Escola Politècnica Superior de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO DE FIN DE CARRERA

TÍTULO: Multiple Description Coding (MDC) para la transmisión de vídeo de alta calidad

TITULACIÓN: Ingeniería Técnica de Telecomunicaciones, especialidad Telemática

AUTOR: Jordi García Garrido

DIRECTOR: Toni Oller Arcas

FECHA: 14 de octubre de 2008

Título: Multiple Description Coding (MDC) para la transmisión de vídeo de alta calidad

Autor: Jordi García Garrido

Director: Toni Oller Arcas

Fecha: 14 de octubre de 2008

Resumen

El uso de las redes “Peer to Peer”, conocidas como P2P, ha sufrido un gran crecimiento en los últimos años. Desde sus primeros usos para intercambiar archivos, su uso se ha ido extendiendo paulatinamente hacia otros servicios. Uno de ellos es la transmisión de video en streaming.

En este sentido, las aplicaciones existentes hasta este momento, trataban a todos los Peers igual, es decir, existía un flujo de video, un peer se conectaba a otros Peers y lo comenzaba a reproducir a la vez que transmitir a otros Peers. Todos recibían el mismo flujo de video y debían tener unos requisitos mínimos de ancho de banda y capacidad de procesamiento.

Este TFC trata de analizar los diferentes mecanismos que existen para crear una red de transmisión de video en streaming por una red P2P en la que sus Peers son heterogéneos, es decir, cada uno tiene unas características (ancho de banda, CPU) y dependiendo de ellas recibirá el video con más o menos calidad.

Además de analizar las diferentes herramientas, se pretende implementar un prototipo como prueba de concepto para la herramienta MDC, Multiple Description Coding.

Title: Multiple Description Coding (MDC) for the high quality video transmission

Author: Jordi García Garrido

Director: Toni Oller Arcas

Date: 14 de octubre de 2008

Overview

The use of "Peer to Peer" nets, known as P2P, has suffered a great growth on last years. From its first applications to interchange files, its use has been extended gradually to other services. One of them is the video streaming transmission.

In this sense, the existing applications until know, treated all Peers as equal, that is, a video flux existed, a peer was connected to other Peers and started to play while also transmitting to other Peers. All of them received the same video flux and must have minimum requirements of bandwidth and capacity processing.

This TFC treats to analyze the different mechanisms that exist to create a video streaming net transmission by a P2P net in which their Peers are heterogeneous, I mean, everyone has its own characteristics and depending on them the video should be received with more or less quality.

As well as analyzing the different tools, pretending to implement a prototype as a proof of the concept for the tool MDC, Multiple Description Coding, is one of the aims of this essay.

AGRADECIMIENTOS

Son muchísimas las personas a las que debo agradecer haber podido realizar este proyecto, así que intentaré ser breve:

En primer lugar, por confiar en mí para la realización del TFC y su apoyo y orientación durante el mismo, al director del proyecto, Toni Oller, para quien este TFC ha sido como traer un niño al mundo. Espero que sea muy feliz.

A todos y cada uno de los miembros del MediaCat por su compañerismo en todas esas horas que hemos pasado juntos y en concreto a los miembros del proyecto Trilogy y a mis vecinos de mesa por su inestimable ayuda. Mención aparte merece Tonino por su enorme paciencia conmigo en mis interminables dudas, gracias, de verdad.

Por supuesto a Rosa, sin la cual estoy perdido, por todos los ánimos, el cariño y la fuerza que me ha dado siempre, y por estar siempre ahí.

A toda mi familia, que jamás dudaron de que podría acabar esta carrera y fueron mi apoyo en los momentos más difíciles. Sin ellos, no habría podido terminar.

Y finalmente, a todos los amigos, en especial a Kike, Xavi, Edu, Maci, Torron, y otros muchos, que han hecho que esta etapa que justo ahora acaba no sea una simple etapa de formación. Son muchísimos los recuerdos y las vivencias y las guardo todas.

A todos, gracias.

ÍNDEX

INTRODUCCIÓN

1. Redes P2P

La rapidísima proliferación de las redes P2P (Peer to Peer) en los últimos años ha provocado que éstas evolucionen a una velocidad realmente vertiginosa. En 1996 se desarrolló la primera red P2P (Hotline Connect), que servía para intercambiar archivos entre sus clientes. A partir de entonces han aparecido numerosas aplicaciones que se basan en estas redes, desde las famosas "Napster, Emule, BitTorrent o Kazaa" para el intercambio de todo tipo de ficheros, hasta "Skype" para la telefonía IP o "Coolstreaming o TVAnts" para la transmisión de video (Streaming). Estas últimas son las que vamos a tratar en este TFC.

2. Streaming

2.1 Ventajas e inconvenientes

El uso de estas redes P2P para distribuir video en una red IP (Internet) tiene unas ventajas muy claras, aunque de todas ellas sobresale una muy visiblemente: el ahorro de ancho de banda que utiliza el servidor del video, ya que no tiene que distribuir el video a todos los clientes (peers), sino que lo distribuye a un número concreto de ellos y cada peer que recibe el video lo distribuye a otros. Esta arquitectura, sin embargo, tiene un gran inconveniente: la complejidad del sistema, ya que resulta más sencillo conectarse a un servidor y recibir el video, que buscar peers y recibir un "pedazo" del video de cada uno. Además hay que tener en cuenta que existen 2 tipos de streaming, "on demand" y "live", lo que significa que todos los peers tienen que ver el mismo instante del video por lo que hay que añadir una componente de complejidad que es la sincronización.

2.2 Historia y evolución

Existen varios programas que funcionan como "televisiones P2P", que conectan con varios canales, de los cuales el más extendido quizás sea el "Coolstreaming". Estos programas han aparecido bastante recientemente, y todos tienen una estructura muy similar: un servidor coge un video, lo trocea en paquetes y lo ofrece a la red, luego un peer recibe este video, lo recompone y lo puede reproducir. Este proceso es sencillo y tiene unas características inherentes: si un paquete no llega, no se puede reproducir ese pedazo del video y, a parte, el peer ve el video a la calidad a la que lo emite el servidor, por lo que se puede decir que se trata a todos los peers por igual. Ahora bien, en la práctica esto no es así, ya que cada vez aparecen más dispositivos capaces de conectarse a Internet, desde un terminal móvil hasta una televisión de alta definición a través de Apple TV (p.ej.). Estos dos dispositivos tienen unas diferencias abismales. Mientras que el terminal móvil no necesita mucha

calidad y suele estar asociado a un bajo ancho de banda, la televisión de alta definición necesita una gran calidad del video y puede tener asociado bastante ancho de banda. Para solventar este problema existen varios mecanismos, como el Layered Coding (LC) o Múltiple Description Coding (MDC) [1] . Éstos lo que hacen es dividir el video en “capas” y distribuirlas por separado por la red y el peer reproducirá el video con más o menos calidad según el número de capas que reciba. Pero estos dos mecanismos tienen diferencias sustanciales. El LC divide el video en diferentes tipos de capas: una base y una o más de información extra. El peer necesita la capa base para reproducir el video a su mínima calidad y añadiendo capas de información extra gana calidad. Si se pierde un paquete de la capa base, el video no se puede reproducir durante ese tiempo. En cambio, el MDC divide el video en n capas iguales. Sólo hace falta una (da igual cual, puede ser cualquiera) para reproducir el video a su mínima calidad y se gana en calidad recibiendo más capas, hasta la calidad máxima que se consigue recibiendo las n capas. Si se pierde un paquete, perderemos calidad en el video durante ese instante.

Existen otros problemas que afectan al correcto funcionamiento de estos sistemas, por ejemplo el free-riding o la alta rotación de peers. El free-riding es que un peer recibe pero no envía información. Este problema afecta por igual a MDC y a LC ya que lo que hacen es “parasitar” el sistema utilizando ancho de banda y sin ofrecer ninguno. La alta rotación de peers, en cambio, puede afectar de diferente manera a los dos sistemas. En LC, si hay un peer A que está recibiendo la capa base de un peer B y éste se desconecta, deberá encontrar otro peer que tenga la capa base para poder reproducir el video. Y la capa base no es problema, porque todos los peers tienen que tenerla para poder reproducir el video, aunque mientras que se busca un nuevo peer, no se reproducirá video, pero con las capas extras puede haber más problemas para encontrarlas. En MDC el problema es similar, ya que si se deja de recibir una capa, hay que volver a encontrarla para reproducir el video con la misma calidad que lo estábamos haciendo, pero no existe el momento crítico de perder la reproducción total del video, a no ser que se pierdan todas las capas.

Es sobre este último mecanismo (MDC) sobre el que se va a basar el TFC.Existen varios programas que funcionan como “televisiones P2P”, que conectan con varios canales, de los cuales el más extendido quizás sea el “Coolstreaming”. Estos programas han aparecido bastante recientemente, y todos tienen una estructura muy similar: un servidor coge un video, lo trocea en paquetes y lo ofrece a la red, luego un peer recibe este video, lo recompone y lo puede reproducir. Este proceso es sencillo y tiene unas características inherentes: si un paquete no llega, no se puede reproducir ese pedazo del video y, a parte, el peer ve el video a la calidad a la que lo emite el servidor, por lo que se puede decir que se trata a todos los peers por igual. Ahora bien, en la práctica esto no es así, ya que cada vez aparecen más dispositivos capaces de conectarse a Internet, desde un terminal móvil hasta una televisión de alta definición a través de Apple TV (p.ej.). Estos dos dispositivos tienen unas diferencias abismales. Mientras que el terminal móvil no necesita mucha calidad y suele estar asociado a un bajo ancho de banda, la televisión de alta definición necesita una gran calidad del video y puede tener asociado bastante ancho de banda. Para solventar este problema existen varios mecanismos,

como el Layered Coding (LC) o Múltiple Description Coding (MDC). Éstos lo que hacen es dividir el video en “capas” y distribuirlas por separado por la red y el peer reproducirá el video con más o menos calidad según el número de capas que reciba. Pero estos dos mecanismos tienen diferencias sustanciales. El LC divide el video en diferentes tipos de capas: una base y una o más de información extra. El peer necesita la capa base para reproducir el video a su mínima calidad y añadiendo capas de información extra gana calidad. Si se pierde un paquete de la capa base, el video no se puede reproducir durante ese tiempo. En cambio, el MDC divide el video en n capas iguales. Sólo hace falta una (da igual cual, puede ser cualquiera) para reproducir el video a su mínima calidad y se gana en calidad recibiendo más capas, hasta la calidad máxima que se consigue recibiendo las n capas. Si se pierde un paquete, perderemos calidad en el video durante ese instante.

