RTSPRequestObject inRTSPRequest,
                                                        QTSS_Object inRTPInfo,
                                                        UInt32 inFlags,...)
{
    if ((inRTSPRequest == NULL) || (inRTPInfo == NULL))
        return QTSS_BadArgument;

    switch (((RTSPRequestInterface*)inRTSPRequest)->GetMethod())
    {
        case qtssDescribeMethod:
            ((RTPSession*)inRTPInfo)->SendDescribeResponse((RTSPRequestInterface*)inRTSPRequest);
            return QTSS_NoErr;
        case qtssSetupMethod:
        {
  // Because QTSS_SendStandardRTSPResponse supports sending a proper 304 Not Modified on a SETUP,
but a caller typically won't be adding a stream for a 304 response, we have the policy of making the
caller pass in the QTSS_ClientSessionObject instead. That means we need to do different things here
depending...
            if (((RTSPRequestInterface*)inRTSPRequest)->GetStatus() == qtssRedirectNotModified)

(void)((RTPSession*)inRTPInfo)>DoSessionSetupResponse((RTSPRequestInterface*)inRTSPRequest);
            else
            {
                if (inFlags & qtssSetupRespDontWriteSSRC)
                    ((RTPStream*)inRTPInfo)->DisableSSRC();
                else
                    ((RTPStream*)inRTPInfo)->EnableSSRC();

                ((RTPStream*)inRTPInfo)->SendSetupResponse((RTSPRequestInterface*)inRTSPRequest);
            }

            return QTSS_NoErr;
        }
        case qtssPlayMethod:
       case qtssRecordMethod:
         //Here we add the two additional parameters to SendPlayResponse
         va_list param_pt;
         va_start(param_pt, inFlags);
            ((RTPSession*)inRTPInfo)->SendPlayResponse((RTSPRequestInterface*)inRTSPRequest,
inFlags, va_arg(param_pt, UInt32),va_arg(param_pt, UInt32));
         va_end (param_pt);
         return QTSS_NoErr;
         case qtssPauseMethod:
            ((RTPSession*)inRTPInfo)->SendPauseResponse((RTSPRequestInterface*)inRTSPRequest);
            return QTSS_NoErr;
         case qtssTeardownMethod:
            ((RTPSession*)inRTPInfo)->SendTeardownResponse((RTSPRequestInterface*)inRTSPRequest);
            return QTSS_NoErr;
         case qtssAnnounceMethod:
            ((RTPSession*)inRTPInfo)->SendAnnounceResponse((RTSPRequestInterface*)inRTSPRequest);
            return QTSS_NoErr;
    }
    return QTSS_BadArgument;
}

//-------------------Modulo SelectorDB: Selector.cpp-------------------//

#include <string.h>
#include <iostream>
#include <pqxx/pqxx>

using namespace std;
using namespace pqxx;

#include "Selector.h"
#include "QTSSModuleUtils.h"

#define SELECTOR_DEBUG 1 //Usefull to print debug information

// STATIC VARIABLES
static QTSS_ModulePrefsObject        sPrefs                 = NULL;
```

```
static QTSS_PrefsObject       sServerPrefs          = NULL;
static QTSS_ServerObject      sServer               = NULL;

// Default values for preferences
static Bool16                         sDefaultModuleEnabled   = true;
static Bool16                 sDefaultWapEnabled            = true;
char *                        sDefaultHiRes         = "352x240";
char *                        sDefaultConnString    = "hostaddr=147.83.39.46 dbname=profilev2
user=root password=futurlink";

// Current values for preferences
static Bool16                 sModuleEnabled        = true;
static Bool16                 sWapEnabled           = true;
string                        sHiRes                = "352x240";
string                        sConnString           = "hostaddr=147.83.39.46 dbname=profilev2
user=root password=futurlink";

// FUNCTION PROTOTYPES
static QTSS_Error   SelectorDispatch(QTSS_Role inRole, QTSS_RoleParamPtr inParamBlock);
static QTSS_Error   Register(QTSS_Register_Params* inParams);
static QTSS_Error   Initialize(QTSS_Initialize_Params* inParams);
static QTSS_Error   RereadPrefs();
static void         ProcessRTSPRequest(QTSS_StandardRTSP_Params* inParams);

QTSS_Error Selector_Main(void* inPrivateArgs)
{
    return _stublibrary_main(inPrivateArgs, SelectorDispatch);
}

QTSS_Error  SelectorDispatch(QTSS_Role inRole, QTSS_RoleParamPtr inParamBlock)
{
    switch (inRole)
    {
        case QTSS_Register_Role:
            return Register(&inParamBlock->regParams);
        case QTSS_Initialize_Role:
            return Initialize(&inParamBlock->initParams);
        case QTSS_RTSPRoute_Role:
         ProcessRTSPRequest(&inParamBlock->rtspRequestParams);
        case QTSS_RereadPrefs_Role:
            return RereadPrefs();
    }
    return QTSS_NoErr;
}

