ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Sistema de Grabación y Reproducción Sincronizada y su integración en un Sistema de Videoconferencia

PROYECTO FIN DE CARRERA

AUTOR:
Horacio Mijail Antón Quiles
DIRIGIDO POR:
Dr. Juan Carlos Guerri Cebollada

SISTEMA DE GRABACIÓN Y REPRODUCCIÓN SINCRONIZADA Y SU INTEGRACIÓN EN UN SISTEMA DE VIDEOCONFERENCIA

Capítulo 1.- Introducción y Objetivos **4**

El sistema de videoconferencia ya existente en el COMM 4

Objetivo del PFC 5

Contenido del resto de esta memoria 7

Capítulo 2.- Tecnologías implicadas **8**

Java 8

JNLP / Java WebStart 9

Java Media Framework (JMF) 11

Definiciones previas 12

Obtención de flujos multimedia a partir de un fichero. 13

Procesado del contenido multimedia 13

Objetos Controller 13

Players y Processors 14

Estados de un Player y un Processor 16

Librería Xvid 17

RTP (*Real-time Transport Protocol*) 19

Capítulo 3.- Decisiones de diseño **21**

La aplicación existente 21

Posibles nuevas configuraciones 22

Formatos contenedores de la grabación 24

Resumen 26

Capítulo 4.- Desarrollo **27**

Preliminares 27

Implementación de la primera configuración 30

Segunda configuración 34

Grabación simultánea 37

Doble reproductor 41

Desincronías entre diferentes reproducciones 42

Reproductor doble basado en un reproductor estándar 44

Implementado con Java WebStart 47

Aplicación a nueva funcionalidad: grabación de la pantalla 47

Capítulo 5.- **Detalles del desarrollo e implementación** **51**

Problemas generales con JMF 52

Problemas de instalación 52

Hyperthreading 53

Fragilidad del código 55

Problema de *frames* corruptos 56

Funcionamiento interno del clonado 57

Productor y consumidor 60

Formatos de grabación 64

Problemas anteriores ya solucionados 70

Desincronía entre reproducciones 71

Capítulo 6.- **Conclusiones y trabajo futuro** **80**

Resumen 80

Trabajo futuro: sugerencias para continuar mejorando JMF 82

Creación de herramientas 83

Documentación escasa y deficiente 85

Refinar y estandarizar trabajo ya hecho 87

Formatos 88

Investigar la temporización de la JVM 89

Estudiar integración con otras librerías, frameworks, etc. 89

Capítulo 7.- **Bibliografía** **91**

Índice de figuras

Figura 1.- Interfaz de la aplicación de videoconferencia del COMM 5
Figura 2 - Etapas del Processor 15
Figura 3 - Estados de un Processor 16
Figura 4 - Esquema de la aplicación original 21
Figura 5 - Esquema de las 3 configuraciones propuestas 22
Figura 6 - Funcionamiento básico de la aplicación de videoconferencia 27
Figura 7 - Configuración sin clonación: una sola posibilidad 28
Figura 8 - Configuración genérica con clonación 29
Figura 9 - Primera configuración considerada para el sistema 30
Figura 10 - Esquema de funcionamiento del servidor en la 1ª configuración 31
Figura 11 - Reproducción perfecta. No faltan frames. 33
Figura 12 - Reproducción correcta ante la falta de frames 33
Figura 13 - Reproducción incorrecta ante la falta de frames 34
Figura 14 - Segunda configuración considerada para el sistema 35
Figura 15 - Funcionamiento de cliente y servidor en la segunda configuración del sistema 35
Figura 16 - Posibles puntos de clonación tras la captura 36
Figura 17 - Sincronización del inicio de la grabación 40
Figura 18 - Segmentación del procesado para hacer posible la clonación 49
Figura 19 - Resincronización NTP en paralelo a la videoconferencia 55
Figura 20 - Resincronización NTP previa a la videoconferencia 56
Figura 21 - Retardo de frame en una reproducción (relativamente) correcta con JMF 73
Figura 22.- Retardo de frame en una reproducción normal con JMF 74
Figura 23.- Detalle del retardo de frame "de 2º orden" de una reproducción normal con JMF 75
Figura 24.- Otro ejemplo de retardo de frame, a partir del mismo fichero 76
Figura 25.- Detalle del retardo de frame "de 2º orden" del 2º ejemplo 77

Índice de tablas

Tabla 1.- Resumen del estado inicial del uso de funcionalidades de JMF en el COMM 26
Tabla 2.- Resumen de resultados respecto al JMF 82
Tabla 3.- Resumen de resultados respecto al Sistema de Videoconferencia 82

Capítulo 1

Introducción y Objetivos

El presente Proyecto Fin de Carrera investiga el posible uso de las librerías Java Media Framework de Sun, ya usado en un sistema de videoconferencia del COMM, para añadir a dicho sistema nuevas funcionalidades relacionadas con la grabación y reproducción de flujos multimedia.

Hasta el momento, el uso de la librería JMF ha dado muy buenos resultados, pero también un creciente número de problemas. El grupo de desarrollo ha llegado pues a un momento en que debe plantearse si tiene sentido seguir evolucionando con el JMF o si, por el contrario, sería mejor buscar nuevas alternativas. Para ello es necesario conocer con detalle las limitaciones prácticas del JMF y saber así las aplicaciones que posibilita o que se hacen impracticables.

El aspecto práctico de este estudio, y final deseable del proyecto, será la implementación en el sistema existente de las funcionalidades de grabación de la videoconferencia, y la posterior reproducción simultánea y sincronizada de las grabaciones de cada uno de los participantes.

El sistema de videoconferencia ya existente en el COMM

El grupo de Comunicaciones Multimedia (COMM) había desarrollado ya un sistema de videoconferencia en Java. Se trata de un programa desarrollado usando JMF y un codec MPEG4 implementado como librería nativa (proyecto open source XVID), lo

que le da buenas prestaciones de velocidad y ancho de banda requerido. La transmisión se lleva a cabo por RTP.

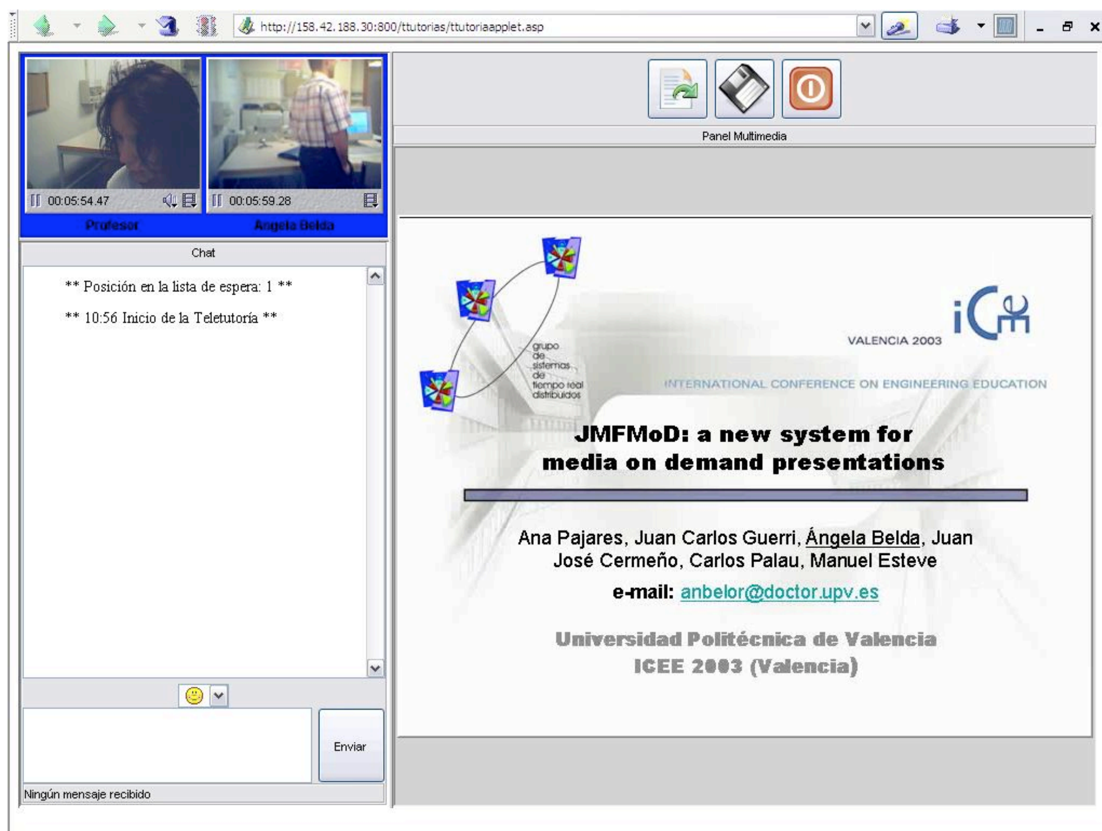

Figura 1.- Interfaz de la aplicación de videoconferencia del COMM

Para algunas de las posibles aplicaciones de este sistema, sería conveniente tener la posibilidad de grabar y reproducir flujos multimedia. De ahí el propósito de este Proyecto Fin de Carrera.

Objetivo del PFC

El objetivo por tanto es explorar las capacidades del JMF y aplicarlas para implementar la funcionalidad requerida: grabación de la videoconferencia y posterior reproducción sincronizada de las diferentes grabaciones. La grabación también se aplicará a la captura de pantalla, que es otra funcionalidad del sistema de videoconferencia.

Por ello no sólo habrá que estudiar el JMF, sino que habrá que plantear si la estructura supuesta en un uso normal del sistema inicial de videoconferencia es aceptable para soportar las nuevas funcionalidades:

- Normalmente hay dos equipos desarrollando la videoconferencia. Se estudiará si se podrían aprovechar para la grabación, y de qué forma. Algunas de las cuestiones serán: ¿Un equipo lo graba todo, o cada uno graba una parte? ¿Qué parte, la local (emitida por el equipo) o la remota (recibida por el equipo)? ¿Sería mejor tener un nuevo equipo exclusivamente dedicado a la grabación?

- En el transcurso normal de una videoconferencia ya se está comprimiendo a MPEG4 el video y a una variedad de formatos el audio (GSM, G.723, por ejemplo). Pero son codificaciones pensadas para la transmisión por red, sobre RTP. En algunos de los escenarios nombrados se podría plantear una codificación alternativa para la grabación, para conseguir mayor calidad sin la restricción de ancho de banda de la red; o al contrario, para conseguir una grabación de pequeño tamaño como mero recordatorio del contenido de la videoconferencia.

- ¿En qué formato de fichero contenedor se guardarán las grabaciones? JMF ofrece soporte para los formatos AVI (de Microsoft) y MOV (Quicktime, de Apple), cada uno con diferentes capacidades, diferente soporte de codecs y diferentes calidades de implementación en el JMF. Además, durante el desarrollo de la aplicación de videoconferencia del COMM, se desarrolló un formato propio (llamado AVS) para superar algunos de los problemas que se encontraron en el uso del JMF; aunque el nuevo formato también planteaba sus propios problemas.

Todos estas posibilidades debían ser consideradas para encontrar un buen compromiso entre diferentes factores: rendimiento, carga de los equipos, calidad de video y audio, facilidad de implementación, de despliegue y de uso.

Contenido del resto de esta memoria

La presente memoria se encuentra dividida en siete capítulo, de los cuales esta introducción constituye el primero. Los restantes seis capítulos abordan los contenidos introducidos con la siguiente distribución:

Capítulo 2.- Tecnologías implicadas

Ofrece una mínima introducción a las diferentes tecnologías implicadas en el proyecto: Java, Java WebStart, JMF, Xvid / MPEG4, RTP.

Capítulo 3.- Decisiones de diseño

Explora las diferentes posibilidades que se presentan a la hora de plantear la implementación de las nuevas funcionalidades pretendidas: desde la configuración del sistema general de videoconferencia hasta los formatos de grabación usables desde JMF.

Capítulo 4.- Desarrollo

Detalla los pasos seguidos durante la implementación de las soluciones seleccionadas, perfilando mientras tanto la frontera entre lo factible y lo problemático en el uso de JMF.

Capítulo 5.- Detalles de desarrollo e implementación

Capítulo dedicado a los detalles a más bajo nivel del trabajo desarrollado, tanto en la implementación de nuestras aplicaciones como internamente al propio JMF.

Capítulo 6.- Conclusiones y trabajo futuro

Se da un resumen de lo conseguido en el presente Proyecto Fin de Carrera, y se plantean los cursos de acción más interesantes según la experiencia acumulada hasta el momento.

Capítulo 7.- Bibliografía.

Capítulo 2

Tecnologías implicadas

Para entender el entorno en que nos movemos y a qué pretendemos llegar, conviene explicar someramente las tecnologías implicadas y las decisiones de diseño tomadas hasta el momento. Como se ha indicado, al fin y al cabo no partimos de cero sino que la meta es añadir funcionalidad a un sistema ya existente e incluso relativamente maduro.

Java

El lenguaje de programación Java es sobradamente conocido. Sólo recordaremos aquí las características más importantes para nuestras aplicaciones y para el desarrollo de los objetivos del presente Proyecto Fin de Carrera:

— Java es altamente portable. En teoría, "*write once, run anywhere*"; aunque en nuestro caso el uso de librerías nativas reduce en parte esa libertad. De todas formas, es una gran ventaja tener la posibilidad de desplegar una misma base de código en varias plataformas (Windows y Linux por el momento).

— Lo que permite la alta portabilidad es que Java es compilado a un *bytecode*, que para ser ejecutado necesita de un intérprete (llamado *Java Virtual Machine*) específico para cada plataforma. Se puede decir que el trabajo de compilar para cada plataforma ya ha sido hecho por los implementadores de cada JVM. Además, la JVM provee otros servicios al programador; como

ejemplo notable, un *Garbage Collector* (recolector de basura), que se encarga de liberar la memoria que ya no está siendo usada por los programas.