Existen otros problemas que afectan al correcto funcionamiento de estos sistemas, por ejemplo el free-riding o la alta rotación de peers. El free-riding se da en los casos que un peer recibe pero no envía información. Este problema afecta por igual a MDC y a LC ya que lo que hacen es “parasitar” el sistema utilizando ancho de banda y sin ofrecer ninguno. La alta rotación de peers, en cambio, puede afectar de diferente manera a los dos sistemas. En LC, si hay un peer A que está recibiendo la capa base de un peer B y éste se desconecta, deberá encontrar otro peer que tenga la capa base para poder reproducir el video. Y la capa base no es problema, porque todos los peers tienen que tenerla para poder reproducir el video, aunque mientras que se busca un nuevo peer, no se reproducirá video, pero con las capas extras puede haber más problemas para encontrarlas. En MDC el problema es similar, ya que si se deja de recibir una capa, hay que volver a encontrarla para reproducir el video con la misma calidad que lo estábamos haciendo, pero no existe el momento crítico de perder la reproducción total del video, a no ser que se pierdan todas las capas.

Es sobre este último mecanismo (MDC) sobre el que se va a basar el TFC.

Esta memoria tiene una estructura de 6 capítulos más uno de Conclusiones. El primero introduce unos conceptos básicos sobre la temática del proyecto. El segundo plantea el diseño de la solución. En el tercero se explica la implementación de la herramienta, así como una explicación de cada módulo. En el cuarto se hace énfasis en la implementación pero desde el nivel de la codificación. En el quinto capítulo se muestra el funcionamiento de la aplicación y en el sexto y último se comenta la planificación que ha tenido la totalidad del proyecto.

CAPÍTULO 1. CONCEPTOS BÁSICOS

Este TFC trata de crear una herramienta para la transmisión de vídeo en una red P2P de usuarios heterogéneos. Para ello, se va a utilizar una técnica llamada MDC (Multiple Description Coding). Esta herramienta trata el video de manera que, unido a una red P2P orientada al streaming de vídeo, permite crear un sistema en el que sus usuarios pueden acceder a los contenidos desde clientes con diferentes características, tanto técnicas, desde un móvil hasta una pantalla HD, como de ancho de banda. Pero antes de entrar a la fase de diseño hay varios conceptos que deben ser clarificados para que luego sean tratados.

1.1. YUV

El YUV[2][3] es una representación del color que divide cada imagen en tres capas o componentes: una de luminancia (Y) y dos de crominancia (U, V). Esta representación del color es más fiel a la que percibe el ojo humano que la clásica RGB, que es la utilizada en las pantallas de ordenador, por ejemplo.

Fig 1.1 Componentes de imagen YUV.

Fig 1.2 Imagen resultante al combinar los 3 componentes.

El tamaño relativo de los planos Y, U y V no es siempre el mismo y para indicarlo, se hace con tres números separados por dos puntos. (Y:U:V). El YUV 4:4:4, por ejemplo, quiere decir que por cada 4 píxeles Y, hay 4 U y 4 V, o en el 4:2:2, por cada 4 píxeles Y hay 2 U y 2 V. Debido a que está comprobado que nuestro ojo es más sensible a la luminancia que a la crominancia, (de noche, por ejemplo, podemos ver la figura de las cosas, pero no el color), los planos U y V pueden ser más pequeños sin que el ojo lo note. Esa es la razón por la cual el formato más extensamente utilizado es el YUV 4:2:0. Su nombre puede llevar a engaño, porque en realidad se trata de un 4:1:1, pero existe un problema: hay dos formatos 4:1:1, y a uno se le conoce como 4:1:1 y al otro como 4:2:0. En los dos casos cada píxel U y V corresponde a 4 píxeles Y, pero se diferencian el uno del otro en la forma que tienen. Mientras que en el 4:2:0 los píxeles U y V son cuadrados, en el 4:1:1 son rectangulares.

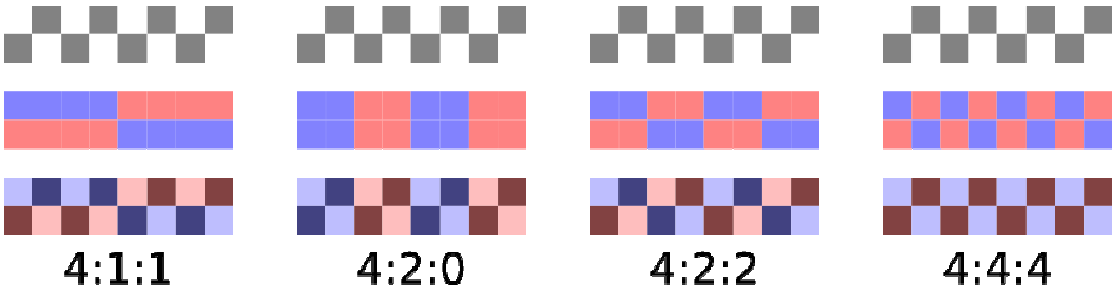

Fig 1.3 Muestreo de los diferentes formatos de YUV.

Informáticamente el formato YUV se representa en un archivo *raw* (crudo, sin cabeceras), con los valores Y, U y V de cada fotograma del vídeo.

Tabla 1.1 Tamaño archivos

Nombre	Resolución	Tamaño frame	Tamaño 1 minuto
QCIF	176x144	38KB	55MB
CIF	352x288	152KB	218MB
PAL (efectivas)	720x576	622KB	895MB
HD Ready	1280x720	1,38MB	2GB
Full HD	1920x1080	3,11MB	4,5GB

1.2. Multiplexado

Multiple Descriptor Coding consiste en dividir un flujo original en varios sublujos más pequeños. Esta división es lo que llamaríamos hacer un multiplexado. Y este multiplexado puede ser tanto temporal como espacial. En el temporal, lo que obtenemos es un número de flujos de la misma resolución que el original pero más cortos. Esto se consigue separando el video por frames, es decir, el primer frame al primer flujo, el segundo frame al segundo flujo,.... En el multiplexado espacial lo que obtenemos el mismo número de flujos de la misma duración que el original, pero de una resolución menor. Esto se consigue separando los flujos por píxeles.

Si tenemos en cuenta el número de flujos, vemos que puede existir un número fijo de descriptores, o un número variable. Si hay un número fijo, se puede deber a 2 motivos: a que se fije el número de descriptores o a que se fije el tamaño de éstos. Si fijamos el número de descriptores *n*, a la salida habrá *n* descriptores sea cual sea la resolución del video de entrada. Los valores típicos de la *n* son: 2, 4, 8, 16,..., aunque los más usados son 2, 4 y 8, ya que valores más altos generan una carga computacional demasiado elevada. En cambio, si fijamos la resolución de los descriptores, a la salida habrá un número fijo de descriptores, pero este número depende de la resolución del video de entrada. Es decir, si fijamos la resolución del descriptor a 5x5 y tenemos un vídeo a la entrada que tiene una resolución de 10x10, obtendremos 4 descriptores, pero si el video es de 20x20, obtendremos 16.

Existe otra manera mucho más dinámica y adaptable al medio en la que el número de descriptores es variable. Esto se puede implementar creando un canal de retorno al splitter en el que se tenga información de los peers y dependiendo de esa información crear más descriptores, menos, más grandes, más pequeños,..., dependiendo de los peers. Una posible implementación sería que si solo existe un peer y tiene ancho de banda suficiente, podríamos dejar de crear descriptores, porque sería una carga innecesaria tanto para el emisor (al hacer el split) como para el receptor (al hacer el merg). En cambio, si tenemos varios descriptores con un ancho de banda muy pequeño, podríamos pasar a crear descriptores más pequeños para que el usuario pueda reproducir un video aunque sea a más baja calidad.

Además, se puede configurar la relación del tamaño de los descriptores entre sí. Hasta ahora hemos supuesto que todos los descriptores eran iguales entre sí, lo que se llama “MDC Balanceado”, pero tenemos la opción de crear unos descriptores más grandes y otros más pequeños, lo que llamaríamos “MDC No Balanceado”. De esta manera, un determinado peer con unas características en concreto podría pedir directamente el descriptor que más le convenga, y si es necesario y tiene capacidad para ello, pedir los otros para aumentar la calidad del video recibido. Aunque tiene sus ventajas, se puede considerar que esta técnica crea unos descriptores más importantes que otros, igual que el LC, donde existe un flujo base y otros de mejora de calidad.

CAPÍTULO 2. DESARROLLO TEÓRICO

Una vez explicados los conceptos básicos, y de la principal premisa, la creación de una aplicación Web capaz de hacer streaming por una red P2P utilizando Multiple Description Coding, el próximo paso es explicar la prueba realiza. En este capítulo se explica a nivel teórico cual es dicha solución.

2.1. Aplicación MDC

Esta herramienta consta de diferentes partes, que básicamente se pueden agrupar en 2: el emisor y el receptor. El emisor transforma un flujo de video en n flujos y el receptor recibe m flujos y los transforma en uno solo. El primero, a su vez, consta de otros subprocesos: fuente, *splitter,* codificación y transmisión. El receptor es básicamente el proceso inverso: recepción, descodificación, *merger* y reproducción.

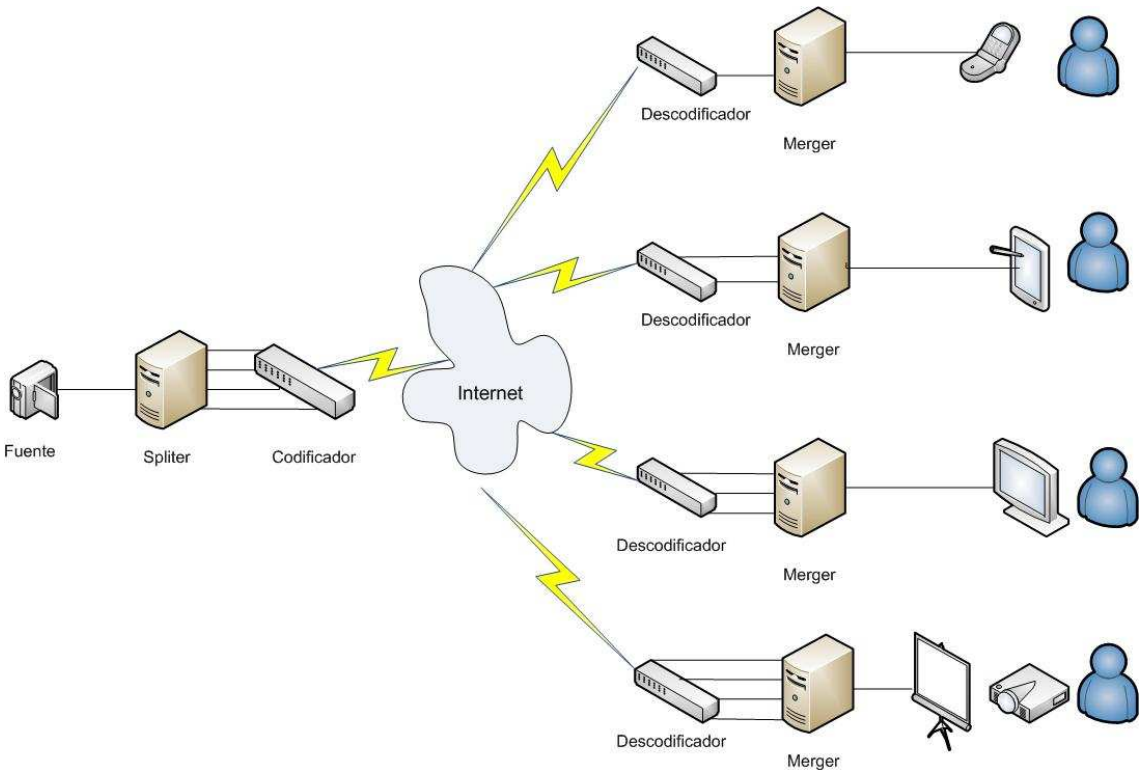

Fig 2.1 Esquema aplicación MDC

Esto, en lo referente a la aplicación del MDC propiamente dicha, sin tener en cuenta la red P2P. Para que pueda funcionar sobre dicha red hay que incorporar algún subproceso más, los propios de este tipo de redes: conexión, búsqueda de contenidos o anuncio, entre otros.