QTSS_Error Register(QTSS_Register_Params* inParams)
{
    //Set the Roles
    (void)QTSS_AddRole(QTSS_Initialize_Role);
    (void)QTSS_AddRole(QTSS_RereadPrefs_Role);
    (void)QTSS_AddRole(QTSS_RTSPRoute_Role);

    // Tell the server our name!
    static char* sModuleName = "Selector";
    ::strcpy(inParams->outModuleName, sModuleName);

    return QTSS_NoErr;
}

QTSS_Error Initialize(QTSS_Initialize_Params* inParams)
{
    QTSSModuleUtils::Initialize(inParams->inMessages, inParams->inServer, inParams-
>inErrorLogStream);
    sServer = inParams->inServer;
    sServerPrefs = inParams->inPrefs;
    sPrefs = QTSSModuleUtils::GetModulePrefsObject(inParams->inModule);
    return RereadPrefs();
}

QTSS_Error RereadPrefs()
{

    QTSSModuleUtils::GetAttribute(sPrefs, "selector_enabled",  qtssAttrDataTypeBool16,
            &sModuleEnabled, &sDefaultModuleEnabled, sizeof(sDefaultModuleEnabled));

    sHiRes = QTSSModuleUtils::GetStringAttribute(sPrefs, "hi_res", sDefaultHiRes);
    sConnString = QTSSModuleUtils::GetStringAttribute(sPrefs, "ConnString", sDefaultConnString);
    return QTSS_NoErr;
}
```

```
void ProcessRTSPRequest(QTSS_StandardRTSP_Params* inParamBlock)
{
    if (!sModuleEnabled) //If the module is disabled, just exit
    return;

    QTSS_RTSPMethod* theMethod = NULL;
    QTSS_Error theErr;
    UInt32 theLen = 0;
    QTSS_GetValuePtr(inParamBlock->inRTSPRequest, qtssRTSPReqMethod, 0, (void**)&theMethod,
&theLen);

    switch (*theMethod)
    {
     case qtssDescribeMethod: //Just process Describe requests
        {

        if(!sWapEnabled) //Check the method from preferences
        return;

         char* thePath = NULL;
        char* theAddress = NULL;
        char* thePtr = NULL;
        Bool16 CliHiRes = false;
        Bool16 CliMP4 = false;
        Bool16 CliAAC = false;
        Bool16 CliH263 = false;
        Bool16 Vendor = false;

        // Get the client IP address in dotted decimal as UInt32
        (void)QTSS_GetValueAsString(inParamBlock->inRTSPSession, qtssRTSPSesRemoteAddrStr, 0,
(char**)&theAddress);

        // Get the request "/file.ext" and assert it is not NULL
        (void)QTSS_GetValueAsString(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0,
(char**)&thePath);

        if((strstr(thePath,".mp4")==NULL) && (strstr(thePath,".3gp")==NULL))
        return;

        thePtr = strstr(thePath,"_"); //the file has _ character
        if(thePath!= NULL)
                {
                try
                        {
                        connection C(sConnString.c_str());
                        if (SELECTOR_DEBUG==1)
                        cout << "Connected to " << C.dbname() << endl;

                        work W(C);

                        if (SELECTOR_DEBUG==1)
                        cout << ("SELECT id FROM summary WHERE id_dev='"+ (PGSTD::string)theAddress
                        +"'") << endl;

                        result R = W.exec("SELECT id FROM summary WHERE id_dev='"+
                        (PGSTD::string)theAddress +"'");
                        if (R.size()>0)        //The IP is included in the Database
                        {
                                if (SELECTOR_DEBUG==1)
                                cout << "Found " << R.size() << " ids:" << endl;
                                result::const_iterator r = R.begin();
                                result::const_iterator s = R.begin();
                                if (SELECTOR_DEBUG==1)
                                cout << r[0].c_str() << endl;

                                // Searching ScreenSize information

                                result R = W.exec("SELECT value FROM attribute WHERE id_summary='"+
                                (PGSTD::string)s[0].c_str() +"' and attr_name='ScreenSize'");

                                if (SELECTOR_DEBUG==1)
                                cout << "Found " << R.size() << " values:" << endl;

                                if (R.size()>0) //We found the screen resolution value
                                {
                                        r = R.begin();
                                        if (SELECTOR_DEBUG==1)
                                        cout << "comparando" << sHiRes << "con"<< r[0].c_str() <<
```

```
        endl;

        if ((sHiRes.compare(r[0].c_str())<0)
        ||((strcspn(r[0].c_str(),"x") > strcspn(sHiRes.c_str(),"x"))))
        {
        qtss_printf("HiRes\n");
        CliHiRes=true;
        }
}