— Java incluye facilidades para hacer interfaz con librerías nativas. Dado que una característica diferenciadora del sistema de videoconferencia a ampliar es el uso de MPEG4 para la transmisión por RTP, y la codificación es llevada a cabo por una librería nativa, esta posibilidad es vital para nosotros.

— JNLP / Java WebStart. Es parte de la especificación Java, pero suficientemente poco conocido y sin embargo significativo como para dedicarle una pequeña explicación aparte.

JNLP / Java WebStart

Cualquier aplicación necesita ser instalada en el lugar donde va a ser ejecutada; más aún si depende de librerías externas, como es nuestro caso. Las posibilidades de instalación y posterior mantenimiento de esa instalación (actualización de la aplicación y librerías, prevención de conflictos con otras librerías instaladas anteriormente o posteriormente) dependerán de muchos factores: tipo de aplicaciones y librerías, cantidad de dependencias, infraestructura del Sistema Operativo, existencia de infraestructuras adicionales para este mantenimiento (gestores de instalación y actualización), etcétera. Pero se puede generalizar y afirmar sin miedo a equivocarnos que normalmente es un auténtico quebradero de cabeza para desarrolladores y usuarios. Compárese la diversidad de entornos considerando por ejemplo:

— una distribución Linux con un gestor de paquetes (que aún incluye variantes, como Debian con su `apt-get` o Gentoo con su `portage`),

— Windows (que pese a usar desde siempre instaladores y desinstaladores completos ha necesitado una década para olvidar el "*DLL hell*"), o

— Mac OS X, que procura limitar la instalación de aplicaciones a un simple “arrastrar y soltar”, aislando al usuario e incluso al desarrollador de toda la problemática (a costa posiblemente de consumir más espacio de disco).

La forma de funcionamiento de Java amenazaba con añadir un requisito más para poder instalar una aplicación Java: la instalación de su Máquina Virtual. Pero Sun hábilmente añadió a la plataforma la posibilidad de no sólo instalar automáticamente la JVM, sino que esa misma funcionalidad se puede usar para encapsular cualquier aplicación Java y sus librerías (ya sean Java o nativas), de forma que prácticamente cualquier aplicación Java puede ser lanzada haciendo click en un enlace de una página web.

JNLP (*Java Network Launching Protocol*) es la especificación del protocolo que hace ésto posible, y Java WebStart es la implementación que Sun provee. WebStart se ocupa entonces de todo lo necesario para que una aplicación Java pueda funcionar:

— Reúne sus librerías, ya sea en la misma descarga o en descargas aparte que serán llevadas a cabo automáticamente según hagan falta;

— Las “instala” de forma que la aplicación las pueda encontrar, pero sin que puedan entrar en conflicto con otras aplicaciones o incluso con otras versiones del mismo programa;

— Provee un entorno de seguridad (*sandbox*) configurable por el usuario y el desarrollador, que puede limitar el acceso a recursos del sistema, protegiendo su integridad y la privacidad del usuario;

— E incluso se encarga de actualizar la aplicación si nuevas versiones aparecen, de instalarla para que sea accesible como una aplicación normal del OS nativo (si el usuario así lo elige), y hasta de desinstalarla llegado el caso.

Las ventajas que todo esto trae para cualquier aplicación pensada para la red, como es precisamente nuestro sistema de videoconferencia, son enormes. Lo que normalmente hubiese sido un proceso de instalación de la JVM, instalación de las librerías nativas,

instalación de la aplicación, y por fin empezar la videoconferencia, queda convertido en simplemente visitar una página web y hacer click en el enlace adecuado. Y reduce por tanto enormemente la "barrera de entrada": cualquier usuario, sin importar su capacidad técnica o la complejidad de su entorno, puede usar el sistema sin tener que pensárselo dos veces.

Java Media Framework (JMF)

JMF es el API desarrollado por Sun para añadir capacidades multimedia a Java, y es la infraestructura que hace posible nuestras aplicaciones; también será lo que ocupe la práctica totalidad de este Proyecto Fin de Carrera.

La primera versión del JMF fue publicada en 1997 y sólo proveía capacidades de reproducción y lectura (desde ficheros locales y protocolo HTTP). Fue implementada independientemente por Silicon Graphics, Intel y Sun.

En su segunda versión (y última hasta el momento), de 1999, se añadían capacidades de recodificación, captura, grabación de ficheros y transmisión y recepción RTP, todo ello basándose en una arquitectura de *plug-ins* que permiten al programador añadir sus propios codificadores y procesadores para tratar los flujos multimedia de cualquier forma. Esta versión fue implementada conjuntamente por IBM y Sun.

JMF está disponible en una versión *Pure Java*, usable en cualquier plataforma que disponga de una JVM versión 1.1 o superior. Aparte, hay tres versiones específicas más, que añaden librerías nativas (llamados *Performance Packs*) para aprovechar facilidades de los Sistemas Operativos Solaris, Linux y Windows; sólo éstas versiones son capaces de capturar video, ya que Java necesita esas librerías nativas para tener acceso a las APIs o el hardware específico de captura de video de cada OS. Además los *Performance Packs* permiten el acceso a más funcionalidades de cada OS, como puede ser hardware de sonido de Sun en Solaris o codecs adicionales instalados en la arquitectura de Windows Media en los OS de Microsoft.

Definiciones previas

Tipo de contenido, o formato contenedor, es el formato en el que se almacenan los datos multimedia. QuickTime, MPEG y WAV son ejemplos de tipos de contenido. Podría considerarse sinónimo de tipo de archivo. Se utiliza este término porque los datos se obtienen a menudo de otras fuentes además de los archivos locales.

Un **flujo multimedia** lo forman los datos obtenidos de un archivo local, adquiridos a través de la red o capturados de una cámara o micrófono. Los flujos multimedia pueden contener varios canales llamados **pistas** (*tracks*), frecuentemente una pista de vídeo y otra de audio. Los flujos multimedia que contienen más de una pista son flujos multiplexados o complejos. Demultiplexar es el proceso de extraer las pistas individuales de un flujo multimedia complejo.

El **tipo de una pista** es la clase de datos que contiene, como audio o vídeo. El formato de una pista define cómo se estructuran los datos de dicha pista.

Un flujo multimedia puede identificarse utilizando su localización y el nombre del protocolo que se va a utilizar para acceder al mismo.

Modelo Temporal: La interfaz *Clock* define la temporización básica y operaciones de sincronización que se necesitan para controlar la presentación de datos multimedia. Las clases de JMF que requieren este tipo de especificaciones implementan la interfaz *Clock*.

Eventos: JMF utiliza un mecanismo de notificación de eventos para mantener a los programas informados del estado actual del sistema multimedia. Cuando un objeto JMF necesita informar de las condiciones actuales produce un *MediaEvent*. Para cada tipo de objeto JMF que puede producir un *MediaEvent*, se define una interfaz *listener* correspondiente

Formato de datos: El formato exacto del contenido multimedia de un objeto se representa con un objeto Format. El propio *Format* no transporta por sí mismo

parámetros específicos de la codificación o información temporal global, sino que describe el nombre de la codificación del formato y el tipo de datos que requiere el formato. JMF extiende la clase *Format* para definir formatos específicos de audio y vídeo.

Obtención de flujos multimedia a partir de un fichero.

Para obtener una referencia al contenido de un fichero multimedia que pueda ser utilizada por la librería JMF con el fin de realizar un procesado posterior necesitamos crear un objeto *DataSource*. Un *DataSource* representa una fuente de datos y encapsula la localización del contenido multimedia, así como el protocolo utilizado para obtener dicho contenido.

JMF define varios tipos de objetos *DataSource* (*PullDataSource*, *PushDataSource*, *MergingDataSource*...). El que nos interesa para extraer los datos multimedia de un fichero local es un objeto *PullDataSource* y el protocolo que utilizaremos para acceder a estos datos es el protocolo FILE.

Un *DataSource* se identifica bien por un *MediaLocator* JMF o bien por un URL (*Universal Resource Locator*). Un *MediaLocator* es similar a un URL y puede instanciarse a partir de uno, pero puede construirse incluso si el correspondiente manejador de protocolo no está instalado en el sistema.

Procesado del contenido multimedia

Objetos Controller

En JMF la interfaz *Controller* define el estado básico y el mecanismo de control para un objeto que controla, presenta o captura datos multimedia en tiempo real. También define las fases que un controlador multimedia atraviesa y proporciona un mecanismo para controlar las transiciones entre esas fases. Algunas de las operaciones que deben realizarse antes de que los datos multimedia de salida estén disponibles pueden

consumir tiempo, por lo que JMF proporciona control programático sobre ellas cuando ocurren.

Un *Controller* lanza una variedad de *MediaEvents* específicos de un controlador para proporcionar notificación de cambios en su estado. Para recibir eventos de un *Controller* se implementa la interfaz *ControllerListener*.

El API JMF define dos tipos de *Controllers*: *Players* y *Processors*, los cuales se construyen para una fuente de datos particular.

Players y Processors

Un *Player* procesa un flujo de entrada de datos multimedia y lo reproduce en un tiempo preciso. Se utiliza un *DataSource* para entregar el flujo de datos al *Player*. El destino de reproducción depende del tipo de media que se presente.

Un *Processor* es un tipo especializado de *Player* que proporciona control sobre el procesado que se realiza en el flujo multimedia de entrada. Un *Processor* toma un *DataSource* como entrada, realiza algún procesado definido por el usuario en los datos multimedia y obtiene como salida los datos media procesados. Un *Processor* puede enviar datos de salida a un dispositivo de presentación o a un *DataSource*. Ese *DataSource* puede usarse como entrada de otro *Player* o *Processor*, o puede utilizarse para obtener los flujos multimedia que se transmitirán en sesiones RTP.

Un *Processor* permite al programador definir el tipo de procesado que se aplica a los datos media. Esto permite la aplicación de efectos, mezclado y composición en tiempo real.

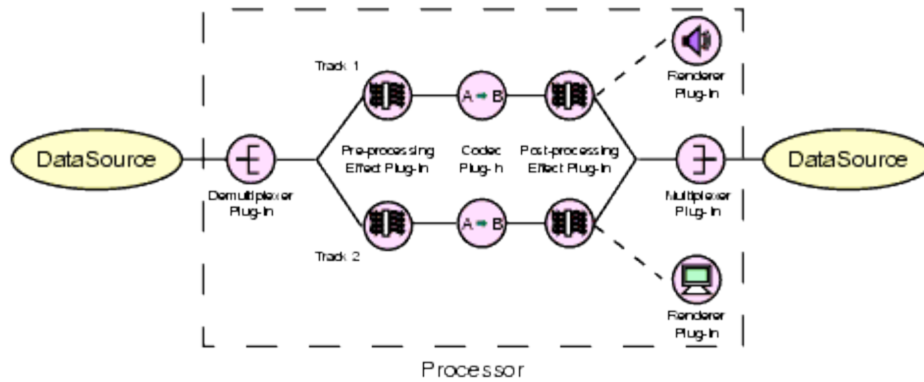

Figura 2 - Etapas del Processor

Aunque un *Processor* permite realizar un Pre-procesado y un Post-procesado, antes y después, respectivamente, de la transcodificación, no profundizaremos en esas etapas por no ser relevantes en la elaboración del presente proyecto. Con esta consideración, el procesado de los datos se divide en las siguientes etapas:

— **Demultiplexación:** proceso de separar el flujo de entrada, si éste contiene múltiples pistas, que se extraen y procesan separadamente.

— **Transcodificación:** proceso de convertir cada pista de datos multimedia de un flujo de entrada de un formato a otro. Un *Codec* realiza la compresión y descompresión de datos multimedia. Cuando se codifica una pista, se convierte a un formato comprimido adecuado para el almacenamiento o la transmisión; cuando se descodifica se convierte a un formato no-comprimido (bruto, *raw*) adecuado para la presentación.

— **Multiplexación:** proceso de intercalar las pistas codificadas a un único flujo de salida. Se puede especificar el tipo de datos del flujo de salida con el método `setOutputContentDescriptor` del *Processor*.

— **Reproducción:** proceso de presentar los datos multimedia al usuario. El procesado de cada etapa lo realiza un componente de procesado distinto. Estos componentes de procesado son plug-ins JMF. Existen cinco tipos de plug-ins JMF: *Demultiplexer* (extraer pistas), *Effect* (aplicar algoritmos de efecto en pre y post-procesado), *Codec* (codificación y decodificación de los

datos), *Multiplexer* (combinar pistas) y *Renderer* (procesar datos de una pista y enviarlos al destino).

Estados de un Player y un Processor

Un *Player* presenta seis estados posibles. La interfaz *Clock* define los dos estados primarios: *Stopped* y *Started*. Para facilitar la gestión de recursos, la interfaz *Controller* divide el estado *Stopped* en cinco estados de standby: *Unrealized*, *Realizing*, *Realized*, *Prefetching* y *Prefetched*.

Un *Processor* tiene dos estados de standby adicionales, *Configuring* y *Configured*, que suceden antes de que el *Processor* entre en el estado *Realizing*.