2.2. Transmisor

2.2.1. Fuente de vídeo

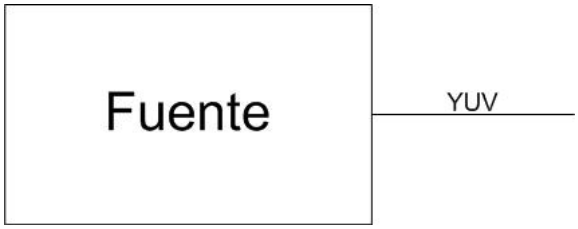

Fig 2.2 Esquema fuente

El primer paso del transmisor es el llamado *Splitter.* En esta fase, que se explica en el apartado 2.2.2, el video se divide en cuatro para su posterior transmisión. La técnica que se va a utilizar para ello es la conocida como *Polyphase DownSampling,* que divide el video separando los píxeles por filas y columnas y pares e impares. Para hacer esto es estrictamente necesario poder tratar el video a nivel de píxel, por lo que tenemos que utilizar, lógicamente, un formato que trate directamente con píxeles. Queda descartada, pues, cualquier tipo de codificación. Existen varias opciones, pero en este TFC se va a utilizar el formato YUV, explicado en el apartado 1.1.

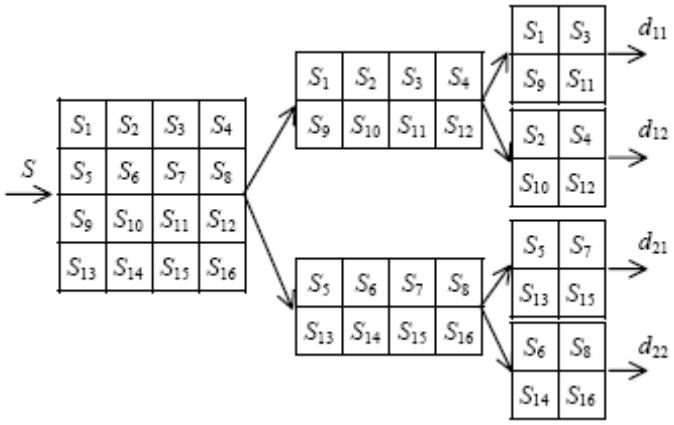

Fig 2.3 Ejemplo de Polyphase DownSampling[10]

En este TFC se ha escogido el YUV 4:2:0 ya que es el mayoritariamente utilizado. Por ejemplo, los DVD tienen una codificación MPEG-2, pero la fuente es YUV 4:2:0. Informáticamente se va a tratar con el tipo de archivo “yuv420”. En este caso, una imagen de 2x2 píxeles ocuparía 6 bytes. A nivel computacional lo óptimo sería utilizar un archivo de vídeo YUV directamente, ya que podríamos acceder a cada píxel sin ninguna acción previa, pero el problema es el tamaño. Tomando como referencia la tabla 1.1 vemos que almacenar un archivo YUV en disco sería altamente desaconsejable, sin embargo existen cámaras que ofrecen a la salida este formato, por lo que son una posibilidad muy a tener en cuenta como fuente de esta herramienta MDC. Por tanto, tenemos 2 opciones para utilizar como fuente: una cámara que

ofrezca salida YUV o un archivo en disco codificado, por lo que habría que añadir una fase previa de “descodificación” para obtener el flujo YUV. En este TFC, sin embargo, vamos a tratar directamente con archivos YUV de diferentes resoluciones y duraciones para simplificar el proceso.

2.2.2. Splitter

Ésta es la fase en la que un flujo de video se convierte en varios subflujos que llamaremos descriptores. En nuestra herramienta pretendemos que un dispositivo de baja resolución (móvil, PDA;...) pueda acceder a la red y reproducir el video. Para ello no necesita un video de alta resolución y que se reproduzca a saltos (multiplexado temporal), si no que necesita un flujo de resolución más pequeña, pero que tenga una reproducción continuada (multiplexado espacial). Es por eso que en este TFC se va a utilizar este último multiplexado.

Fig 2.4 Esquema Splitter

En cuanto al MDC balanceado o no balanceado, el MDC no balanceado tiene sus ventajas, pero merma a esta herramienta en cuanto a su dinamismo a la hora de recibir cualquier flujo, ya que si no recibimos la capa en concreto se baja considerablemente de calidad de video. Es por esta razón que en este TFC vamos a trabajar con un MDC Balanceado con multiplexado espacial y en concreto con 4 descriptores.

2.2.3. Codificación

Fig 2.5 Esquema codificador

Del proceso de Splitter obtenemos 4 flujos de video YUV. El siguiente paso sería transmitirlos por la red, pero ya hemos visto en la Tabla 1.1 el enorme

tamaño de estos archivos. El volumen de tráfico que se generaría, muy por encima de lo que son capaces de transmitir las redes convencionales, hace que transmitir YUV directamente sea completamente inviable a día de hoy, por lo que es necesario reducir esa cantidad de alguna manera. Lo que vamos a hacer es codificar el video con algún códec. Existen varios codificadores que pueden hacer esto (ffmpeg[7], mencoder, vlan,...), pero para este sistema necesitamos que el que se use tenga unas características bastante concretas, por ejemplo que tenga soporte para el mayor número posible de códecs, que tenga poco consumo de RAM, ya que en el emisor vamos a ejecutar, al menos, 4 en paralelo, o que sea capaz de tratar con flujos ya que es nuestra "unidad de trabajo", por llamarlo así. Es decir, no trabajamos con archivos de video, si no con el flujo del video, al vuelo. Además, existen otro tipo de características, no técnicas, sino de uso, que aunque no son determinantes son recomendables, como que tenga un uso sencillo, o que exista suficiente documentación a la que poder acceder en caso de duda.

Este proceso de codificación es realmente importante para la viabilidad del sistema, ya que reduce el tamaño de los datos hasta en un factor de 50.

2.2.4. Transmisión

Este es el último paso del transmisor. En este proceso, obtenemos un flujo de vídeo del codificador, en realidad 4, pero vamos a tratar todos los flujos de la misma manera, así que hablaremos de 1. Este flujo tiene que ser transmitido por la red y para ello necesitamos dos cosas. La primera, lógicamente, necesitamos saber a dónde hay que enviarlo, pero esto es tarea de la red P2P que se explica más adelante. Y la segunda, necesitamos adecuar el video para su transmisión por una red IP. Es decir, hay que trocear el video en paquetes IP. Para transmitir el video tenemos varios protocolos de transporte: UDP, TCP o RTP. TCP, está descartado para aplicaciones multimedia en tiempo real por varios motivos: es más complejo, por tanto tiene más carga computacional, lo que podría perjudicar la reproducción del video. Sus cabeceras son mayores, añadiendo tráfico a la transmisión y en caso de congestión de la red, puede "ayudar" a empeorarla debido a sus retransmisiones. En cambio UDP y RTP no tienen estos problemas. Son más ligeros y no retransmiten en caso de pérdida. En servicios multimedia se puede tolerar hasta un 5% de pérdidas. Además, el RTP (Realtime Transfer Protocol), es un protocolo creado específicamente para ello, por lo que a priori lo convierte en un candidato ideal. Sin embargo existe un problema: la inmensa mayoría de SO actuales vienen con el TCP y el UDP implementadas en su kernel, por lo que su uso es transparente, pero no para RTP, por lo que habría que añadir el soporte para cada SO.

2.3. Receptor

Si al transmisor le diésemos la vuelta como un calcetín obtendríamos el receptor. Con algunas pequeñas diferencias, los subprocesos del receptor son los inversos al del transmisor. Pasamos a explicarlos a continuación.

2.3.1. Recepción de flujos

Lo primero que debe hacer el receptor es recibir los flujos. Para ello, a través de los módulos de la red P2P ha anunciado los puertos a través de los cuales va a recibir el video, por lo tanto hay que abrir esos puertos y ponerlos a escuchar. De esos puertos vamos a recibir, con más o menos pérdidas y retrasos, 4 flujos de video. Antes de pasar estos flujos al siguiente proceso hay que sincronizarlos ya que pueden llegar paquetes desordenados, o no llegar. Una vez recibidos los flujos y sincronizados, se pueden pasar al siguiente proceso.

2.3.2. Descodificación.

Fig 2.6 Esquema decodificador

En este punto tenemos 4 flujos de video codificados y nuestro objetivo es unirlos para rehacer el flujo original. Pero para ello tenemos que tratar de nuevo el video a nivel de píxel, por lo que debemos pasar el video, de nuevo, a formato YUV. Este proceso se debería poder hacer con el mismo codificador que se ha utilizado en el codificado del transmisor, por lo que ya tenemos una característica más a la hora de escoger codificador. A partir de este punto, y siempre intentando asegurar el sincronismo de los flujos, podemos pasarlos al próximo proceso, el *merger.* Ahora bien, existe una excepción. En caso de que solo recibamos un flujo porque somos un dispositivo de pequeñas características, nos podemos saltar este paso y el siguiente, el merger, y reproducir el flujo que recibamos, ya que será el video tal y como lo recibiremos.

2.3.3. Merger

Fig 2.7 Esquema merger

El proceso del merger es el proceso central del receptor. Aquí uniremos los flujos recibidos en uno solo. Lo primero que tenemos que hacer es asegurarnos de que los flujos estén sincronizados. Aunque esto ya lo hemos hecho en el proceso anterior nos tenemos que asegurar que aquí llegan completamente sincronizados, ya que si no, al unir el video, el flujo resultante sería algo completamente distinto a lo que queremos.