// Searching MP4 compatibility
R = W.exec("SELECT value FROM attribute WHERE id_summary='"+
(PGSTD::string)s[0].c_str() +"' and attr_name='ThreeGPAccept'");

if (SELECTOR_DEBUG==1)
cout << "ThreeGP: Found " << R.size() << " values:" << endl;

if (R.size()>0) //We found an answer
{
        r = R.begin();
        cout << r[0].c_str() << endl;
        if(strstr(r[0].c_str(),"video/mp4")!=NULL) CliMP4=true;
        //video/mp4 appears in the supported MIME list
        if(strstr(r[0].c_str(),"audio/AAC")!=NULL ||
        strstr(r[0].c_str(),"audio/aac")!=NULL) CliAAC=true;
        //audio/aac appears in the supported MIME list
        if(strstr(r[0].c_str(),"video/H263")!=NULL) CliH263=true;
        //video/mp4 appears in the supported MIME list
}

R = W.exec("SELECT value FROM attribute WHERE id_summary='"+
(PGSTD::string)s[0].c_str() +"' and attr_name='CcppAccept'");

if (SELECTOR_DEBUG==1)
cout << "Found " << R.size() << " values:" << endl;

if (R.size()>0) //We found an answer
{
        r = R.begin();
        cout << r[0].c_str() << endl;
        if(strstr(r[0].c_str(),"video/mp4")!=NULL) CliMP4=true;
        //video/mp4 appears in the supported MIME list
        if(strstr(r[0].c_str(),"audio/AAC" )!=NULL ||
        strstr(r[0].c_str(),"audio/aac")!=NULL) CliAAC=true;
        //audio/aac appears in the supported MIME list
        if(strstr(r[0].c_str(),"video/H263")!=NULL) CliH263=true;
        //video/mp4 appears in the supported MIME list
}

R = W.exec("SELECT value FROM attribute WHERE id_summary='"+
(PGSTD::string)s[0].c_str() +"' and attr_name='StreamingAccept'");

if (SELECTOR_DEBUG==1)
cout << "Found " << R.size() << " values:" << endl;

if (R.size()>0) //We found an answer
{
        r = R.begin();
        cout << r[0].c_str() << endl;
        if(strstr(r[0].c_str(),"video/mp4")!=NULL) CliMP4=true;
        //video/mp4 appears in the supported MIME list
        if(strstr(r[0].c_str(),"audio/AAC")!=NULL ||
        strstr(r[0].c_str(),"audio/aac")!=NULL) CliAAC=true;
        //audio/aac appears in the supported MIME list
        if(strstr(r[0].c_str(),"video/H263")!=NULL) CliH263=true;
        //video/mp4 appears in the supported MIME list
}

R = W.exec("SELECT value FROM attribute WHERE id_summary='"+
(PGSTD::string)s[0].c_str() +"' and attr_name='Vendor'");
if (R.size()>0) //We found an answer
{
cout << "Vendor" <<endl;
if(strstr(r[0].c_str(),"Nokia")!=NULL)
{
qtss_printf("Vendor = NOKIA\n");
Vendor=true;
}
}
```

```
                                if (thePtr!=NULL && CliAAC==true && CliHiRes==true)
                                {
                                        cout << "HI" << endl;
                                        *thePtr++='_';
                                        *thePtr++='h';
                                        *thePtr++='i';
                                }
                                else if (thePtr!=NULL && CliAAC==true && Vendor==true &&
                                CliHiRes==false)
                                {
                                        cout << "ME" << endl;
                                        *thePtr++='_';
                                        *thePtr++='m';
                                        *thePtr++='e';
                                }
                                else
                                {
                                        cout << "LO" << endl;
                                        *thePtr++='_';
                                        *thePtr++='l';
                                        *thePtr++='o';
                                }
                                if (thePtr!=NULL && CliMP4==true)
                                {
                                        cout << "MP4" << endl;
                                        *thePtr++='.';
                                        *thePtr++='m';
                                        *thePtr++='p';
                                        *thePtr++='4';
                                }
                        theErr = QTSS_SetValue(inParamBlock->inRTSPRequest, qtssRTSPReqFilePath, 0,
thePath, strlen(thePath));
                        qtss_printf("The path: %s\n", thePath);
                        }
                        }
                catch (const exception &e)
                {
                        cerr << e.what() << endl;
                        return;
                }
        }

        break;}
      default:
                break;}
return;
}

//-------------------Modulo SelectorDB: Selector.h-------------------/
/*
        File:       Selector.h
        Contains:   Headers for Selector.cpp
*/

#ifndef __SELECTOR_H__
#define __SELECTOR_H__
#include "QTSS.h"

extern "C"
{
    QTSS_Error Selector_Main(void* inPrivateArgs);
}

#endif
```