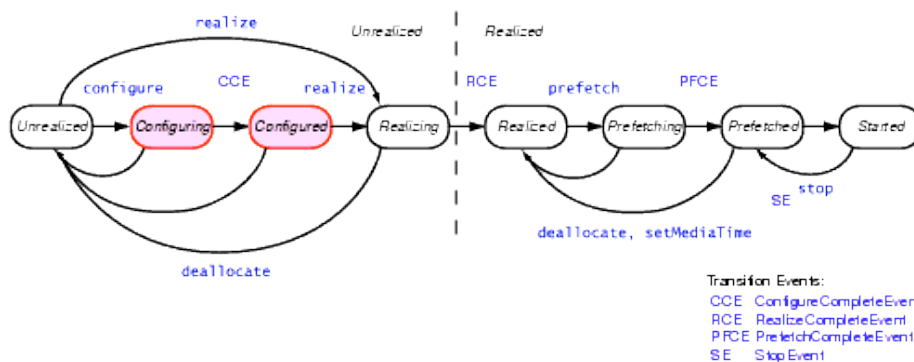

Figura 3 - Estados de un Processor

En una operación normal, un *Player* o *Processor* pasa por cada estado hasta que alcanza el estado *Started*, siguiendo el siguiente proceso:

— Cuando un *Player* o *Processor* se crea está en el estado *Unrealized*. Para crear un *Player* o *Processor* se necesita un URL, un *MediaLocator* o un *DataSource* que referencie el contenido multimedia que pretendemos procesar.

— Al llamar al método `configure()`, el *Processor* pasa al estado *Configuring*. Aquí se conecta al *DataSouce*, demultiplexa el flujo de entrada y accede a la información sobre los formatos.

— El *Processor* pasa al estado *Configured* cuando completa las operaciones descritas y lanza un evento *ConfigureCompleteEvent*. Mientras el *Processor* está en el estado *Configured*, se puede llamar al método **`getTrackControls()`** para conseguir los objetos *TrackControls* de las pistas individuales del flujo multimedia. Estos objetos permiten especificar las operaciones de procesado que se desean realizar sobre cada pista. Si se conoce el formato de datos de salida que se desea para cada pista, se puede usar el método **`setFormat(Format fmt)`** para especificar el *Format* y dejar que el *Processor* escoja un codec apropiado.

— Cuando se llama a **`realize()`** el *Player* o el *Processor* pasa al estado *Realizing*, donde determina sus requerimientos de recursos y adquiere los recursos de uso no exclusivo que sólo necesita adquirir una vez.

— Cuando se completan las operaciones anteriores el *Player* o *Processor* pasa al estado *Realized*. En este punto sabe qué recursos necesita y tiene información del tipo de datos a presentar. A partir de este estado, el *Player* o *Processor* puede proporcionar componentes de control y visuales para la reproducción del contenido multimedia.

— Al llamar al método **`prefetch()`**, se pasa al estado *Prefetching*. Sólo llamaremos a éste método si queremos reproducir el *DataSource*. En el estado *Prefetched* el *Player* o *Processor* está listo para empezar. El método **`start()`** pone en marcha la reproducción.

Un *Player* no proporciona ningún control sobre el procesado que realiza o cómo reproduce los datos multimedia. El cliente utiliza objetos *Player* para la reproducción de los flujos recibidos. Estos *Players* se encargan del procesado y reproducción de los datos multimedia.

Librería Xvid

Xvid es el nombre de un proyecto open source cuya meta es crear una librería de codificación y decodificación de video según el estándar MPEG4. Surgido como

derivación de los comienzos open source del famoso codec comercial DivX, Xvid ha acabado presentándole una seria competencia; no sólo por su calidad como mínimo comparable, sino por algunas ventajas concretas, como el ser usable en un gran número de plataformas y no estar guiado en su evolución sólo por intereses comerciales, lo que permite la aparición de características innovadoras.

Como se ha dicho, una de las grandes mejoras de JMF en su versión 2 es haberse convertido en una arquitectura basada en *plug-ins*. Y esto es lo que se ha aprovechado en el COMM para crear un *wrapper* Java alrededor de la librería Xvid, aportando la capacidad de codificación y decodificación MPEG4 a velocidades nativas en nuestros programas. Esto nos permite conseguir flujos de video de buena calidad y pequeño *bitrate*, siendo esto en todo caso configurable para cambiar un número de parámetros que afectarán tanto a la calidad como al *bitrate*.

Así como los estándares de codificación de video MPEG1 y MPEG2 tenían un campo de acción bastante específico, el estándar MPEG4 está pensado para abarcar un amplio rango de aplicaciones; desde el video robusto y a muy bajo *bitrate* necesario para entornos móviles (pantallas pequeñas, poca capacidad de proceso, limitado ancho de banda, probable corrupción del flujo de datos) hasta sistemas de video de alta calidad (*broadcast* con canales dedicados, de alto ancho de banda y buena fiabilidad).

Como los anteriores estándares, un flujo de video MPEG4 se estructura con series de *frames* de diferentes tipos:

- I: Intra, independientes de todas las demás, las que más información aportan y que permiten al decodificador comenzar a funcionar. Son necesarias para permitir acceso aleatorio al flujo.
- P: Predictivas, basadas en la información de *frames* anteriores, y
- B: Bidireccionales, las que menos bytes necesitan, aprovechando para su codificación información proveniente de las *frames* anteriores y posteriores en el flujo.

La forma en que se distribuyen estos *frames*, así como los parámetros con los que se codifican, definen las características del flujo de video y las capacidades necesarias por parte del decodificador. Para simplificar el gran rango de configuraciones posibles, el estándar MPEG4 define un número de conjuntos de parámetros (*profiles*, o perfiles de aplicación) según el destino del flujo: capacidades del decodificador, características del canal, *bitrate* disponible...

La librería Xvid soporta toda esta libertad de configuración, y el wrapper desarrollado en el COMM procura hacerla disponible desde Java siguiendo los protocolos del JMF; de forma que es posible usar transparentemente buena parte de la funcionalidad de Xvid desde JMF sin tener que salir de su API estándar.

RTP (*Real-time Transport Protocol*)

La última pieza que define el sistema de videoconferencia del COMM es el protocolo RTP (y su protocolo auxiliar RTCP, *RTP Control Protocol*), desarrollados por el grupo *Transport Audio/Video* del IETF y especificados en las RFCs (*Request For Comments*) 1889 y 1890. Ambos son protocolos de nivel de aplicación que se encapsulan sobre el protocolo de transporte de la arquitectura de red subyacente. Nuestro sistema de videoconferencia supone redes con arquitectura IP, por lo que RTP y RTCP se encapsulan en el protocolo UDP (*User Datagram Protocol*).

La aparición de la pareja de protocolos RTP/RTCP vino dirigida a posibilitar la transmisión de audio y vídeo incluso en redes en las que no se garantiza una calidad de servicio (QoS), proporcionando servicios de identificación del tipo de carga útil, numeración de secuencia, estampado de tiempo y monitorización de entregas.
Las aplicaciones para las que fueron desarrollados estos protocolos fueron aplicaciones de tiempo real (Videoconferencias, difusión de video y audio, telefonía sobre IP). RTP también soporta la transferencia de datos a múltiples destinos usando distribución *multicast*, si ésta es soportada por la red de transporte utilizada. Además aporta sincronización de flujos multimedia y adaptación de la reproducción al estado de la red. La pareja de protocolos RTP/RTCP transporta la información necesaria para identificar la codificación de los flujos, reconstruir la base de tiempos de los flujos de

audio y vídeo, sincronizar los flujos multimedia, reconstruir la secuencia de paquetes del emisor, detectar pérdidas de paquetes y variaciones del retardo. Estas funciones suplen la falta de robustez implícita en el uso de UDP como protocolo de transporte.

RTP se encarga de la transmisión de paquetes junto con información para su reproducción e identificación del contenido, mientras que RTCP (*Real-Time Control Protocol*) se encarga de gestionar y monitorizar la sesión RTP.

Hay que decir que JMF soporta la transmisión de flujos sobre TCP, y de hecho permite gestionar los flujos sobre cualquier protocolo que podamos implementar. Pero, mientras que esto podría ser útil en ciertos escenarios muy concretos, la utilidad y preferencia de RTP/RTCP para nuestros objetivos es evidente.

Capítulo 3

Decisiones de diseño

La aplicación existente

Para tener una referencia de la situación inicial del sistema de videoconferencia, aquí tenemos un esquema de su uso (Figura 4): dos equipos, en el que uno es servidor y otro es cliente. El cliente se conecta al servidor, que puede aceptar la conexión de varios clientes y mantener conversaciones de tipo texto con todos. Cuando el servidor lo decida, comienza la videoconferencia y se transmiten los flujos media por RTP.

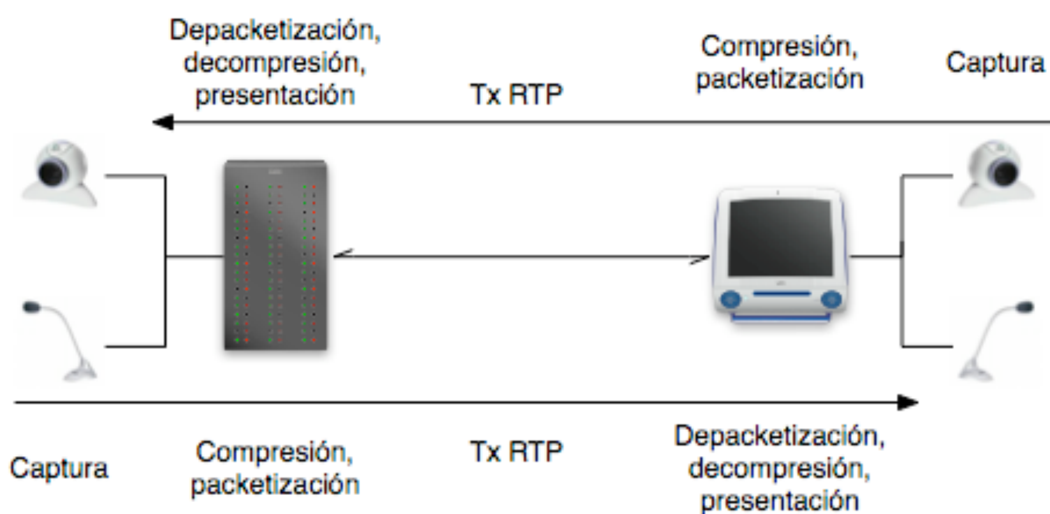

Figura 4 - Esquema de la aplicación original

Posibles nuevas configuraciones

Veamos ahora las posibles variaciones que hemos contemplado para implementar la grabación de la videoconferencia (ordenadas, como veremos, por orden de preferencia) (Figura 5):

- Primera opción: un equipo participante graba todos los flujos. Lo más natural es que el equipo participante sea el servidor.
- Segunda opción: cada equipo participante graba el flujo que está generando.
- Tercera opción: un equipo extra, dedicado, se encarga de grabar todos los flujos.

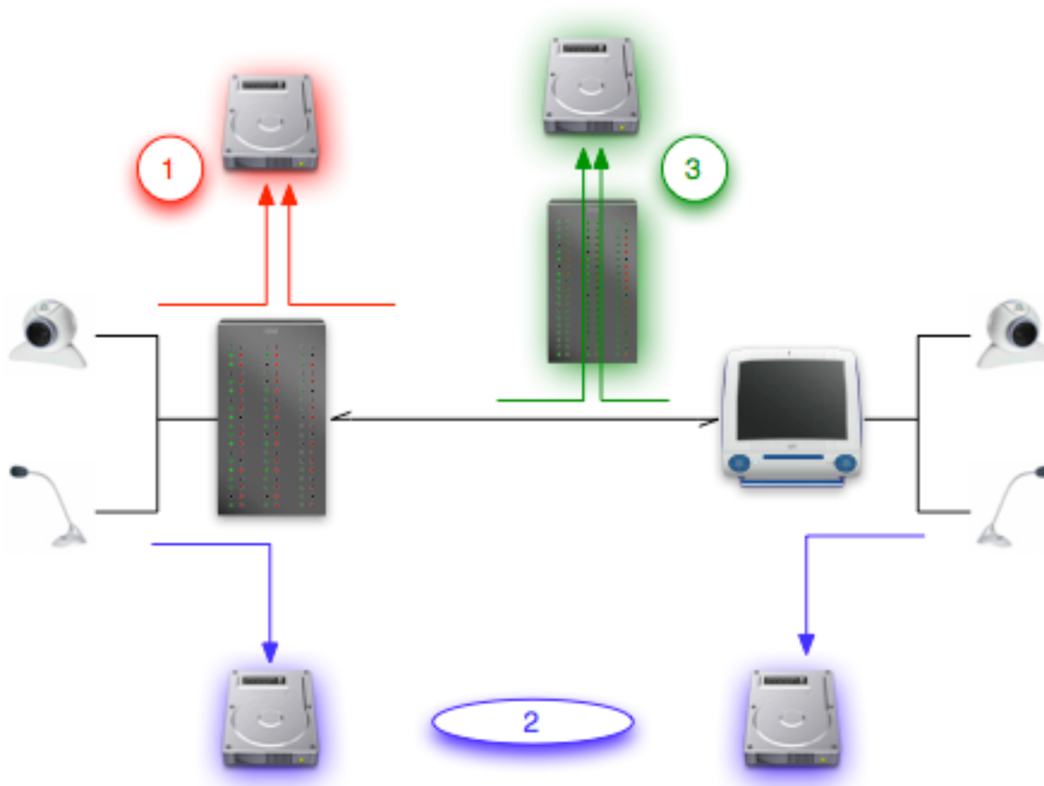

Figura 5 - Esquema de las 3 configuraciones propuestas

La **tercera opción** sería lógicamente la más escalable, en el sentido de que el equipo dedicado evitaría cargar más a los equipos que participan en la videoconferencia. Podría encargarse de la grabación de más usuarios (para un posible uso futuro de varios usuarios simultáneos) y con diferentes calidades. Pero evidentemente exige un

equipo extra, una configuración adecuada de la red y modificar cliente y servidor de la videoconferencia para que envíen los flujos al equipo grabador. Por todo ello no nos ha parecido pertinente para el uso esperado del sistema de videoconferencia.