Una vez estamos seguros que los flujos nos llegan con el orden y la latencia correctos, procedemos a unirlos en solo uno. Este proceso es diferente, dependiendo de cuál es el perfil de cada usuario. Es decir, los móviles se saltan este paso ya que ya tienen un flujo completo. Los usuarios que quieran recibir dos flujos, deben unirlos de una manera, los de tres, de otra y los de cuatro de otra. Aunque básicamente, será cogiendo un píxel de cada flujo de entrada y copiándolo en el flujo de salida. Este proceso es altamente delicado, y por tanto hay que tener mucho cuidado con él.

Lo que haremos para reconstruir el flujo original es dividir la imagen en una rejilla de cuadros de 2x2 píxeles, y asignarle a un flujo la misma posición en todos los cuadros (esquinas superior o inferior e izquierda o derecha). Esta posición tiene que ser, lógicamente, la misma que en el video original. Es decir, el flujo 1 del emisor, será el flujo 1 del receptor. Si nos falta algún flujo (ya sea porque aun no nos llega, lo hemos perdido, o porque no somos un usuario que necesite los 4 flujos), rellenaremos el hueco teniendo en cuenta el valor de los píxeles cercanos.

2.3.4. Reproducción

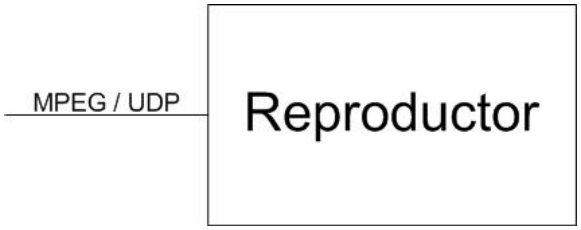

Fig 2.8 Esquema reproductor

Finalmente, del proceso anterior ya obtenemos el flujo original en el destinatario, el receptor. Ahora solo nos queda visionarlo, pero existe un problema. El flujo que tenemos es YUV, y además de ser un formato no muy apto para ser reproducido, debido a la gran cantidad de datos que hay que tratar, la verdad es que no existen muchos reproductores con soporte para este formato. Así que la solución más sencilla es codificar este flujo a algún formato conocido por todos los reproductores y utilizar un reproductor que lo soporte. En concreto, en este TFC usaremos el VLC debido a que se trata de un reproductor muy robusto, potente y a la vez bastante ligero en cuanto a consumo de recursos y sencillo a la hora de tratar con él (ejecutarlo, pasarle parámetros, ...). Además, tiene soporte para trabajar con un flujo como entrada. Para codificar el video utilizaremos el mismo codificador que en el emisor, en el proceso de codificación.

2.4. P2P

Una vez tenemos un sistema MDC diseñado e implementado, lo tenemos que integrar en una red P2P. Este paso no lo implementaremos, ya que va más allá del TFC, pero sí que lo explicaremos teóricamente y definiremos sus funcionalidades.

Hemos definido lo que tiene que ser un emisor y un receptor MDC, pero dentro de la red, los usuarios no son ni una cosa ni otra, sino una mezcla de los 2, ya que, por definición, en una red P2P, los peers son tanto clientes como servidores, o lo que es lo mismo, tanto receptores como emisores.

2.4.1. Fuente

Eso sí, a pesar de ser una red P2P, que idealmente son redes completamente descentralizadas, existe un elemento que se escapa ligeramente de este concepto, la fuente. Para fomentar el concepto de “todos iguales”, se tendría que habilitar la posibilidad de que cualquier peer pueda ser fuente de videos. Eso sí, tiene que tener en cuenta sus características técnicas, CPU, RAM, ancho de banda, a la hora de elegir qué tipo de video (resolución, bitrate,...) tiene capacidad de servir. Aun así, consideramos que debe de haber un elemento que sea, por decirlo así, un “servidor oficial” que tenga capacidad para servir canales de manera estable y permanente. El proceso para servir un video debe ser el siguiente:

- Acceder al programa.
- Elegir el video que se quiere transmitir, indicar su resolución.
- El video se empieza a tratar localmente y se abre un socket a la espera de recibir peticiones de transmisión de flujos.
- A la vez, en el servidor de la aplicación se crea un registro del vídeo, con la IP del servidor del vídeo y el puerto, y se publica el video para los peers.

Hay que tener en cuenta que algunos de los pasos siguientes se ejecutan en la máquina local, y otros en el servidor de la aplicación.

2.4.2. Peer

Estos son los usuarios finales de la aplicación. Hasta ahora se consideraba que los Peers de una Red P2P eran homogéneos, es decir, todos iguales, pero esta tendencia está cambiando debido a la diversidad de dispositivos que se pueden conectar a Internet y en este TFC estamos tratando específicamente con peers heterogéneos, diferentes. Por lo tanto existen diferentes perfiles de usuarios que pueden conectarse a la red. Dependiendo del número de flujos que quiera

cada Peer su comportamiento será diferente en cuanto al número de flujos a tratar, pero el funcionamiento básico será muy diferente entre todos ellos:

- El Peer se da de alta en la web y aparece una lista de los vídeos o canales que están siendo transmitidos en cada momento.
- Una vez escogemos uno, el servidor de la aplicación nos dará la IP del servidor del video y el puerto al cual tenemos que enviar nuestra petición.
- Una vez enviada la petición con nuestra IP y puerto/s, que será para un flujo determinado (el 2, el 3...), abrimos el puerto/s para recibir los flujos que esperamos.
- Una vez se empieza a recibir el video, se envía al receptor MDC que ya se encarga de tratarlo y reproducirlo.
- Como es una red P2P necesitamos abrir un puerto para esperar peticiones de otros Peers.
- En caso de recibir alguna petición, si nosotros tenemos ese flujo lo enviaremos.

Lógicamente, el proceso que acabamos de explicar solo sirve para los primeros videos, ya que de lo contrario, la aplicación acabaría siendo cliente/servidor ya que todos los Peers recibirían el video de la fuente original. Esto no es sostenible ni escalable, por lo que hay que crear un método para que cada Peer anuncie que flujo puede ofrecer. Esto se puede hacer de dos maneras:

1. El servidor del video, si no puede enviar el flujo porque está al límite de su capacidad, le envíe la IP de algún cliente que tenga el video. En caso de que este cliente se desconecte, volveríamos a hacer una petición al servidor.
2. En el servidor de la aplicación crear una pequeña base de datos en la que cada peer anuncie que flujos puede ofrecer. Cada vez que un Peer quiera un flujo determinado quien se lo puede ofrecer en esta base de datos, como en unas Páginas Amarillas. Si el cliente que nos está sirviendo el vídeo se desconecta volveríamos a consultar la base de datos.

2.4.3. Interfaz gráfica

Toda esta aplicación necesita una interfaz gráfica. Debido a su bajo coste de recursos, y compatibilidad con cualquier sistema operativo y navegador, una buena solución podría ser usar una aplicación web. En ella habría un login, tras el cual, dependiendo de nuestro perfil, accederíamos a una parte de la aplicación o a otra.

Si somos servidor de vídeo, por ejemplo, la aplicación nos pedirá datos del video: la resolución (ya que el YUV es un formato raw), y el nombre. Una vez indicados, habría un botón “transmitir” que al pulsarlo se ejecutara la aplicación de transmisión de vídeo. Es importante tener en cuenta que si queremos que el video se suba desde nuestro terminal, deberemos ejecutar el programa

transmisor localmente, por lo tanto hay que pensar como interactuar desde el navegador con programas locales.

Si somos Peer, al acceder a la aplicación, veríamos una lista de los vídeos o canales disponibles y un botón para reproducirlos. Nuevamente, para que el vídeo recibido sea tratado y reproducido localmente, necesitamos ejecutar el programa Peer (receptor + transmisor) de manera local, así como el reproductor, por lo tanto tenemos que volver a interactuar desde el navegador con el sistema local.

CAPÍTULO 3. IMPLEMENTACIÓN

3.1. Lenguajes y herramientas

En este capítulo se va a hacer una explicación de las tecnologías y herramientas utilizadas para el desarrollo de la aplicación. También se van a definir los módulos implementados.

3.1.1. Módulos MDC

Para la implementación de la herramienta MDC se ha utilizado el lenguaje de programación Java [4]. La elección de este lenguaje frente a otros se debe a su carácter de multiplataforma, facilitando así la compatibilidad entre diferentes máquinas para su desarrollo y su funcionamiento. También se valoró el rendimiento que podría tener la aplicación si estaba escrita en Java contra Java Nativo. Después de varias pruebas se resolvió que el rendimiento era similar y se optó por utilizar Java directamente.

Para el desarrollo de la aplicación se ha utilizado el programa Eclipse[5]. Se trata de una herramienta de código abierto de una gran potencia. Además, su gran número de plugins disponibles permiten configurar la aplicación a medida de nuestras necesidades.

3.1.2. Interfaz Gráfica

Toda esta aplicación necesita una interfaz gráfica para la interacción del usuario. En esto caso hemos optado por una interfaz muy sencilla y funcional para facilitar al máximo su uso. Los altos requerimientos de procesado que necesita la aplicación tanto en el merger como en el Splitter hacen que el resto de módulos tengan que reducir su consumo de CPU al máximo, por eso descartamos automáticamente el Swing de Java, ya que tiene un alto consumo de procesamiento y podría ralentizar la aplicación de manera innecesaria. Nos vamos a decantar, pues, por una aplicación web, ya que éstas apenas tienen carga computacional.

La aplicación Web se ha desarrollado utilizando Struts[8], que es un patrón de diseño de aplicaciones web basadas en el Modelo Vista Controlador (MVC). Este patrón, como dice su nombre, tiene 3 componentes: la Vista se encarga de mostrar los datos por pantalla, y se trata a través de páginas JSP. El Controlador es la parte lógica de la aplicación. Se programa en Java y se llama a través de servlets llamados Actions. El Modelo consta de formularios que funcionan a modo de intermediario entre la Vista y el Controlador. Para trabajar con Struts vamos a utilizar un plug-in de Eclipse diseñado especialmente para ello llamado MyEclipse [6]. Y como servidor de la aplicación Web un Apache Tomcat 5.5.[9]

3.2. Módulos

3.2.1. Fuente

La fuente de video que vamos a utilizar en este TFC es un archivo en disco. Para futuras versiones se puede plantear dotar de soporte para utilizar cámaras de video como fuente, ya que existen varias en el mercado que tienen salida de YUV. Así se podría utilizar esta herramienta para la transmisión de vídeo en directo. Pero en este TFC nos vamos a centrar en la transmisión desde un archivo.