La **primera opción** es la ideal: no requiere cambios en el cliente, sino que queda totalmente contenida en el servidor. El operador del servidor tiene el control de la grabación y dispone de los resultados de la grabación inmediatamente, con lo que los problemas de mantenimiento de la grabación, seguridad, implementación, despliegue, etcétera quedan limpiamente soslayados.

Hay que destacar el importante matiz de que la primera opción requiere grabar tanto la recepción del flujo RTP como la captura local de audio y video; en la segunda opción se graban sólo flujos capturados localmente; y en la tercera se graban sólo flujos recibidos por RTP. Dado que cada tipo de grabación pide una aproximación específica y dará unos resultados diferentes, habrá que tenerlo en cuenta.

Una de las características distintivas del sistema de videoconferencia es el uso de codificación MPEG4 para los flujos de video. Este codec es para codificación con pérdidas; así que una vez codificado, recodificar sólo tendría sentido para reducir la calidad. Además, un flujo de video recibido por RTP puede haber sufrido pérdida de paquetes, con lo que la calidad empeoraría aún más.

Así que la única crítica que se puede hacer a la primera opción nombrada es que la calidad de los flujos generados por el servidor se mantendrá intacta en la grabación, mientras que los flujos recibidos de los clientes estará limitada por la codificación con pérdidas a la que han sido sometidos para su transmisión. Aunque se puede argumentar que esa es la calidad que estaba percibiendo el operador del servidor durante la videoconferencia.

Quedaría la duda de si el servidor soportaría la carga de las nuevas tareas requeridas para la grabación. El servidor original no cargaba destacablemente el procesador de cualquier equipo medianamente reciente; y en realidad la nueva carga es mínima, ya que en principio no es necesario recodificar el video y audio, sino que se puede volcar tal como viene (ya comprimido) en ficheros del formato contenedor elegido.

Dicho todo esto respecto de la primera opción, queda por calificar la **segunda opción**; que es interesante, puesto que al encargar a cada participante de su propia grabación, aporta escalabilidad (ningún equipo queda más cargado que los demás) y calidad (ya que todas las grabaciones son locales, sin pérdidas de paquetes).

Pero esta configuración pierde en facilidad de administración: al acabar la grabación, los ficheros resultantes estarán desperdigados en cada uno de los equipos participantes; y posiblemente perdemos control sobre las especificaciones de los equipos que graban, ya que cualquier cliente que quiera participar en una grabación deberá soportar su propia carga (mientras que en la primera opción, todo dependía del servidor).

El problema de que los ficheros resultantes estén desperdigados se puede solucionar fácilmente en el propio programa de videoconferencia, por ejemplo haciendo que el participante mande su grabación a un repositorio centralizado de forma transparente al acabar de grabar; pero el hecho es que es una complicación extra.

Por todo ello, esta segunda opción nos pareció interesante pero menos que la primera. Con lo que la primera configuración quedó elegida como solución preferente a ser implementada.

Formatos contenedores de la grabación

Otra elección que hacer se refería a los formatos contenedores disponibles para las grabaciones de flujos, cada uno con sus ventajas e inconvenientes.

JMF ofrece soporte para el formato AVI. Es un formato muy conocido, por ser de el primero popularizado en Windows por Microsoft. Curiosamente es un formato bastante limitado, cuyo soporte para muchas características de los codecs modernos de video es inexistente, limitado o dependiente de extensiones no oficiales del formato. Por ejemplo, el formato no soporta (oficialmente) codecs con *frames* B, ni *framerate* variable (lo cual importante si se pretende grabar una fuente que puede de

repente perder *frames*, como es una recepción RTP). La propia Microsoft recomienda desde hace años abandonar AVI para usar WMV en su lugar.

JMF también soporta el formato MOV, conocido por ser el formato usado por la arquitectura Quicktime de Apple. Se trata de un formato diseñado para ser extensible, lo que explica que haya servido precisamente de base para el desarrollo del contenedor estándar del formato MPEG4. Como ejemplo, este formato si que soporta *frames* B y *framerate* variable, pese a ser cronológicamente anterior a AVI.

Estos dos formatos son estándares conocidos, así como los codecs con los que codificamos el audio y el video (GSM/G.723 y MPEG4); lo cual nos debería permitir usar una variedad de reproductores para luego poder reproducir nuestras grabaciones. Pero en la práctica aparecen un número de problemas, algunos motivados por el propio JMF (implementación incompleta o dudosa de algunas características) y otros por el uso de los formatos en la vida real (por ejemplo, puede ser difícil encontrar un codec Xvid para el Quicktime oficial en plataformas Windows). Siempre está la opción de crear nuestro propio reproductor aprovechando el mismo JMF; pero si nos vemos obligados a usar nuestro propio reproductor, poco justificación tiene entonces usar un formato estándar.

Además los formatos AVI y MOV dieron problemas durante el desarrollo de la aplicación de videoconferencia, lo que provocó que se desarrollaran en el COMM unos *plug-ins* para JMF (de/multiplexor) para implementar un formato contenedor propietario para nuestro uso interno, llamado AVS. Este formato volcaba los flujos de audio y video a disco con la información extra de temporización propia de un flujo RTP; de forma que al reproducir un fichero AVS, el JMF se comporta como si recibiese un flujo RTP. Con el AVS se consiguieron solucionar algunos de los problemas que se habían encontrado antes (principalmente relacionados con corrupción de *frames* y con desincronías entre audio y video), pero quedaban otros problemas (relacionados con la disonancia entre un flujo tratado como RTP aunque es leído desde un fichero: capacidades de seeking, reinicio de la reproducción...).

Así que AVS era el formato más prometedor para solucionar ciertos graves problemas, pero al ser un formato propio limitaría luego la utilidad de las grabaciones

conseguidas. Por otra parte, no estaba clara la razón por la que MOV y AVI no grababan correctamente. Con lo que se partía de la idea de que lo más interesante era continuar el trabajo con AVS, pero había que re-explorar los otros dos formatos (MOV principalmente).

En todo caso, la elección de formatos contenedores es algo abstraíble del resto del proyecto, ya que cambiar entre formatos apenas implicaba (idealmente) cambios al código tanto de grabación como de reproducción. Más bien, lo importante era conocer qué opciones y problemas nos daba cada formato.

Resumen

Podemos entonces resumir el propósito de este PFC y la situación inicial con esta pequeña tabla:

Funcionalidad de JMF	Status en JMF original	Status en COMM
Captura	OK	En uso
De/codificación	Pocos codecs	Añadido MPEG4 (Xvid)
Transmisión RTP	*Timestamps* incorrectos	Corregido
Recepción RTP	OK	En uso
Reproducción	OK	En uso
Clonación de flujos	**Problemas (por especificar)**	
Grabación	**Problemas (por especificar)**	

Tabla 1.- Resumen del estado inicial del uso de funcionalidades de JMF en el COMM

Recordando que las mejoras a investigar darán forma a las nuevas funcionalidades de grabación y reproducción simultánea y sincronizada del sistema existente de videoconferencia.

Capítulo 4

Desarrollo

Preliminares

Como se ha visto, JMF ofrece APIs para captura de audio y video, para recepción de flujos RTP, y para el tratamiento de los flujos resultantes (codificación, reproducción, transmisión RTP, grabación).

El proyecto comenzó con la creación de pequeños programas de prueba que reproducían la forma en que se usa el JMF en el sistema de videoconferencia del COMM (Figura 6).

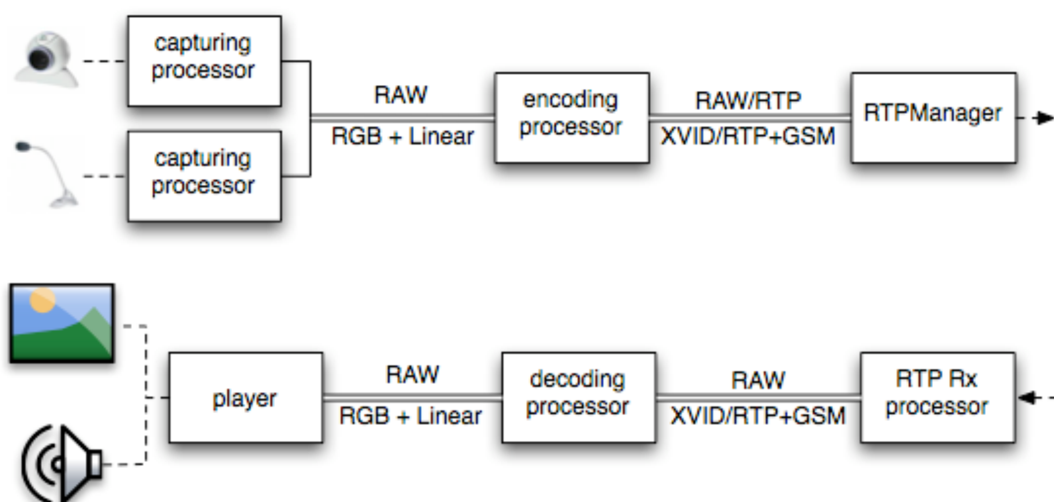

Figura 6 - Funcionamiento básico de la aplicación de videoconferencia

JMF captura recurriendo a las APIs de captura de audio y video del Sistema Operativo. Genera así unos flujos de audio y video sin comprimir, de características configurables:

- frecuencia de muestreo, tamaño de muestra y número de canales para el sonido
- tamaño de imagen, profundidad de bits, espacio de color (RGB o YUV), frecuencia de cuadro (framerate), ... para el video.

La aplicación de videoconferencia codifica esos flujos a MPEG4 para el video y a GSM para el audio, también con parámetros configurables (como bitrate, proporción de *frames* B, etc para el video); y por último los packetiza y transmite sobre RTP al interlocutor, que hará el proceso contrario: depacketización, decodificación y presentación al usuario.

Para que el usuario sepa lo que está transmitiendo se usa una ventana de monitorización de la captura, provisto por la propia API de captura del JMF. Ésta no es una solución óptima, ya que lo que se presenta al usuario es la captura sin las pérdidas debidas a la compresión, con lo que le puede dar una idea equivocada de la calidad de video que está transmitiendo; y, más importante, no es un simple reproductor de video, sino que controla algunos aspectos de la captura que normalmente no interesa que sean accesibles por el usuario.

La implementación ideal de la monitorización por el usuario sería mediante un player normal. Pero un flujo sólo puede tener un consumidor; si es consumido por un player, ya no podemos transmitirlo por RTP.

Figura 7 - Configuración sin clonación: una sola posibilidad

Para solucionar esto existe otra funcionalidad aportada por el JMF: el clonado de flujos. Durante el desarrollo de la aplicación de videoconferencia no se usó esta aproximación porque esta funcionalidad daba serios problemas, y la posibilidad de la ventana de monitorización era una solución aceptable. En algunos casos en que se necesitaba llevar un flujo a varios destinos (*processors* o *players*), se recurrió a recibir una copia del propio flujo RTP a la vez que se transmitía a una dirección de broadcast en la red local.

Pero para implementar la grabación de un flujo a la vez que se transmite por RTP en un entorno en que no se va a usar transmisión broadcast de los flujos, era imprescindible hacer funcionar el clonado.

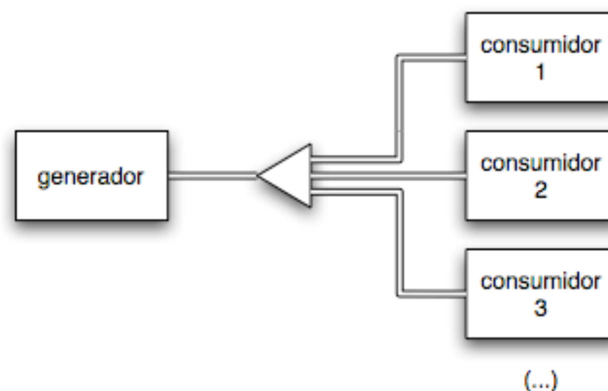

Figura 8 - Configuración genérica con clonación

Durante el desarrollo de la aplicación de videoconferencia ya se hicieron pruebas en el COMM para hacer funcionar la clonación de flujos, pero aparecían problemas graves de desincronías y calidad. En el desarrollo del proyecto actual esos problemas reaparecieron y se agravaron: desincronías variables e intermitentes, *frames* corruptos, e incluso cuelgues completos de la Máquina Virtual. Pero conseguimos ir delimitándolos y corrigiéndolos. Se trataba principalmente de problemas de concurrencia: el uso de varios clones de un flujo implicaba el uso (internamente al JMF) de varias threads que se encargaban de replicar la información en diferentes colas. Pero se daba un problema de productor / consumidor clásico en programación multihilo (incluso comentado en el tutorial estándar de Java), y deficientemente resuelto en el JMF, que provocaba la corrupción de la información de los flujos. Esto

provocaba la corrupción del video y el mal funcionamiento de la librería nativa en casos extremos, lo que llevaba al cuelgue completo de la Máquina Virtual. Se comentarán los detalles en el apartado correspondiente de esta memoria.

Implementación de la primera configuración

Una vez que conseguimos hacer funcionar el clonado de flujos, desarrollamos el primer prototipo, implementando la primera configuración comentada anteriormente: una aplicación servidor esperaba la conexión de una aplicación cliente para comenzar una videoconferencia. Cuando el servidor lo decidiese, empezaría a grabar tanto los flujos de audio y video generados localmente como de los recibidos por RTP, quedando los ficheros almacenados en el servidor.

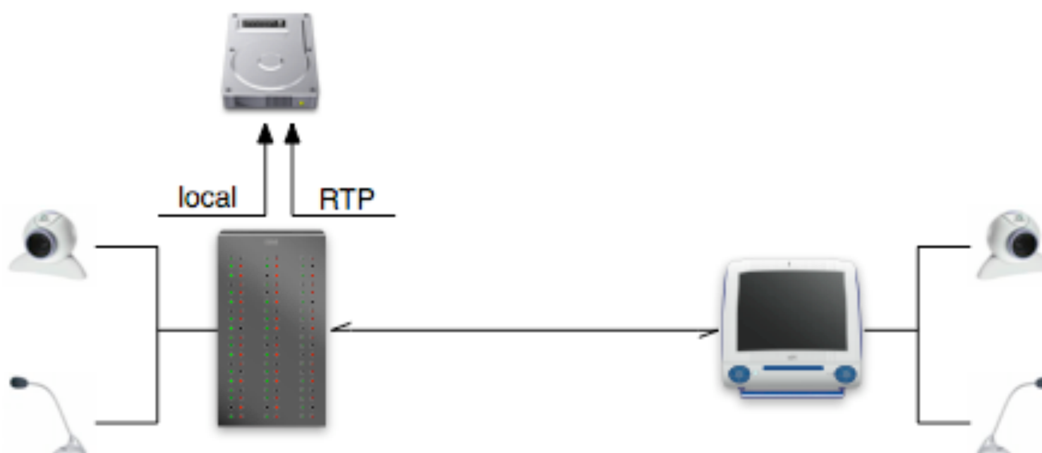

Figura 9 - Primera configuración considerada para el sistema

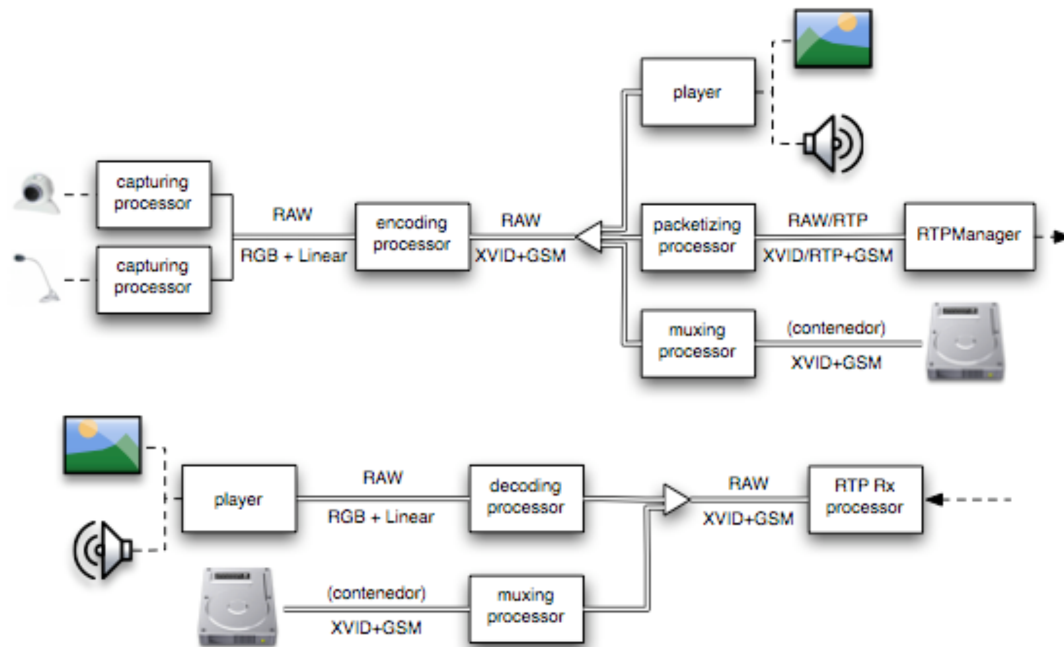

Figura 10 - Esquema de funcionamiento del servidor en la 1ª configuración

Mostramos el nuevo esquema del servidor, que es el único que sufre cambios. El cliente sigue siendo un simple cliente de videoconferencia, que ni siquiera tiene por qué saber que hay una grabación en curso.

En la aplicación de videoconferencia la conversión de video y audio sin comprimir a los formatos comprimidos y packetizados se hacía en un sólo paso. Pero para insertar la funcionalidad de grabación, es más conveniente separarlo en dos pasos: un primer paso de codificación y otro de packetización. Así podemos clonar el flujo resultante del primer paso y disponer de varios flujos ya codificados. Es la solución idónea para conseguir la nueva funcionalidad sin que suponga más carga para el servidor, ya que el clonado es un proceso que apenas carga la máquina (carga despreciable en comparación a la introducida por la compresión y descompresión del audio y sobre todo video).

Una mejora secundaria pero importante fue la que permitió dejar de usar el monitor de captura del JMF y substituirlo por un simple player. Como se puede ver, desde el momento en que tenemos el mecanismo de clonado funcionando, podemos conseguir un nuevo flujo que dedicar a un player que reproducirá lo que se está capturando. Este

player sí que representa una carga extra para el servidor, ya que está descomprimiendo el audio y el video que acababan de ser comprimidos. Como ventaja se puede aducir que permite al usuario ver con qué calidad real está transmitiendo, mientras que el monitor de captura simplemente mostraba la captura sin pérdidas. Además, tenemos más control sobre el player que sobre el monitor de captura; podemos actuar sobre él sin afectar a la captura y su procesado. Por ejemplo, podemos detenerlo o cambiarle el volumen para implementar un *comfort echo*. Hacer eso mismo con el monitor de captura era imposible, ya que respectivamente pararía o cambiaría el volumen del flujo resultante de la captura.

Lamentablemente, tras un periodo de pruebas y de análisis de los resultados, llegamos a la conclusión de que no era una configuración adecuada. La videoconferencia transcurría correctamente, pero las grabaciones presentaban desincronías. La razón es que cuando el JMF recibe flujos de audio y video por RTP, puede suceder que se pierda algún paquete. Si se recibe un paquete de audio con marca temporal adelantada respecto a la esperada, se asume que el paquete con la marca esperada se ha perdido, y se adelanta la base de tiempos hasta la nueva marca recibida. Durante una recepción RTP este comportamiento es correcto y normalmente da buenos resultados, ya que puede pasar que el paquete esperado se haya perdido o retrasado y la simple existencia del paquete más reciente significa que estamos quedándonos retrasados; el salto temporal nos pone realmente en el "momento adecuado" de la reproducción. Puede notarse alguna irregularidad en la reproducción pero normalmente será poco apreciable y de todas formas esperable en una videoconferencia.

Pero la grabación resultante de la recepción RTP también tendrá esos "huecos" en el flujo de paquetes, y un reproductor encontrará que siempre están disponibles los siguientes paquetes. El resultado es que, cuando falte un determinado paquete, en vez de la posible espera implícita en RTP, el reproductor recibe el siguiente paquete disponible, con lo que la base de tiempos salta a la nueva marca temporal. No sólo puede haber algún defecto momentáneo en ese instante (chasquido de audio, imagen corrupta), sino que la base temporal ha sido afectada para el resto de la reproducción: el tiempo se ha adelantado, y la grabación parecerá durar menos. Esto puede ser poco importante en una grabación aislada; pero cuando tenemos varias grabaciones que deben reproducirse simultáneamente, no es aceptable.

Para explicar mejor el problema, lo esbozamos con tres gráficas.

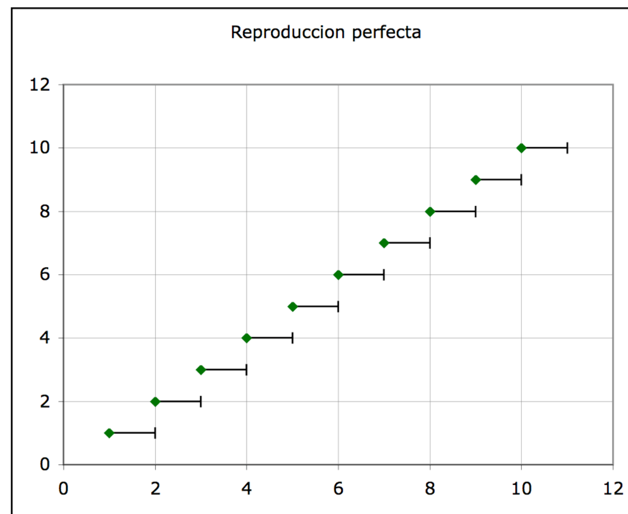

Figura 11 - Reproducción perfecta. No faltan frames.

La primera representa una reproducción perfecta: hay 10 *frames*, cada uno de ellos de duración un segundo. La duración total es de 10 segundos.

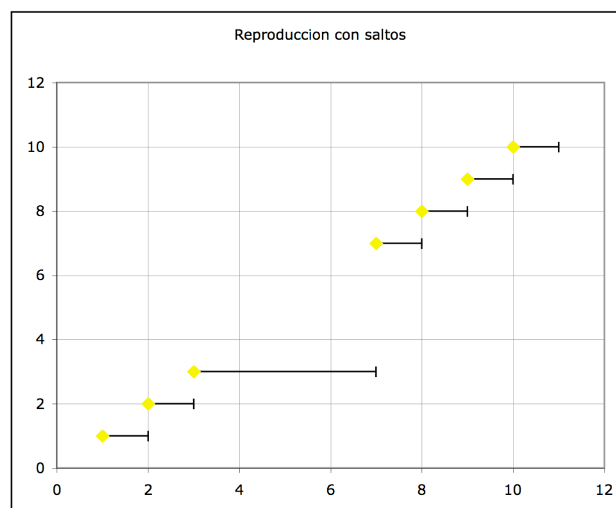

Figura 12 - Reproducción correcta ante la falta de frames

La segunda gráfica representa lo que sucede cuando perdemos los *frames* 4, 5 y 6 durante una transmisión RTP: sólo tenemos 6 *frames*, de duración 1 segundo cada

uno; pero el *frame* 3 es mantenido hasta que tenemos nueva información, o sea, 4 segundos. Duración total: 10 segundos.

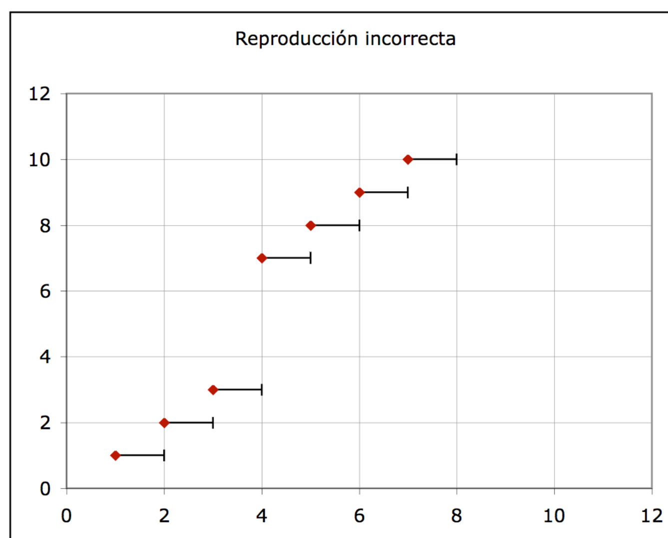

Figura 13 - Reproducción incorrecta ante la falta de frames

La tercera gráfica muestra el comportamiento del JMF durante una reproducción, que resulta ser incorrecto: ante la falta de los *frames* 4, 5 y 6, la reproducción salta hasta el siguiente *frame* que encuentra: el *frame* 7. Tenemos 6 *frames* de un segundo, y la duración total por tanto es de 6 segundos. Esto acarrea problemas.

Nótese que éste comportamiento podría explicarse con que el JMF esté siguiendo la misma estrategia cuando reproduce un fichero que cuando recibe flujos por RTP. Obviamente, el resultado no es aceptable para el propósito de este Proyecto Fin de Carrera.

Segunda configuración

Dado que este problema aparece al grabar los flujos que llegan por RTP, pasamos a implementar la segunda configuración propuesta, que consistía en la grabación de los flujos generados localmente a cada usuario.