Java permite abrir un archivo y leer desde él. Esto se hace a través de un FileInputStream y un DataInputStream que permiten leer el archivo de diferentes maneras. En concreto, vamos a crear un array de bytes de un tamaño en concreto y vamos a copiar en él los bytes necesarios hasta llenarlo. Una vez hecho esto se pasarán al módulo del Splitter para que los divida en 4 flujos. Existe la opción de leer los bytes de uno en uno, pero esto obliga a hacer una acción de I/O para cada byte lo que ralentiza el sistema. En concreto, un vídeo de unos 30 segundos con una resolución de 624x352 tarda unos 10 minutos en que todos los bytes sean leídos. Por lo tanto se descartó esta opción y se decidió leer los bytes del archivo en bloques. Debido a que lo mínimo que se puede representar en pantalla es un frame se decidió que el bloque tenía que temer el tamaño de un frame. Así, tenemos la ventaja de que solo hay una acción I/O para cada frame, acelerando el sistema, ya que los bytes, una vez copiados en el array de bytes y estar en RAM son mucho más rápidos de recorrer. Para un video como el anterior, de 624x352 píxeles, el proceso se puede ejecutar en tiempo real, es decir, a la velocidad de reproducción del video.

Este es el código para leer el archivo:

```
//Creación del fichero y del DataInputStream para su lectura.
File fichero =new File (nombreFichero)
FileInputStream fin = new FileInputStream (nombreFichero)
DataInputStream din = new DataInputStream(fin)

//Creación del array de bytes para leer el archivo
BufferBytes frame = new BufferBytes (tamañoFrame)

//Lectura del fichero, tantas veces como frames tenga el archivo
din.read(frame.bufbytes);
```

BufferBytes es una clase propia que tiene un atributo ‘bufbytes’ que es un array de bytes y un método ‘AddByte’ para añadir un byte al array, que aunque en este caso no se ha utilizado, se usará más adelante.

3.2.1.1. Control de flujo

Durante la implementación de la aplicación apareció un problema que tenía que ver con la velocidad de lectura del video y el tamaño del mismo. Si el video es de un tamaño estándar, del orden de 600x400, no existe ningún problema, pero para videos más pequeños, como el famoso 'Foreman', un video muy utilizado en la implementación de herramientas de video, que tiene un tamaño CIF, la lectura se hacía demasiado rápido. Al leer el video más rápido de lo que se tiene que reproducir y transmitirse a la misma velocidad provoca que el sistema sea inestable, porque en la recepción la aplicación recibe más datos de los que puede reproducir, llenando así los buffers de recepción y perdiendo inevitablemente datos. Para solventar este problema introduje un control de flujo en el programa que funciona de la siguiente manera: Existe un atributo booleano llamado 'readframe' que funciona como semáforo para seguir leyendo frames de la fuente. Cada vez que el programa intenta leer un frame mira ese atributo, si está 'true' lo lee y si está 'false' se espera hasta que cambia. Por otro lado se crea un Thread que cada 1/24 segundos accede a ese atributo y establece su valor como 'true'. Esto hace que cada 1/24 segundos sólo se pueda leer un frame, evitando así posibles problemas en la reproducción de los archivos.

Esta problemática solo afectaba a los vídeos con una resolución baja, pero la máquina en la que se ha implementado y probado la aplicación es de gama media-baja, por lo que si se ejecutara en un servidor o simplemente en otra máquina que tenga una capacidad de procesado mayor los videos con una definición estándar, o incluso en HD, podrían verse afectados.

3.2.2. Splitter

Para dividir el flujo en 4 descriptores lo único que tenemos que hacer es crear otros 4 ‘BufferBytes’ y recorrer el anterior copiando los bytes en el BufferByte del descriptor correspondiente. Aquí aparecieron dos posibilidades. Lo primero que se pensó fue en hacer coincidir cada grupo de píxeles de luminancia con sus respectivos píxeles de crominancia. Es decir, a modo práctico, se dividía el frame en grupos de 2x2 píxeles y se repartían dichos grupos en los respectivos descriptores. De este modo pretendía respetar la relación original de los píxeles Y, U y V, para alterar lo mínimo posible el video. El resultado, sin embargo, demostró que esta manera de dividir el video, aunque más fiel al video original, ofrecía un resultado de baja calidad. Los descriptores creados parecían pixelizados. Por esta razón probé hacer la creación de los descriptores de otra manera, repartiendo los píxeles a los descriptores de uno en uno. De esta manera la relación entre los componentes Y, U y V se deshace con respecto al video original, y aunque a priori podría parecer que los descriptores resultantes serían menos fieles al original, la realidad es que se crean unas imágenes mucho más suavizadas y más parecidas al original. La razón para ello (a pesar de no respetar la correspondencia Y, U y V del archivo original) es la similitud de un píxel con el píxel de al lado. En una imagen los píxeles suelen ser muy parecidos a sus vecinos. Esta característica nos beneficia en este caso porque un píxel Y, que en el video original se corresponde con un píxel U[1], en el

descriptor se corresponderá con otro píxel U^2 , y estos dos píxeles U^1 y U^2, al ser vecinos en el video original, hay muchas posibilidades de que sean parecidos. Además, se da el caso que los cambios de crominancia son menos bruscos que los de la luminancia, cosa que nos beneficia, porque se evita un salto muy grande entre píxeles vecinos.

Para hacer la división del video (split), partimos del proceso anterior, el de capturar la fuente. Nos habíamos quedado en que capturamos el video frame a frame. Una vez tenemos el frame en memoria, en concreto en un 'byte[ ]' del tamaño del frame lo que tenemos que hacer es recorrerlo y copiar los bytes en otros arrays de bytes que se corresponden con los descriptores. Si queremos hacer el split de la primera manera, el proceso es el siguiente: se recorren 2 líneas del frame y se copian los bytes de dos en dos a los descriptores 1 y 2 (dos al 1, dos al 2, dos al 1,...). Luego se recorren las dos siguientes y se hace lo mismo que antes, pero con los descriptores 3 y 4. Esto para la luminancia. Para la crominancia tenemos que separar los pixeles de uno en uno, y recorrer las líneas de una en una.

Si queremos hacer el split de la segunda manera, el proceso es parecido. En este caso las líneas del frame de una en una, y los píxeles se separan de uno en uno. O sea, los píxeles de la primera línea serán para los descriptores 1 y 2 (uno para el 1, uno para el 2,....) y los de la segunda línea para los descriptores 3 y 4. En este caso no hace falta diferenciar entre luminancia y crominancias, haciendo el código algo menos complejo, ya que en ambos casos, los píxeles se separan de uno en uno.

A continuación vemos el código para los dos métodos de hacer el split:

Método 1:

```
      while(g<framesize){ //Hasta llegar al final del frame
            j=0;
            while (j<2*size){ //size es el tamaño de la línea
                        frame1.AddByte(frame.bufbytes[g]);
                  frame1.AddByte(frame.bufbytes[g+1]);
                  frame2.AddByte(frame.bufbytes[g+2]);
                  frame2.AddByte(frame.bufbytes[g+3]);
                  g=g+4; //Recorre el frame
                  j=j+4; //Recorre cada línea
            }

            while(j<4*size){
                  frame3.AddByte(frame.bufbytes[g]);
                  frame3.AddByte(frame.bufbytes[g+1]);
                  frame4.AddByte(frame.bufbytes[g+2]);
                  frame4.AddByte(frame.bufbytes[g+3]);
                  j=j+4;
                  g=g+4;
            }
    }
```

Método 2:

```
    while(g<framesize){ //Hasta llegar al final del frame
            j=0;
            while (j<size){ //size es el tamaño de la línea
                  frame1.AddByte(frame.bufbytes[g]);
                  frame2.AddByte(frame.bufbytes[g+1]);
                  g=g+2; //Recorre el frame
                  j=j+2; //Recorre cada línea
            }

            while(j<2*size){
                  frame3.AddByte(frame.bufbytes[g]);
                  frame4.AddByte(frame.bufbytes[g+1]);
                  j=j+2;
                  g=g+2;
            }
     }
```

3.2.3. Codificador

Para codificar el video se ha utilizado un codificador externo a Java ya que no se ha encontrado ninguna librería de java con las suficientes prestaciones ni rendimiento para la herramienta. Hay que tener en cuenta que en el servidor del video se va a necesitar una instancia de dicho codificador ejecutándose para cada descriptor ya que cada uno se tiene que codificar por separado. Existen varios codificadores que se pueden utilizar: 'mencoder', 'ffmpeg', 'ffmpeg2theora' o 'vlan'. De entre todos ellos el que ofrece unas prestaciones más acordes a las necesidades de la herramienta es el 'ffmpeg'. Básicamente, lo que se necesita es que sea capaz de tratar con flujos de video (vía UDP y TCP), que sea capaz de tratar con videos YUV, que soporte el mayor número posible de códecs y que consuma los menos recursos posibles. Todas estas características las cumple 'ffmpeg' y son la razón por las cuales fue el elegido. A continuación, las principales características de los otros codificadores y la razón por las que fueron descartados:

- **Ffmpeg2theora**: Una versión del Ffmpeg especializada en el formato “ogg theora”. Incapaz de tratar con flujos TCP ni UDP.
- **Mencoder**: Funciona con unas librerías creadas para Ffmpeg “libavcodec”, por lo que parece más óptimo utilizar el codificador para el que fueron creadas. Además, se trata de un codificador bastante centrado en el reproductor Mplayer y es menos configurable que Ffmpeg.
- **Vlan**: Se trata de un codificador del propio reproductor. Es el menos recomendable de todos, ya que es muy pesado al tener que levantar una interfaz gráfica para cada instancia. Además, no soporta YUV.

Para usar el 'ffmpeg' en la aplicación debemos ejecutar una rutina externa en java. Esto lo hacemos con el comando típico para esto:

```
Runtime r = Runtime.getRuntime();
Process p = null;
try {
        p = r.exec("ffmpeg");
}
catch (IOException e){
}
```

Con este código java ejecuta la comanda “ffmpeg” en el sistema. Con esta comanda, se ejecutaría “ffmpeg”, pero ese ffmpeg no haría nada ya que no le hemos indicado ni las opciones ni el fichero de entrada ni el de salida. A continuación se expone una lista con las opciones más interesantes que se van a usar en esta herramienta:

- -s El tamaño del video. Debemos indicárselo ya que la fuente está en YUV. El tamaño se indica en píxeles con la siguiente estructura: 'ancho'x'alto'.
- -r Indica el framerate
- -f Tipo de video.
- -sameq Decimos que la calidad de la salida sea la misma que la de la entrada.
- -i Vídeo de entrada.

La estructura de la oración debe ser la siguiente:

- ffmpeg (opciones video entrada) -i (video entrada) (opciones video salida) (video salida).

Se observa que el video de salida no se indica con ningún flag. Ffmpeg busca el parámetro sin flag.