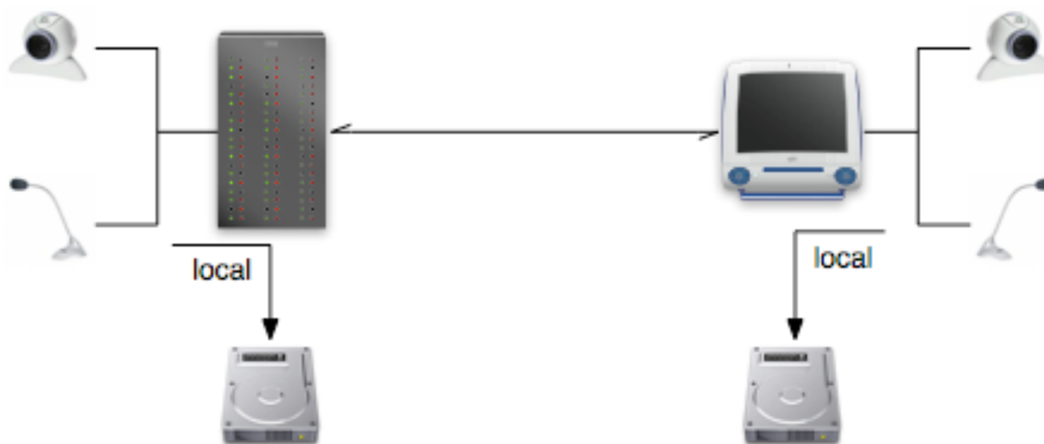

Figura 14 - Segunda configuración considerada para el sistema

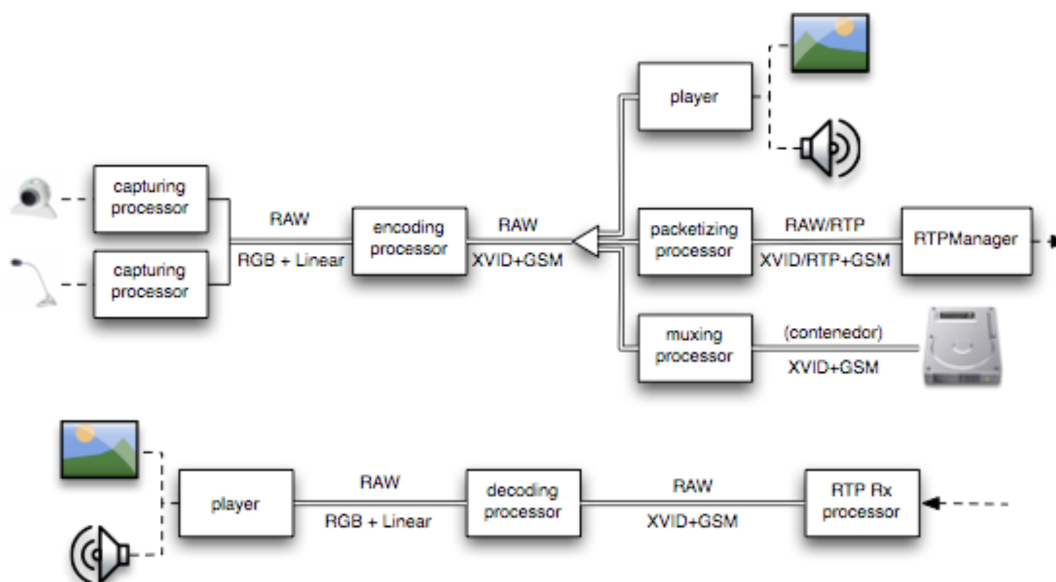

Figura 15 - Funcionamiento de cliente y servidor en la segunda configuración del sistema

Como se puede observar, la implementación consiste en el clonado de flujos generados localmente en cada extremo de la videoconferencia, para su grabación. La parte del programa referente a la recepción y presentación del flujo RTP se mantiene tal como era en el programa de videoconferencia básico. Hay que modificar tanto cliente como servidor; y, dado que las grabaciones quedarán almacenadas en el disco de cada uno de los participantes, habrá que añadir un paso de unificación o publicación de las grabaciones para que los participantes puedan tener acceso a ellas.

Lógicamente, la diferencia entre cliente y servidor no radica en el comportamiento respecto a la videoconferencia, ya que tanto uno como el otro hacen lo mismo: capturar y grabar, mientras transmiten y reciben la videoconferencia. La diferencia está tan sólo en la lógica de control del comienzo y parada de la grabación: se puede hablar más propiamente de maestro y esclavo.

En esta configuración, al no haber pérdidas de paquetes, el transcurso del tiempo en la grabación será correcto. Además tenemos la ventaja de que la calidad de la grabación será mejor, e incluso podría configurarse el sistema para que lo grabado no sea lo mismo que se transmitirá, sino que se podrían clonar los flujos de audio y video en otros puntos del sistema para conseguir diferentes funcionalidades:

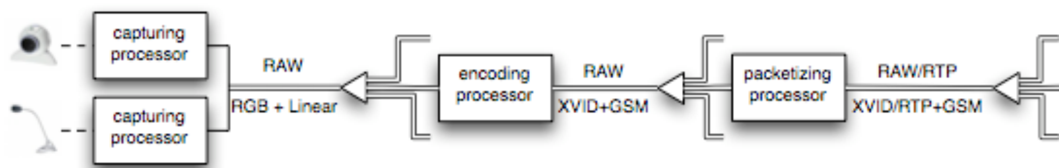

Figura 16 - Posibles puntos de clonación tras la captura

Por ejemplo, clonar justo tras la captura, con los flujos aún sin codificación, nos permitiría disponer de los flujos capturados sin alteraciones. Podría ser interesante para grabación a máxima calidad, con unos parámetros impracticables para la codificación destinada a la transmisión RTP. O al contrario, podría grabarse con una calidad mucho menor a la que se está transmitiendo, para sólo guardar un testigo de lo que fue la videoconferencia. También podría usarse para dar la funcionalidad de monitorización al usuario, sin incurrir en la carga que impone esta misma funcionalidad si se implementa a partir de un flujo que hay que decodificar.

Clonar tras la codificación nos permite disponer de varios flujos codificados con la carga de un sólo codificador. Como ya se ha visto, nuestra implementación usa uno de estos flujos para la funcionalidad de monitorización, otro para grabación y otro para la transmisión RTP necesaria para la videoconferencia. Pero podrían encontrarse más utilidades: por ejemplo, una transmisión paralela de otro tipo, por ejemplo por TCP, que podría ser interesante para enviar una copia íntegra de la videoconferencia a otro

destinatario. Esta funcionalidad se consideró como alternativa a la grabación local en el disco del cliente de la videoconferencia, pero el escenario más común sería que la transmisión TCP competiría con la propia transmisión RTP de la videoconferencia por el ancho de banda de la conexión; razón por la cual se desestimó esta posibilidad.

Aún hay una posibilidad más de clonado en otro punto: tras la packetización de los flujos. Pero obviamente, cuanto más nos alejamos de la captura (la fuente de señal), más especializados y restringidos son los usos que podemos dar a los flujos clonados, y en este caso sólo tendría sentido clonar tras la packetización si se quiere enviar los flujos RTP a diferentes direcciones.

Por supuesto, hay más posibles usos para el clonado; sólo se han mencionado los que podrían tener alguna utilidad para el tema que nos ocupa. Como ejemplo, uno de los prototipos que se desarrollaron en este proyecto para comprobar la fiabilidad del clonado y la carga de procesador que introducía fue un *videowall*, que clonaba N veces un flujo y presentaba simultáneamente todas las copias. La única limitación (como se detallará posteriormente) es que tras el clonado el primer flujo generado siempre debe mantenerse "ocupado"; ya que ése será el encargado de seguir pidiendo datos al flujo original.

Por completitud hay que decir que el clonado también se puede aplicar en cada uno de los pasos tras la recepción de los flujos RTP; pero como siempre sucede en telecomunicaciones, la señal es más útil cuanto menos ruido sufra, y la transmisión RTP se puede considerar la peor fuente de ruido (en forma de pérdida de paquetes); por lo que tras la recepción RTP tenemos una señal con pérdidas y bastante especializada (codificada a medida para la transmisión), lo que la hace poco interesante mas que para su presentación.

Grabación simultánea

Una vez que comprobamos que la implementación de esta segunda configuración era viable, pasamos a la siguiente funcionalidad necesaria: sincronización de la grabación en ambos extremos. Recordemos el escenario: hay una videoconferencia en marcha, y

en un momento determinado se desea comenzar la grabación. Para que luego las dos grabaciones sean reproducibles sincronizadamente, ambas deberán haber sido grabadas sincronizadamente también, independientemente del retraso o incluso jitter que pueda haber en la videoconferencia. Nos estamos refiriendo especialmente al momento de inicio de la grabación.

Se pueden imaginar dos casos de uso principales para el control de la grabación:

- Uno de los extremos de la videoconferencia decide cuándo comienza la grabación. El otro usuario es notificado pero no puede decidir cuándo empieza o acaba.
- Los dos extremos deciden. Cada usuario puede dar luz verde a la grabación (con un botón por ejemplo), pero sólo se efectuará cuando ambos lo hayan pulsado. La grabación acabará cuando cualquiera de los dos vuelva a pulsar el botón.

Otras posibilidades, como que la grabación pueda empezar cuando cualquiera de los usuarios decida, son simplificaciones de estos dos casos principales.

Para nuestra implementación nos decidimos por el primer caso de uso, que concuerda con el caso de cliente-servidor del programa de videoconferencia existente.

De cualquier forma, una vez se ha tomado la decisión de grabar, los extremos de la videoconferencia deben empezar sus grabaciones a la vez. Obviamente la información sobre el momento de comienzo será intercambiada por la red, pero habrá un retraso entre la emisión y la recepción de la orden, que habría que medir y compensar. Además como parte normal de la videoconferencia se está intercambiando información de temporización en los paquetes RTP y RTSP. Así que barajamos la posibilidad de aprovechar esta información para hacer algún tipo de compensación o sincronización; pero finalmente nos decidimos por usar una solución más simple y robusta: aprovechar que el reloj del sistema de cualquier OS actual es sincronizable por NTP.

Mediante el protocolo NTP, Windows (por ejemplo) sincroniza periódicamente el reloj del sistema con un servidor de tiempo. Así se puede contar con que el reloj del sistema nunca se alejará "demasiado" de un tiempo centralizado y confiable. Pero un problema con el uso de NTP por parte de Windows es que sólo se sincroniza con los servidores alrededor de una vez por semana, cuando NTP está pensado para ser usado con mucha más frecuencia (por ejemplo, una vez al día). En ese tiempo la deriva del reloj del sistema puede llegar a ser de decenas de segundos, mientras que para la grabación de videoconferencias el desfase entre grabaciones debería ser mucho menor que un segundo. Por tanto, decidimos que nuestro programa debería forzar una resincronización NTP al comenzar la videoconferencia. Esto lo hacemos con una llamada a la utilidad incluida en Windows para la administración del tiempo del sistema desde la línea de comandos (w32tm), pidiendo una resincronización del reloj con el servidor NTP.

El desfase entre los relojes de cliente y servidor será despreciable en el tiempo que dura una videoconferencia (lo estimamos mucho menor de un milisegundo por minuto), así que no tomamos ninguna medida especial para evitarlo.

Dado entonces que tenemos sincronizados los relojes del sistema entre cliente y servidor (ya que ambos acaban de sincronizarse con un servidor NTP), sólo queda por fin hacer que las grabaciones empiecen en el mismo momento, una vez el servidor haya dado la orden. No podemos simplemente mandar la orden por la red, ya que habrá un retraso potencialmente importante y, peor aún, variable. Así que seguimos el siguiente procedimiento para asegurar el comienzo sincronizado de la grabación:

0. Se supone que los extremos de la videoconferencia han sincronizado sus relojes con un servidor NTP, y la videoconferencia está en marcha.
1. El usuario en el servidor pulsa el botón de grabar. El servidor elige un momento en el futuro inmediato (próximos segundos) para comenzar a grabar, y lo comunica al cliente.
2. El cliente confirma la recepción de la orden y avisa al usuario que la grabación comenzará en x segundos.
3. Al llegar el momento designado, cliente y servidor comienzan a grabar.

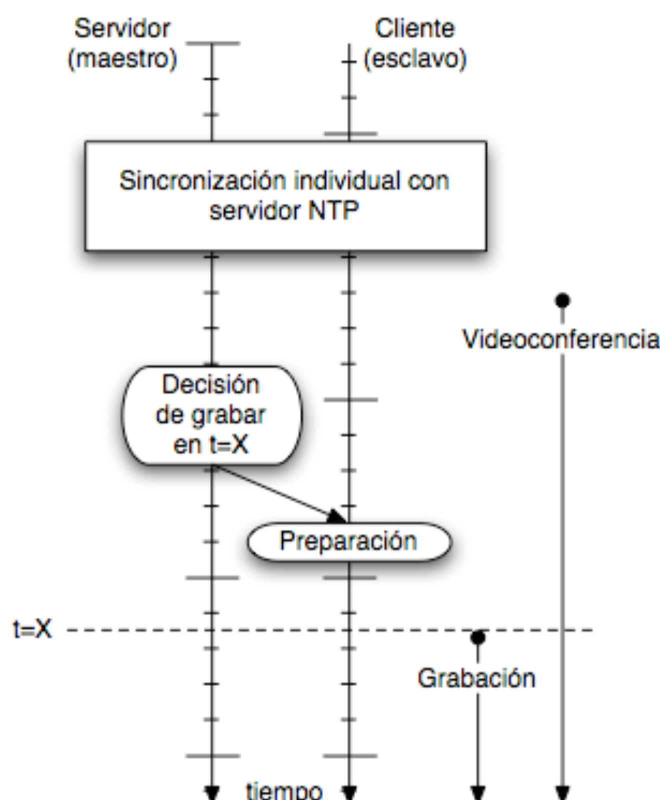

Figura 17 - Sincronización del inicio de la grabación

Esta forma de sincronizar los comienzos tiene la ventaja de que es totalmente independiente de los retrasos que pueda haber en la red (refiriéndonos, claro está, a los retrasos esperables en una red suficientemente funcional como para mantener una videoconferencia); asegura la sincronía; y no recarga el procesador en los momentos anteriores al comienzo de la grabación (lo cual, como veremos, es una ventaja importante en el trabajo con el JMF). La desventaja obvia es que hace falta soporte NTP en el sistema operativo, pero como ya hemos dicho viene incluido y activado por defecto en Windows 98 y posteriores, Linux y Mac OS X.

La implementación de este método de sincronización se tuvo que hacer con su propia serie de prototipos, de forma que conseguimos la sincronía de forma totalmente independiente del JMF. Después fuimos juntándolo con el prototipo de grabación hasta conseguir la grabación sincronizada. Mencionamos esto porque los programas hechos con JMF demuestran ser terriblemente delicados ante lo que se hace en el programa; de forma que simplemente abrir un *socket* al principio del programa puede hacer que las grabaciones que se hagan posteriormente tengan desincronías

importantes entre sus flujos de audio y video. Así que añadir una nueva funcionalidad a un programa que funciona correctamente suele ser no-trivial. Se detallarán este tipo de problemas y las soluciones encontradas en el apartado correspondiente de esta memoria.

Doble reproductor

Una vez conseguimos un prototipo funcionado con la videoconferencia, la sincronización y la grabación simultáneamente, sólo quedaba preparar un reproductor capaz de reproducir simultáneamente también los videos generados en cada extremo de la videoconferencia.