Todo esto se aplica 4 veces en esta aplicación, una para cada descriptor. Para pasar cada uno al 'ffmpeg' se tiene que hacer con un flujo. Es decir, cuando el splitter está leyendo la fuente tiene que pasarlo a la vez al codificador. Esto se puede hacer mediante el protocolo UDP ya que se trata de un protocolo bastante ligero. El problema es que lo que estamos transmitiendo es YUV, lo que significa una enorme cantidad de datos, tal y como se ha explicado en el apartado 1.2.1. Esto para videos pequeños no es problema, pero para videos de un tamaño mayor (PAL) supone que se van a perder paquetes irremediablemente. Y al tratarse de un video YUV, la pérdida de un solo paquete se arrastra por todo el video ya que no existen cabeceras que permitan resincronizar el video, por lo que debemos asegurar esta conexión y descartar UDP. La opción que queda es TCP, que asegura la conexión y la transmisión íntegra de los archivos. Cuando 'ffmpeg' recibe un flujo mediante TCP lo hace como cliente. Esto significa que al ejecutar 'ffmpeg', éste intenta conectar con un servidor que le enviará el video. Hay que tener en cuenta que es el propio 'ffmpeg' el que inicia la conexión, es el que envía el mensaje 'syn' y si el puerto al que lo envía no está escuchando, se queda zombi esperando respuesta. No existen reenvíos de 'syn', lo que por otra parte, sería más que

recomendable. Pero debemos asegurar que tenemos un ServerSocket escuchando el puerto al que vayamos a enviar las peticiones. En esta aplicación, esto se hace antes de nada. Primero se crean los sockets de envío, las instancias de 'ffmpeg' y la conexión entre ellos, y luego se empieza a leer el video. La creación de las instancias de 'ffmpeg' se hace mediante Threads que se ejecutan paralelamente al hilo principal del programa, y para que funcionen correctamente cada instancia necesita ejecutarse en un Thread independiente, porque 'ffmpeg' da problemas ejecutando 2 instancias en el mismo proceso. En cambio, los sockets se crean de manera secuencial, por lo que el programa no empieza a leer el video hasta que no se establezca la conexión. Hay que comentar que jamás me ha fallado este proceso de establecimiento de conexiones en ninguna de las numerosísimas pruebas que he realizado a lo largo de toda la implementación, pero quizás sería interesante a la hora de futuras implementaciones la creación de un mecanismo de seguridad en caso de que este proceso falle.

3.2.4. Transmisión.

El vídeo, cuando sale del codificador ya está preparado para su envío a través de la red P2P. Pero la red P2P es algo que no debía tratarse en este TFC más que a nivel teórico. La implementación no ha sido tratada por lo que la transmisión va a ser a un puerto en local, en la misma máquina, conocido por el cliente para simular una red óptima de laboratorio, es decir, una red sin pérdidas y sin retardos. Futuras implementaciones deberían empezar en este punto. En este caso vamos a transmitir a la misma máquina a través de un puerto UDP. El uso de este protocolo se justifica ya que al ser un video codificado, la cantidad de bytes que se transmiten es mucho menor que en el caso de los videos YUV, y al ser un video en local es muy poco probable que tenga pérdidas. La transmisión vía UDP la puede hacer el mismo 'ffmpeg', estableciendo como archivo destino una dirección UDP. Como por ejemplo la siguiente : udp://localhost?localport=1234. Con el ejemplo anterior, el video será transmitido al puerto 1234 UDP de nuestra máquina.

3.2.5. Recepción + Descodificación.

A priori éstas 2 acciones deberían ser módulos diferentes pero en la práctica no. Estando en el receptor, lo que vemos es que tenemos que escuchar unos puertos UDP a los que nos van a llegar flujos codificados en MPEG y ffmpeg, que es el descodificador que vamos a usar, ya es capaz de hacer esta tarea. Es decir, al arrancar ffmpeg se configura para que escuche los puertos donde se van a recibir los flujos y ya está. En este punto debemos recibir de 1 a 4 flujos, dependiendo del tipo de peer que se conecte, por lo tanto se deben abrir tantas instancias de ffmpeg como flujos se necesiten. Igual que en el emisor, estas instancias deben ser abiertas cada una en un hilo independiente ya que ffmpeg tiene problemas para ejecutar más de una instancia en el mismo proceso. Aquí aparece un problema. A la salida del ffmpeg hay de nuevo un flujo en raw, lo que significa una gran cantidad de datos. Dependiendo de las características del receptor, puede no haber problemas, pero en la práctica, en

la mayoría de los casos se pierden paquetes, lo que en este punto es crítico, perder un solo paquete, al ser un flujo raw, se arrastra a lo largo de la transmisión. Es por eso que la conexión de salida del ffmpeg se tiene que hacer por TCP. Ahora bien, ya se han comentado las particularidades de ffmpeg con TCP. Ffmpeg se comporta como un cliente TCP, envía una petición (SYN) en el momento en el que se ejecuta y se queda esperando respuesta sin enviar repeticiones. Es decir, si no recibe respuesta se queda como un proceso en estado de *wait* . Para solucionarlo, antes de ejecutar ffmpeg, se creará un ServerSocket que atenderá las peticiones de ffmpeg. Es muy importante que estos dos pasos (creación del ServerSocket y de las instancias de ffmpeg) estén coordinados ya que de hacerlo descoordinadamente puede dar con que la aplicación no funciona.

3.2.6. Merger.

Los sockets creados en el punto anterior reciben una serie de bytes. De hecho, en teoría deberán tratar con una serie infinita de bytes. En cada paquete TCP se envía un número concreto de bytes que al ser recibidos se guardan en un atributo del estilo “byte[]”. Estos arrays de bytes luego se mezclan para dar como resultado un único flujo, que ya será el video recompuesto (en YUV). Esta recepción i mezcla (merg) de dichos bytes no es trivial ya que pueden darse problemas de sincronía entre flujos. La solución que se ha dado para solventar este apartado es la siguiente:

- El merger sabe a priori cuantos flujos debe mezclar, más adelante se explica cómo lo sabe.
- Teniendo en cuenta ese número, abre tantos sockets como sean necesarios, uno por cada flujo.
- Los sockets recibirán de manera recursiva los arrays de bytes.
- Una vez recibido un paquete en cada socket, deben ser mezclados y enviados al codificador.

El merg de los flujos variará, lógicamente, dependiendo de los flujos de los que se dispongan, pero en esencia es igual en todos los casos. Básicamente, imaginamos un frame dividido en una cuadrícula. Cada cuadro es de 2x2 píxeles y cada uno estos píxeles pertenece un flujo. En concreto, el flujo 1 corresponde con el cuadro superior izquierdo. El flujo 2 con el superior derecho. El 3 con el inferior izquierdo y el 4 con el inferior derecho. Lógicamente, los 4 píxeles solo se rellenaran en el caso del cliente de 4 descriptores. En el resto de casos nos faltará al menos uno de ellos. La manera en que se rellenan los huecos es la sencilla: se copia un píxel vecino. Al tener el principio de “similitud por vecindad”, los píxeles se parecen bastante y no causan ningún efecto visual no deseado. Ahora, entonces, tenemos un flujo de video ya rehecho, pero este video no se puede reproducir, al estar en YUV. Además, este flujo resultante tiene la misma resolución tanto si solo se ha recibido 1 flujo como si se han recibido 4 y, al entender que un cliente que solo reciba un flujo tendrá una pantalla pequeña, no es óptimo. Es por ello que se hace necesaria otra etapa de codificación.

3.2.7. Codificación.

De nuevo ffmpeg se cruza con este proyecto. En esta ocasión recibe un flujo de bytes en formato raw y debe ofrecer a la salida un video reproducible por nuestro proyector. En este caso se ha decidido que sea MPEG2. Además, debe hacer un reescalado dependiendo de cuantos flujos estemos recibiendo. Para los clientes con solo un descriptor, la resolución será *x*, y para una con dos descriptores será *2x.* Así sucesivamente y de manera proporcional hasta 4 descriptores

De nuevo hay que tener cuidado con ffmpeg y el protocolo TCP. En este caso se usará a la entrada, pero no a la salida. A la salida ya habrá un flujo que estará codificado, con su considerable rebaja de tamaño comparada con el YUV. Es por esa razón que podemos usar UDP para transportar el flujo hasta el reproductor.

3.2.8. Reproducción.

Para reproducir el video se utilizará el programa VLAN. Dicho programa puede escuchar un puerto y reproducir lo que le llegue por él. Simplemente debemos arrancarlo con los parámetros necesarios y El VLAN escucha el puerto indicado y lo reproduce. En la siguiente figura se puede ver al propio VLAN reproduciendo un archivo.

3.3. Struts

Struts es un Framework de JAVA que permite la creación de aplicaciones web mediante Servlet de manera sencilla bajo el patrón MVC, Modelo Vista Controlador. Es decir, divide la aplicación en estas 3 capas.

Con Struts solo se ha implementado la parte del servidor. El cliente es enlazado desde Struts, pero se trata de una ejecución diferente. La aplicación tiene la estructura que podemos ver en la siguiente imagen:

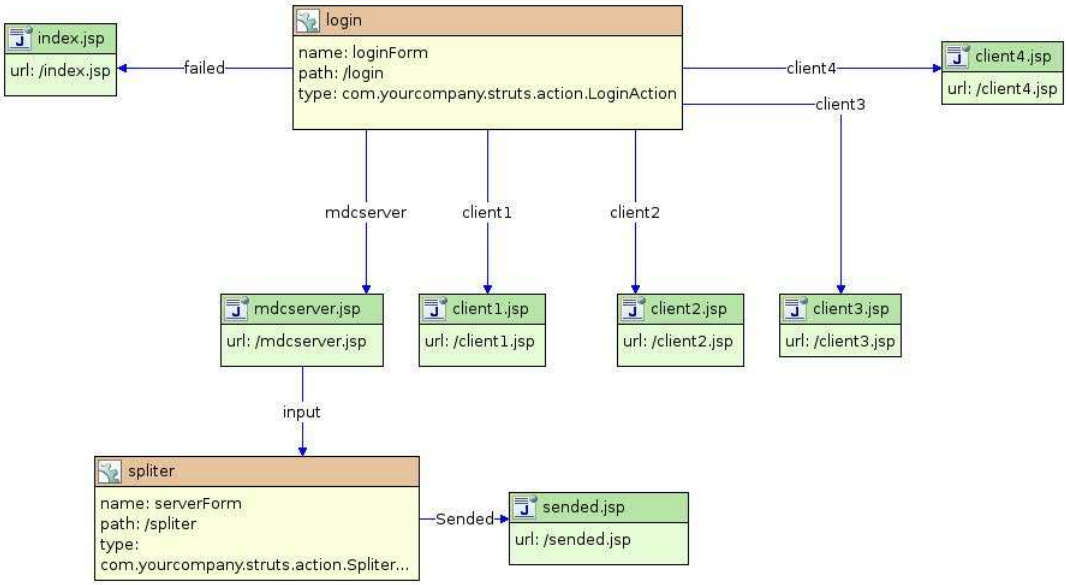

Fig 3.1 Archivo struts-config.xml

3.3.1. Vista

La Vista se compone de páginas web, se podría decir que es la interfaz gráfica. Son una serie de páginas JSP que permiten al usuario interactuar con la aplicación. Tiene formularios que permiten esta comunicación En esta aplicación consta de 7 páginas JSP. La primera es un *login.* En él hay que identificarse. Con el nombre de usuario la aplicación distingue de que tipo es y envía hacía la página JSP correspondiente. En la página *mdcserver,* una vez introducidos los datos del video que se quiere servir, nos envía a la página *sended*, que es igual que la anterior pero añadiendo el mensaje que el video está siendo servido. Las páginas de los clientes lo único que hacen es mostrar una lista con los videos que hay sirviéndose en cada momento. En cada uno de los videos de la lista hay un enlace mdc:// que abre el video que queremos reproducir.