Se preparó una pequeña aplicación también usando el JMF para este fin, y con vistas a cubrir las características especiales necesarias para nuestro sistema:

— Los ficheros podrían estar situados tanto en el disco local como en un servidor remoto, quizás accesible por HTTP. Esto es necesario dada la grabación local en cada extremo. Para las pruebas en el laboratorio, por ejemplo, tras la grabación se accedía para la reproducción por un lado al fichero en el disco local y por el otro lado al fichero remoto, que había sido publicado en un servidor HTTP.

— El GUI estaba pensado para reproducir síncronamente los dos videos. No tiene sentido permitir que un video sea controlado independientemente; adelantar o retrasar un vídeo debe llevar al otro al mismo punto de la reproducción, para que ésta siga teniendo sentido. Por ello mismo no tiene sentido mostrar un GUI de control para cada video, sino que un GUI deberá bastar para controlar las dos grabaciones.

Nuevos problemas con la reproducción en el JMF nos hizo probar diferentes formas de implementar el reproductor, algunas de ellas recortando funcionalidad. De nuevo se verán estos problemas con más detalle en el apartado correspondiente de esta memoria.

Desincronías entre diferentes reproducciones

Finalmente lo conseguimos. Pero apareció un nuevo problema: las reproducciones se iban desincronizando poco a poco. Por ejemplo, podían empezar correctamente sincronizadas; pero al cabo de pocos minutos de reproducción una de las grabaciones había avanzado hasta estar 2 o 3 segundos por delante de la otra: las respuestas de un interlocutor se adelantaban a las preguntas del otro. Y al cabo de otros pocos minutos la situación podía invertirse.

Tras muchas más pruebas encontramos que el problema vuelve a ser de base en el JMF: la velocidad de reproducción de una grabación no es predecible, sino que pocas veces se mantiene constante (y aún más raramente concuerda con lo que debería ser). Por ejemplo, una grabación que debería reproducirse a 15 *frames* por segundo se reproduce a menos de 13 *frames* por segundo. Y lo hace así con varios matices que dan gravedad al problema:

— Ni siquiera es un número X constante de *frames* por segundo, sino que hay fluctuaciones continuas e importantes en la velocidad de reproducción. Se pueden apreciar que hay dos componentes: una primaria, que hace que la velocidad media sea de alrededor de 12.5 fps (por ejemplo; puede ser también 12 o 13, según la ocasión); y una secundaria, que aparece si se intenta compensar esa primera componente, y que puede hacer que un determinado *frame* aparezca más de 2 segundos adelantado o atrasado respecto a su momento correspondiente.

— El procesador está cargado a menos del 25% en todo momento, por lo que no se puede achacar a problemas de sobrecarga del equipo.

— Las grabaciones, cuando se graban en un formato estándar (MOV por ejemplo), parecen correctas (o con errores de poca importancia) cuando son reproducidas con otros reproductores (mplayer, Quicktime).

— No siempre sucede. Muy de vez en cuando las grabaciones se reproducen perfectamente (en nuestro caso, a 15 *frames* por segundo, siendo la distancia entre el momento de reproducción de un *frame* y el momento correcto de 100 ms o menos).

Todo ello parece indicar que el problema está en el JMF, pero que no será fácil de encontrar. Concretamente el hecho de que el procesador esté apenas cargado, que los retrasos sean variables y que a veces incluso funcione bien hace pensar que el problema puede ser profundo; por ejemplo, y para aventurar una idea, podría estar relacionado con el uso de las APIs de sonido del propio OS, ya que se sabe que algunos reproductores nativos (mplayer por ejemplo) pueden tener problemas de latencia y velocidad precisamente por eso.

Otra posibilidad sería el hecho de que la especificación de Java no provee al usuario con temporizadores con buena resolución. Por ejemplo, algunos métodos de Java como `Thread.sleep()` admiten un parámetro en milisegundos. Este tipo de método es usado para que un thread espere cierto tiempo sin hacer nada y sin cargar el procesador, lo cual es muy útil si, por ejemplo, acabamos de presentar un *frame* en pantalla y tenemos que esperar un número de milisegundos hasta la llegada del siguiente *frame*. Pero,

— No está garantizado que el control vuelva a ese thread después de ese número exacto de milisegundos; incluso si no hay más threads de usuario, el JVM siempre usa threads auxiliares, como las que mantienen el GUI o el Garbage Collector funcionando.

— Y peor aún, en la práctica la granularidad del tiempo medible en Java es de alrededor de 10 ms (dependiendo de la plataforma; por ejemplo, hemos visto valores de 1 ms para Linux y Mac OS X, 15 ms para Windows XP y se manejan valores de 50 ms en Windows 98) **[8]**.

Incluso para el método `System.nanotime()`, introducido en Java 1.5 y que devuelve el número de nanosegundos pasados desde un punto arbitrario en el tiempo, se

advierte en la propia documentación oficial que no hay garantías sobre la precisión ni la frecuencia con la que cambia el valor devuelto **[6]** . Compárese esto con el hecho de que cualquier OS actual permite acceder a temporizadores con precisión de nanosegundos, a veces menos; llegando por ejemplo a contadores de ciclos de CPU, que en cualquier procesador actual en que las velocidades de reloj son mayores de 1GHz daría precisiones mejores que 1 nanosegundo. No es de extrañar pues que haya un número de pequeñas librerías nativas disponibles para añadir este tipo de precisión a Java; aunque hay que recordar que estas librerías sólo nos brindarían precisión de medida de tiempo, y no precisión ni control de tiempos de ejecución. Para ello, la única respuesta desde Sun es que Java no ha sido diseñado para ejecución en tiempo real **[7]** .

Si esto fuese la causa de la reproducción multimedia imprecisa en Java, se puede decir que hemos tocado fondo respecto a las capacidades del lenguaje. Aún así, se podría entonces estudiar si es aceptable quedarse en velocidades de reproducción que no se acerquen a los límites de temporización de Java (10 *frames* por segundo quizás).

Reproductor doble basado en un reproductor estándar

El problema de la desincronía entre reproducciones viene de la reproducción con JMF. Quedaba la posibilidad de usar un reproductor estándar para reproducir las grabaciones.
Se planteaban entonces nuevas cuestiones:

— Habíamos elegido nuestro formato AVS para conseguir reproducciones individuales correctas, cuando aún no habíamos detectado el problema general de la reproducción fluctuante en JMF. El trabajo dedicado hasta ahora a estudiar y mejorar el multiplexor y demultiplexor de AVS quedaba ahora en saco roto y nos dejaba en un callejón sin salida; habría que volver atrás y volver a intentar usar MOV. Como se ha explicado ya, AVI quedaba prácticamente descartado a priori.

— MOV presentaba desincronías al reproducirse en JMF; no estudiamos cómo se comportaban las grabaciones hechas con JMF al ser reproducidas en en

player estándar, como el player Quicktime oficial de Apple o el open source mplayer. Habría que comprobar si en esos players el comportamiento era mejor.

— Por supuesto, cómo implementar el player doble a partir de uno de estos players.

Por suerte, parte del trabajo desarrollado en los multiplexores aún era válido: concretamente, el dedicado a asegurar que los frames capturados al principio de la grabación son los esperados, ya que se limitó a cambios en una clase abstracta de la que los demás multiplexores heredaban.

Las pruebas de reproducción de grabaciones hechas con JMF mostraron comportamiento uniforme tanto en Quicktime de Apple como en mplayer: desincronías, pero de poca magnitud, y muy predecibles: del orden de 1 segundo de desincronía cada 20 minutos de grabación a 10 fps. Además, la desincronía aumenta con el framerate de la grabación, lo que hace pensar que (como ya se dijo en el estudio inicial del formato en JMF) los timestamps se están grabando sistemáticamente incorrectos. Eso servirá para ayudar a encontrar el fallo concreto en el multiplexor del JMF, pero más importante ahora es que este resultado nos da una idea de lo que podemos conseguir con la grabación con JMF y reproducción con otro player: si aceptamos una desincronía de 250 ms entre audio y video como aceptable, se pueden grabar 5 minutos a 10 fps con ese resultado.

Estas pruebas además muestran que dos grabaciones reproducidas simultáneamente mantienen una sorprendentemente buena sincronía del video, mientras que el audio varía ligeramente. Además en una grabación el flujo de audio siempre es más corto que el flujo de video, siendo esta la fuente de la desincronía, y siendo la desincronía proporcional al tiempo transcurrido en la grabación. Esto nuevamente nos da otra idea de lo que puede estar fallando en el multiplexor del JMF: se pierden esporádicamente paquetes de audio, pero no de video.

Además, la cabecera de los ficheros MOV generados son incorrectas siempre. Podría ser un fallo poco importante, ya que los reproductores sólo parecen usar esa

información para dar detalles textuales sobre el formato del fichero; pero al fin y al cabo ese fallo indica que hay que revisar la generación de la cabecera, que sabemos que también incluye información crítica (como la duración de los paquetes de audio).

Respecto a la implementación, hemos experimentado tanto con mplayer como con Quicktime.

— Quicktime permite crear muy fácilmente ficheros personalizados (vs. players personalizados). Se pueden reunir dos ficheros MOV mediante un fichero extra que incluirá optativamente el interfaz que hayamos diseñado nosotros (ya sea un simple logo añadido a la ventana de reproducción normal o un interfaz completo, con nuestros propios botones); de forma que en cualquier caso podemos tener los dos ficheros reproduciéndose lado con lado, simultáneamente y sincronizadamente, con un único GUI para controlarlos a los dos. La creación del fichero extra se puede hacer a mano o, idealmente, scriptar, mediante AppleScript en Mac OS X o mediante un componente ActiveX / COMX en Windows. De todas formas, es muy posible que se pueda generar ese fichero de forma genérica para que cualquier par de grabaciones sea usable de esa forma; y hay que recordar que, ya que seguramente habrá que modificar en un futuro el multiplexor de MOV del JMF, seguramente podremos generar directamente cabeceras de fichero que nos ayuden.

— Mplayer tiene un modo de funcionamiento esclavo, en que acepta una serie de comandos para afectar la reproducción (pausa, continuar, volumen, *seeking*, etc). La entrada de comandos es en principio por `stdin`. Una forma con la que hemos experimentado para implementar un reproductor doble a partir de esto es usar la utilidad `nc` (*netcat*) del mundo Unix (también para Windows) para redirigir lo que se reciba en un puerto TCP o UDP a su salida `stdout`, y de ahí llevarlo con una *pipe* o tubería a mplayer. Como frontend a los players esclavos que generemos así, bastará cualquier aplicación que pueda mandar los comandos adecuados a los n puertos en los que esperan las utilidades `nc`. Por ejemplo, un pequeño programa Java podría conectarse y mandar el mismo comando a los dos puertos ante cada

interacción con el usuario; o incluso se puede usar la misma utilidad `nc` para enviar los comandos desde un simple fichero `.bat` en Windows o un shell script en Linux o Mac OS X. La presentación de un player doble construido así sería mucho más austera que en el caso comentado de Quicktime, pero aún así es perfecctamente funcional y usable por ejemplo en Linux, donde no se puede recurrir a Quicktime.

Implementado con Java WebStart

Volviendo al desarrollo del presente Proyecto Fin de Carrera, hay que mencionar que prácticamente todo el desarrollo se ha hecho manteniendo los prototipos ejecutables y desplegables por Java Web Start, e incluso hemos modificado partes del JMF para hacerlo plenamente usable con esta tecnología; ya que JMF apareció anteriormente a Java WebStart.

Estos desarrollos permiten usar las aplicaciones y prototipos sin requisitos de instalación previa en cualquier ordenador con un navegador web. Ésta es la implementación más adecuada para el tipo de uso previsto para la aplicación de videoconferencia y para el reproductor, ya que permiten que dos equipos cualquiera sean cliente y servidor. Si además se hacen accesibles las grabaciones tras la videoconferencia, el reproductor (también usable por Java WebStart) permitirá la reproducción en cualquier equipo, de nuevo sin instalaciones engorrosas.

Aplicación a nueva funcionalidad: grabación de la pantalla

El trabajo desarrollado hasta el momento es aprovechable para un rango de aplicaciones. Concretamente, el haber hecho funcionar el clonado y la grabación nos permite añadir esas funcionalidades a cualquier aplicación que use el JMF. Y los problemas encontrados en la grabación son relativamente poco importantes cuando tratamos con una sola grabación.

Así que se planteó la posibilidad de añadir al sistema de videoconferencia la funcionalidad de grabación de la captura y transmisión de la información en pantalla; así se tendría la posibilidad de guardar videos tipo tutorial sobre el uso de aplicaciones, clases magistrales, etc.

Hubo que comenzar analizando una vez más la forma en que se usaba el JMF en el programa para este propósito, y preparando los sources para poder introducir el nuevo código. De nuevo hay que hacer notar que el funcionamiento del JMF es extremadamente delicado y hay que ser muy cuidadoso al añadir o cambiar funcionalidad a un programa que ya funciona bien. Con lo que de nuevo empezamos con un prototipo que reproducía el uso del JMF y clases auxiliares en éste segundo programa.

La captura de pantalla funciona de forma diferente a la captura desde la cámara; para el caso de la cámara el JMF ya incluía lo necesario para usar las interfaces del SO (mediante librerías nativas JNI, específicas para Windows y desarrolladas por Sun) y así capturar desde cualquier dispositivo de captura de video usable por Windows. Pero no había soporte para capturar el contenido de la pantalla, así que hubo que implementarlo en el COMM.

La captura de pantalla se hace de dos formas diferentes, una de ellas *pure Java* que usa métodos provistos por la Máquina Virtual y otra mediante una nueva librería nativa que aprovecha código del proyecto open source VNC. Usar la librería nativa aporta velocidad, pero en cualquier caso lo importante es que se consigue una captura de pantalla que es inyectado en el JMF como un *frame* aportado por una nueva clase de dispositivo de captura de imagen.