3.3.2. Controlador

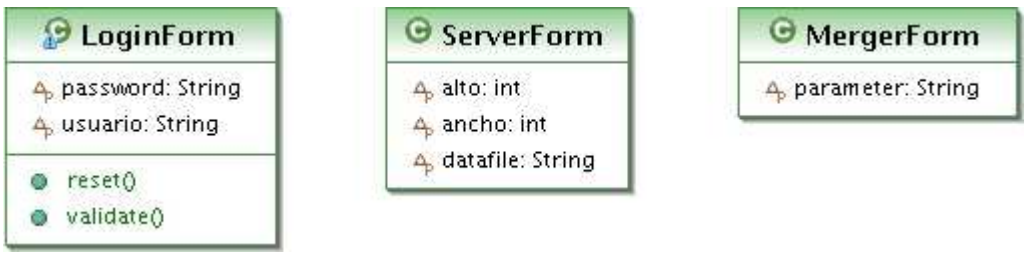

Fig 3.2 UML del controlador

El controlador de una aplicación Struts vienen a ser formularios que recogen la información de la vista y se la pasan al controlador para que las gestionen. En este caso solo tenemos 2 formularios en la aplicación: uno en la página index.jsp, para recoger los datos del *login* y otro en la página mdcserver.jsp. Para recoger los datos del video a servir. Las páginas de los clientes no tienen ningún formulario, simplemente un enlace mdc:// que dependiendo de sus atributos abre la aplicación cliente. El Form del login consta de 2 Strings (nombre y usuario) y el del servidor consta de un String (nombre del video) y 2 enteros (ancho y alto).

3.3.3. Modelo

Fig 3.3 UML del modelo

El modelo es la lógica de negocio de la aplicación. Una vez recibidos los datos de la vista, ejecuta las acciones necesarias. En este caso, existen 2 actions. El del login compara las 2 cadenas introducidas para verificar que se trata de un usuario válido del sistema y reenvía a la página correspondiente a cada uno. El action del servidor recoge los datos del video y crea una instancia del splitter con los con los datos del video como parámetros.

CAPÍTULO 4. FUNCIONAMIENTO

En este capítulo se va a explicar el funcionamiento de la aplicación.

Cuando abrimos el navegador web y accedemos a la aplicación vemos la imagen de la figura 4.1.

Fig 4.1 Página inicial de la aplicación

Es un menú muy sencillo en el que tenemos que poner nuestro usuario y nuestra contraseña. Si accedemos como servidor vemos en pantalla la imagen de la figura 4.2.

Fig 4.2 Página de servidor de vídeo

En esta pantalla se nos pide que introduzcamos los datos del video que queremos servir, el nombre, el ancho y el alto. El video se servirá desde nuestra máquina y lo deberemos tener alojado en un directorio en concreto del sistema. Este directorio no es configurable, y varía si usamos Windows como SO, o Linux. En este TFC se ha usado Linux y el directorio era /home/usuario/workspace. Si accedemos como cliente, accedemos a una pantalla similar a la que se muestra en la figura 4.3.

Fig 4.3 Página de cliente. 1 descriptor

Solo hay una diferencia entre las pantallas de los clientes: tal y como se muestra en la figura 4.3, hay una frase que dice “página de cliente. 1 descriptor”. Esta frase cambia según sean 2, 3 o 4 descriptores.

En la figura 4.4 se muestra el descriptor que se reproduce cuando pulsamos “Ver este video”. Al lado se muestra el video original para observar la diferencia de tamaño.

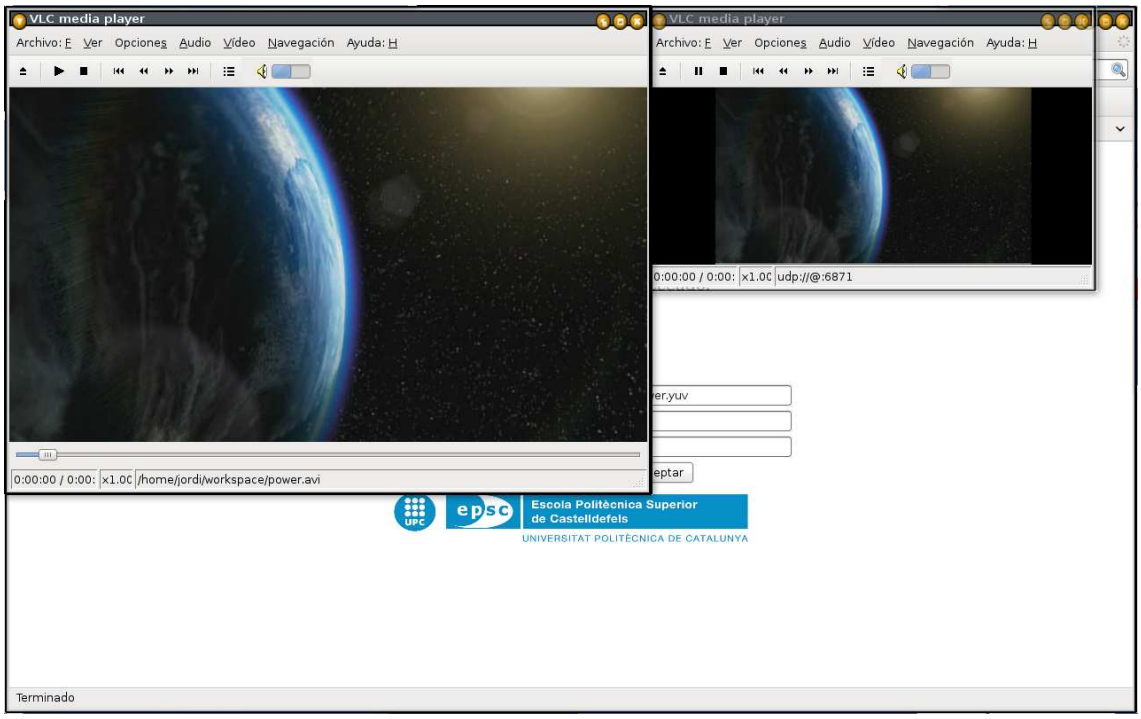

Fig 4.4 Reproducción de 1 descriptor al lado del video original

CAPÍTULO 5. FUENTES DE SOFTWARE

En este capítulo se van a explicar las diferentes clases creadas a lo largo del proyecto.

4.1. Fuente

4.1.1. MdcOpener

Fig 4.1 UML de MdcOpener

Esta clase es instanciada desde el action del servidor. Una vez le entramos los datos del video, llama a esta clase, que inicializa los datos del fichero en los atributos file, size y weight y además guarda en el atributo puerto el número del puerto donde tendrá ser servido el primer flujo. El resto serán los consecutivos. Esta clase extiende de Thread y por eso implementa el método run();.

4.1.2. Splitter

Fig 4.2 UML de Splitter

Esta clase es la que hace la división del vídeo. El MdcOpener llama a ésta y le inicializa los atributos con los que comparte nombre. Splitter implementa un método llamado Split() que es el que propiamente hace la división del flujo, pero antes ejecuta métodos de otras clases como son SocketCreator y FfmpegOpener.

4.1.3. SocketCreator

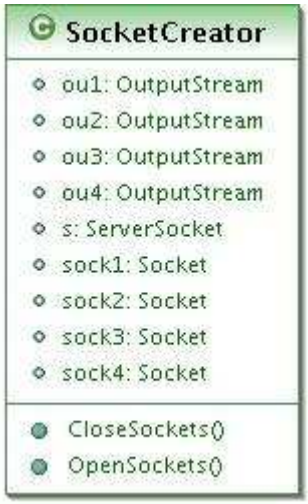

Fig 4.2 UML de SocketCreator

Esta clase se encarga de abrir los sockets para comunicarse con las diferentes instancias de Ffmpeg. Tiene 1 ServerSocket para escuchar las peticiones, 4 Sockets para conectar con cada una de las 4 instancias de Ffmpeg y 4 OutputStream para la el envío de los datos. Desde Splitter se llama al método OpenSockets(), que al ejecutarse en modo iterativo paraliza la ejecución del programa hasta que establece las 4 conexiones. Por esta razón es tan importante la coordinación con la siguiente clase, FfmpegCreator.

4.1.4. FfmpegCreator

Fig 4.4 UML de FfmpegCreator

Esta clase, como su nombre indica es la que se encarga de crear las instancias de Ffmpeg. Es una clase que extiende de Thread, por lo que implementa el método run(). Además tiene una clase anidada, RunFfmpeg, que también extiende de Thread, cuya única función es ejecutar una instancia de Ffmpeg. La razón para la creación de esta clase anidada es los problemas que da Ffmpeg cuando se ejecutan varias instancias en el mismo proceso, así que FfmpegOpener ejecuta 4 veces el run() de RunFfmpeg y muere. Tiene como atributos los datos de la resolución del video, ya que Ffmpeg necesita saber qué le va a llegar, y al tratarse de un archivo raw sin cabeceras hay que indicarlo.

4.1.5. ControlFlux

Fig 4.5 UML de ControlFlux

Esta pequeña clase se ejecuta en paralelo a la ejecución principal del programa, por eso extiende de Thread e implementa el método run(). Su única función es establecer el valor true a un atributo boolean del Splitter cada 1/24 segundos. El Splitter lee un frame entero cada vez y cada vez que hace una iteración, no continua la ejecución hasta que el atributo esté true (y le cambia el valor a false).

4.1.6. BufferBytes

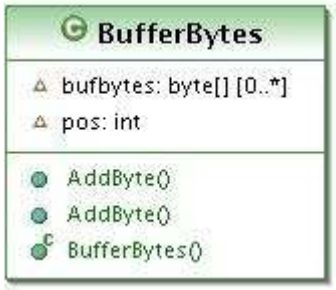

Fig 4.6 UML de BufferBytes

Esta clase sirve como estructura de datos. Todos los flujos que se leen durante el proyecto se copian al atributo bufbytes[] de esta clase. Además tiene implementados métodos para añadir bytes nuevos.

4.1.7. VideoObject

Fig 4.7 UML de VideoObject

Esta clase también es una estructura de datos. Como métodos solo tiene Getters y Setters. Cuando un vídeo se comienza a servir se guardan los datos en un Array de VideoObjects. Este Array es estático para que pueda ser consultado luego por el jsp de los clientes para listar los videos disponibles.

4.2. Receptor

El receptor es básicamente el proceso inverso, por lo que las clases serán similares, pero con la peculiaridad de que deberán estar instaladas en el ordenador del cliente. El navegador, al clicar en “Ver este video”, que en realidad es un enlace mdc://, llama a un pequeño script al que le pasa por parámetros los datos del video. Este Script llama al Merger y ejecuta el Vlan. A continuación todas las clases que el cliente tendrá instaladas en su máquina.

4.2.1. Merger

Fig 4.8 UML de Merger

Esta clase es la equivalente a Splitter en la fuente. Se encarga de la unión de los flujos. Tiene un método main, al que se le pasan por parámetros los datos del video. Además, igual que pasa en la Fuente, llama a métodos de otras clases, en este caso ReceiveFfOpener y SocketCreatroReceptor.

4.2.2. ReceiveFfOpener

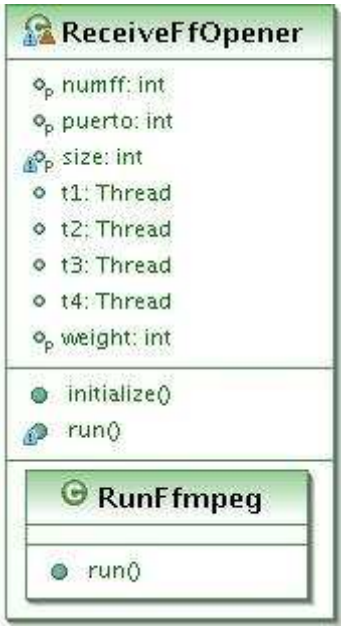

Fig 4.9 UML de ReceiveFfOpener

Tiene exactamente la misma funcionalidad que la clase FfmpegOpener. El cambio de nombre se debe a que durante la implementación hubo problemas de ejecución al llamarse las dos clases igual. La única diferencia que tiene con su clase "hermana" es que ésta tiene un atributo llamado numff que es un int que indica que tipo de clientes somos, y por tanto cuantos flujos vamos a recibir.

4.2.3. SocketCreatorReceptor

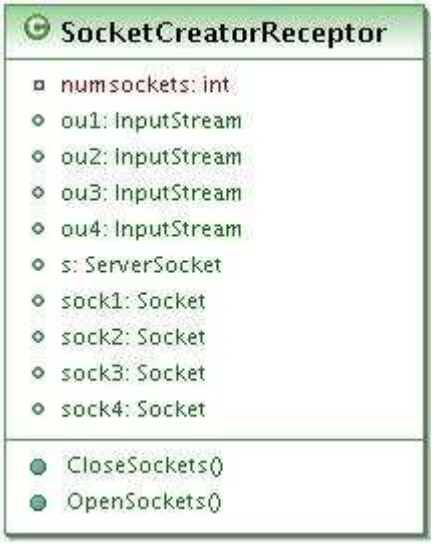

Fig 4.10 UML de SocketCreatorReceptor

A esta clase le pasa exactamente lo mismo que a la anterior: tiene una clase hermana en el servidor, en este caso la llamada SocketCreator. Son idénticas, excepto en el atributo que dice el número de descriptores que vamos a recibir.

CAPÍTULO 6. PLANIFICACIÓN

La realización de un proyecto de esta envergadura conlleva la división del trabajo en diferentes tareas. En este capítulo se definen cada una de ellas así como la planificación seguida a lo largo de todo el proyecto. Las diferentes tareas se definen a continuación:

- **Estudio previo**: Consiste en documentarse sobre la temática del proyecto, así como en el estudio de las diferentes tecnologías a utilizar.

- **Diseño**: Una vez elegidas las tecnologías se hace un primer diseño teórico.

- **Implementación:** Consiste en implementar la aplicación a partir del diseño.

- **Testeo**: Durante esta tarea se comprueba que la implementación funciona correctamente.

- **Documentación**: La redacción de esta memoria entra en este apartado.

6.1. Distribución temporal

A continuación vemos la distribución temporal del proyecto con las diferentes fases:

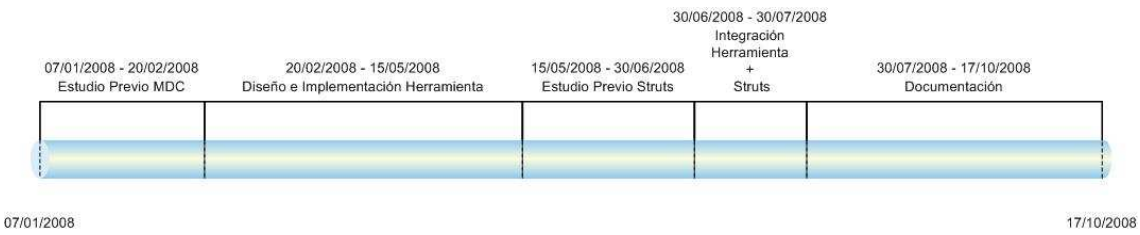

Fig 5.1. Distribución temporal.

6.2. Temporización de tareas.

A continuación se muestra la temporización detallada del proyecto:

Tabla 6.1 Desglose de tiempo.

Estudio Previo MDC		**60 horas**
	Lectura de Papers	20 horas
	Estudio de otras herramientas (LC, Coolruc, ...)	20 horas
	Estudio archivos YUV	10 horas
	Elección de codificador y reproductor	10 horas
Diseño e implementación Herramienta.		**150 horas**
	Configuración entorno	10 horas
	Programación	100 horas
	Testeo de la herramienta	40 horas
Estudio Previo Struts		**60 horas**
	Lectura documentación	50 horas
	Configuración entorno Struts	10 horas
Integración Struts + Herramienta		**50 horas**
	Creación aplicación modelo	15 horas
	Integración herramienta	35 horas
Documentación		**100 horas**
TOTAL		**420 horas**

CONCLUSIONES

C.1. Conclusiones tecnológicas.

A lo largo de este TFC se ha logrado el objetivo principal, la implementación de una herramienta MDC. Esta herramienta es capaz de servir un vídeo de una calidad media-alta en tiempo real. De hecho ha habido que crear un control de flujo que permitiese servir videos de menos calidad. Teniendo en cuenta que las pruebas se han hecho en una máquina de gama media-baja y los resultados han sido satisfactorios para vídeos de media calidad (700x400) se puede decir que en un servidor más potente la herramienta es capaz de servir vídeos de alta definición en tiempo real. El ancho de banda necesario para recibir 4 flujos es entre un 10% y un 15% superior a la recepción del vídeo original.

Para el desarrollo de la herramienta se han tenido que utilizar e integrar diferentes herramientas y técnicas: Java para la programación, Struts para la interfaz gráfica, ffmpeg para la codificación, vlan para la reproducción, y otras varias.

C.2. Conclusiones personales.

A lo largo del proyecto he adquirido conocimientos sobre diferentes tecnologías. Algunas de ellas no las conocía o solo a nivel teórico, como el framework Struts, los archivos YUV o los diferentes codificadores. Y en las que conocía he adquirido más práctica y agilidad en su uso, como por ejemplo Java, en concreto la gestión de ficheros y la programación de Sockets.

También he adquirido conocimientos en la gestión de proyectos de larga duración, creando un planning e intentando seguirlo para la correcta finalización del mismo.

Por otra parte he aplicado algo que a mi parecer es más importante que el propio conocimiento y que en la escuela, y en la universidad en general, se hace mucho énfasis, saber buscar la información o el conocimiento que no se posee. Así he descubierto, por ejemplo, el codificador ffmpeg, así como sus comandos.

Finalmente creo que ha sido muy interesante buscar poder trabajar con tecnologías tan diferentes como pueden ser Java, YUV o MPEG y su integración para la creación de una única herramienta.

C.3. Próximos pasos.

Desde mi punto de vista, los próximos pasos a seguir están muy claros: por un lado desarrollar la herramienta de una forma más extensa. Poder configurar el número de descriptores, el tipo de MDC utilizado en cada caso o usar diferentes codificaciones son unas características muy interesantes. Y por otro lado la implementación de la herramienta en una red P2P.

Otro tema que es importante mirar es la seguridad, que en este TFC no se ha tratado. Poder enviar los flujos codificados para que solo él/los legítimos receptores puedan reproducirlos o una gestión de usuarios más eficaz. Actualmente envía el nombre y la contraseña del login en claro, sin codificar.

C.4. Impacto medioambiental.

El impacto medioambiental que tiene este proyecto es el ahorro de energía que supone no necesitar una gran máquina para reproducir los contenidos. El permitir a dispositivos con un bajo consumo hace que no se necesiten ordenadores de sobremesa, que tienen un consumo de energía mucho menos optimizado, ya que con cualquier dispositivo móvil, podremos acceder a los contenidos ofrecidos.

ACRÓNIMOS

MDC: Multiple Description Coding.

P2P: Peer to Peer.

LC: Layered Coding.

HD: High Definition.

RTP: Realtime Transfer Protocol.

TCP: Transmission Control Protocol.

UDP: User Datagram Protocol.

PAL: Phase Alternating Line.

MVC: Modelo Vista Controlador.

UML: Unified Modelling Language.

TFC: Trabajo de Fin de Carrera.

BIBLIOGRAFÍA

[1] **Layered Multiple Description Coding**: Philip A. Chou, Helen J. Wang, and Venkata N. Padmanabhan.
URL: http://research.microsoft.com/~helenw/papers/ChouWP03.pdf

[2] **YUV**: Foro con lista de videos YUV en calidad HD (En línea).
URL: http://www.highdefforum.com/showthread.php?t=6537

[3] **YUV**: Información sobre este formato y sus subformatos (En línea).
URL: http://www.fourcc.org/yuv.php

[4] **JAVA Sun**: Web oficial de Java Sun, (en línea).
URL: http://java.sun.com/

[5] **Eclipse**: Web oficial del IDE eclipse. Descarga de la herramienta y sus módulos. (En línea).
URL: http://www.eclipse.org/

[6] **MyEclipse**: Web oficial de la extensión IDE eclipse propietaria. Descarga de la versión de prueba de los módulos. (En línea).
URL: http://www.myeclipseide.com/

[7] **Ffmpeg**: Web oficial de la aplicación Ffmpeg.
Descarga de la aplicación, manual de instrucciones y soporte. (En línea).
URL: http://ffmpeg.mplayerhq.hu/

[8] **Struts**: Web del proyecto apache para desarrollar con el *framework* Struts. Manual de instalación, guía de desarrollo, ejemplos, etc. (en línea).
URL: http://struts.apache.org

[9] **Apache Tomcat**: Web oficial del proyecto apache Tomcat. Descarga del servidor de aplicaciones Apache Tomcat. (En línea).
URL: http://tomcat.apache.org/

[10] Architecture for a Peer-to-peer Live Streaming Service for High Quality Media. Projecte Trilogy.

Algunos de los conceptos definidos, se han obtenido de: Wikipedia – La enciclopedia libre (En línea). URL:

http://es.wikipedia.org