Cuando se desarrolló este mecanismo de captura de pantalla e inyectado en el JMF, se hizo específicamente para ser incluido en el programa de videoconferencia, así que se procuró que funcionase correctamente la cadena captura → compresión XVID → packetización XVID/RTP.

Para la nueva funcionalidad de grabación de la pantalla, tal como se hizo para el programa básico de videoconferencia, lo ideal era clonar el flujo a la salida del

codificador Xvid. Pero, tras una nueva temporada de pruebas y prototipos, encontramos que el código existente no es capaz de soportar la funcionalidad. La razón es que, por la forma en que funciona el JMF, el programador puede pedir directamente una instancia de un processor con salida XVID/RTP, y el JMF se encargará de crear internamente la cadena de *plug-ins* que convertirá desde el formato de video de entrada hasta el de salida: en este caso, compresión XVID y luego packetización. Éso es exactamente lo que se estaba haciendo anteriormente.

Para añadir la nueva funcionalidad de grabación hace falta segmentar el procesado en dos estadios explícitos, uno primero de codificación XVID y otro de packetización XVID/RTP, de forma que se puedan extraer clones de los flujos cuando ya están codificados pero aún no han sido packetizados.

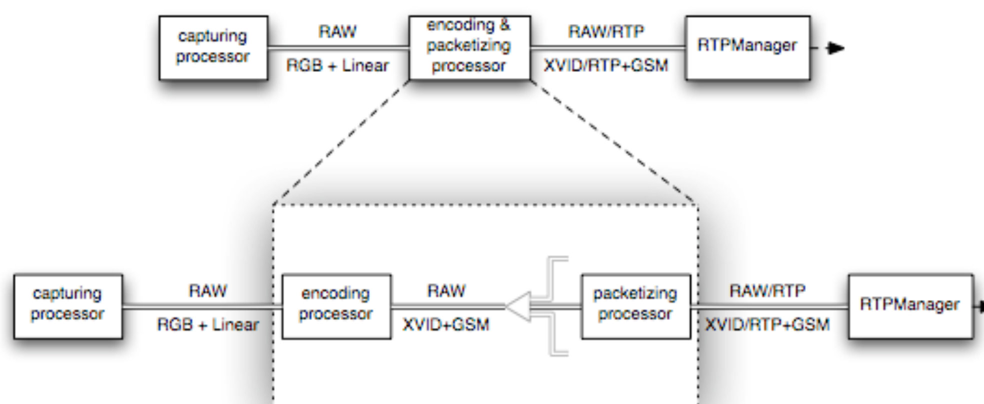

Figura 18 - Segmentación del procesado para hacer posible la clonación

Pero por un conjunto de problemas en la implementación del *wrapper* Java para el codificador XVID y la implementación del packetizador XVID/RTP (quizás incluso llegando hasta la implementación de la fuente de datos de captura de pantalla), dicha segmentación no es posible. Concretamente, el codificador XVID nativo acepta un conjunto de parámetros de salida que lo hacen bastante flexible y usable en una variedad de escenarios; pero, tal como está implementado su *wrapper* Java, sólo se acepta un subconjunto de configuraciones. Cuando no se usa directamente el codificador XVID sino que se crea un processor sólo especificando el uso del

packetizador, el JMF se encarga de buscar una configuración compatible entre la entrada del packetizador y la salida del codificador, y entonces todo funciona; pero usar sólo el codificador no es posible. Incluso forzando el uso de la configuración más genérica nos encontramos con que la librería nativa falla y provoca el cuelgue completo de la Máquina Virtual Java. Es un problema totalmente nuevo, ya que como se ha visto el uso directo del codificador XVID cuando capturábamos de la cámara de vídeo no planteaba ninguna dificultad parecida. Por supuesto, esto parece señalar indirectamente a algún problema en la fuente (indirectamente porque la fuente funciona, es el codificador XVID el que falla o no se deja instanciar): posiblemente usa una combinación de parámetros de salida del video no comprimido que provoca el mal funcionamiento del *wrapper* del codificador XVID.

Lo más importante es que, dado que la transmisión RTP de la captura de pantalla codificada en formato XVID funciona, el packetizador XVID/RTP tiene algo que le permite llamar correctamente al codificador XVID, aunque ni siquiera lo hace directamente (como se ha dicho, ése encadenamiento es tarea del JMF). Además, ese "algo" seguramente se debe a particularidades de la implementación de ambos; por ejemplo, el *wrapper* Java del codificador XVID no usa coherentemente sus interfaces, sino que hay que conocer los casos de uso que funcionan. Por tanto, sería recomendable revisar las implementaciones del *wrapper* del codificador y del packetizador, para eliminar esas particularidades y hacer que funcionen los casos generales, con lo que tanto codificador como packetizador serían usables con nuevas fuentes de datos o casos de uso sin más problema. Posiblemente habría que añadir también la comprobación de errores adecuada para prevenir los casos que se sabe que no funcionarían (por experiencia o por diseño de *wrapper*, codificador nativo o packetizador), para que de esa forma no aparezcan fallos catastróficos en otros puntos del programa, sino que se sepa en el momento de la configuración que los parámetros pedidos no son posibles; ya que JMF provee mecanismos para implementar todas esas posibilidades.

Capítulo 5

Detalles del desarrollo e implementación

Como se puede observar, el grueso del presente Proyecto Fin de Carrera ha acabado consistiendo en ir encontrando problemas en el JMF, tratar de definirlos con cierta precisión y finalmente encontrar una forma de rodearlos o de solucionarlos en su raíz.

Desde la publicación de la versión 2 de JMF en 1999, el API no ha evolucionado; la última novedad en el frente del JMF fue la publicación en el 2003 de la versión 2.1.1e de la implementación, que sólo aporta bugfixes y mejoras de compatibilidad. Sun apenas ha dado indicaciones de interés por mantener viva la tecnología; se ha hablado de reabsorción del JMF por parte del equipo de Swing, pero eso fue en el 2005 y no se ha vuelto a saber nada. Por otra parte, J2ME (la especificación de la plataforma Java para dispositivos móviles) tiene sus propias APIs para multimedia, que si bien no cuentan con todas las capacidades del JMF sí podrían representar competencia, o al menos eliminar la necesidad del JMF en ciertos escenarios.

Mientras tanto, han aparecido un número de proyectos open source para suplir las principales faltas del JMF (reducido número de codecs, principalmente, de los cuales el mundo open source tiene un buen número). También hay intentos de reimplementar todo el JMF (proyecto open source FMJ – *Freedom for Media in Java*). Incluso parte del código fuente del JMF ha sido publicado. Pero hay quien pide que Sun declare abiertamente en qué estado se encuentra el desarrollo del JMF, ya que todo parece apuntar a que simplemente está muerto, excepto el hecho de que Sun sigue teniendo online las descargas y documentación relativas al JMF, sin ninguna indicación de qué se espera en el futuro.

Hay que tener en cuenta que JMF es en realidad una especificación de interfaces y funcionalidades, y que el JMF descargable es simplemente la implementación conjunta por parte de (para la versión 2) Sun e IBM.

Parte de los problemas que hemos ido encontrando tienen sus raíces en el hecho de que JMF lleve tanto tiempo sin ser actualizado, como son algunos bugs o algunas funcionalidades que dan problemas en hardware moderno. En otros casos podría deberse a problemas de diseño o por la implementación conjunta de Sun e IBM.

Problemas generales con JMF

Problemas de instalación

Java 1.2 definió un mecanismo de extensiones: una forma de hacer accesibles a la JVM jerarquías de clases extras, para añadir capacidades nuevas al lenguaje. Lógicamente, esto siempre se ha podido hacer añadiendo los ficheros adecuados al Classpath de la JVM; pero las extensiones van un paso más allá, siendo accesibles a cualquier código como si fuesen parte del propio runtime de Java sin tener que importar sus nombres o modificar el Classpath: por eso se llaman extensiones.

Las extensiones suelen estar contenidas en packages cuyos nombres empiezan por javax, y su instalación se limita a poner su fichero .jar en el directorio correspondiente de la instalación Java (por ejemplo, java/bin/lib/ext).

Entonces, dado el status de extensión de JMF (y que sus clases principales pertenecen al package `javax.media.*`), es lógico suponer que JMF debería ser instalado como una extensión más. Su instalador no lo hace así, sino que copia los ficheros a un directorio y modifica el Classpath del sistema para que la JVM los tenga en cuenta; pero también puede ser comprensible, dado que JMF fue pensado para funcionar incluso en Java 1.1, donde aún no existía el concepto de extensión.

Entre nuestras pruebas, e intentando mantener un entorno de trabajo “limpio”, reinstalamos los ficheros de JMF en el directorio de extensiones, siguiendo el estándar. Sin embargo esto lleva a problemas: por la forma en que funcionan los ClassLoaders en Java, una extensión no es capaz de encontrar clases definidas en el Classpath (aunque sí al contrario, lógicamente). Y precisamente la arquitectura de *plug-ins* de JMF pretenderá hacer justamente eso, trabajar con clases definidas fuera de la instalación del JMF y por tanto fuera de las extensiones. Por lo que encontramos una razón práctica (y determinante) para no instalar JMF como una extensión normal.

Pero esto no quiere decir que lo mejor sea usar el instalador y olvidarse. El instalador de JMF para Windows instala las librerías nativas en el directorio Windows; esto provocará que Java WebStart no pueda proveer las librerías nativas que nosotros queramos, sino que ante un conflicto la Máquina Virtual sólo encontrará las librerías ya instaladas. Posiblemente se podría solucionar modificando la variable de entorno de Path, pero eso seguiría significando que Java WebStart no funcionaría en cualquier ordenador. Por tanto, irónicamente, hay que exigir que JMF no haya sido instalado en ordenadores que vayan a usar nuestras aplicaciones desde Java WebStart.
Por suerte, como veremos, es una exigencia razonable.

Hyperthreading

Los procesadores Pentium 4 de Intel soportan la tecnología Hyperthreading, que a grandes rasgos consiste en cambiar la forma en que se usan algunas partes del procesador para conseguir una mejora de rendimiento en cierta forma similar a lo que se conseguiría con dos procesadores. Según las pruebas de rendimiento, esta tecnología puede llegar a conseguir un 30% de mejora... o un ligero empeoramiento, según el tipo de aplicación y de código.

Cuando el procesador soporta esta tecnología, el Sistema Operativo ve el procesador como si se tratase en realidad de dos procesadores. Y en principio el tipo de aplicaciones que deberían ver una mejora en su rendimiento son las que hacen uso de multithreading, en las que varias threads aprovechan la tecnología de Hyperthreading para mantener en ejecución dos threads a la vez: una en cada “procesador lógico”.

La especificación de la Máquina Virtual Java desde siempre ha hecho uso de multithreading, tanto a nivel del lenguaje Java como a nivel de la implementación de la propia Máquina Virtual. Así que se podría pensar que Hyperthreading será una ventaja. Pero el hecho es que nuestras aplicaciones Java con JMF provocan cuelgues completos de la Máquina Virtual Java con una frecuencia alarmante cuando se usan en máquinas con Hyperthreading activado. No sabemos la causa concreta, ya que este tipo de fallos son extremadamente difíciles de analizar. De hecho, costó semanas de pruebas descartar otras sospechas (instalación de otros programas que podrían estar afectando de alguna forma, corrupción de ficheros o del Sistema Operativo, bugs en las versiones del Sistema Operativo o la Máquina Virtual, problemas del hardware de sonido, de video, USB o cualquier otro tipo...). Al final encontramos que desactivando Hyperthreading en la BIOS de la máquina eliminaba los cuelgues totalmente.

La relación de Hyperthreading con los cuelgues la podemos buscar a posteriori:

— en el hecho de que la Máquina Virtual de manera natural use threads;

— en que nuestra librería nativa no tiene gran protección frente a errores, con lo que podría estar ayudando a los cuelgues completos, y al ser nativa queda fuera de la gestión de errores procurada por la JVM;

— que los cuelgues son aleatorios e impredecibles, lo cual es típico de los problemas relacionados con programación multithread;

— que los cuelgues son más frecuentes a medida que aumenta el uso de threads en nuestro código, lo cual a veces no es evidente, ya que el JMF hace uso interno de threads. En todo caso, aumentar la cantidad de processors y clonado en los programas hace aumentar al menos linealmente el número de threads, y hace que los cuelgues pasen de ser un problema casi anecdótico para las primeras aplicaciones y prototipos (sin clonado, procesado "en serie" de los datos) a suceder más del 50% de las veces que se ejecutaban los prototipos finales (con clonado, paralelismo). Hay que hacer notar que esto

sólo lo descubrimos a medida que fuimos profundizando en el código fuente del JMF.

Fragilidad del código

Hemos mencionado la fragilidad de los programas que usan el JMF. Es un problema que permea todo el desarrollo seguido durante este Proyecto Fin de Carrera, pero que podemos ver concretado en un ejemplo claro de cómo ha afectado la funcionalidad de la aplicación de videoconferencia final.

Se ha explicado ya el proceso por el que se llega a sincronizar el punto de comienzo de la grabación entre los participantes en la videoconferencia. Para ello es necesario comenzar con una sincronización de cada participante con un servidor NTP. En el caso de los Sistemas Operativos Windows, hacemos una llamada a la aplicación de administración del sistema encargada de comenzar esa sincronización.

Pero esa sincronización tarda siempre algo más de 15 segundos, sin importar que el reloj del sistema ya esté sincronizado. Por ello, y aprovechando que una de las ventajas de Java es la facilidad que ofrece para la programación multithread (multihilo), diseñamos la aplicación de videoconferencia de forma que las fases de captura y transmisión de la videoconferencia pudiesen llevarse a cabo mientras un thread secundario se encargaba de la sincronización del reloj del sistema con el servidor NTP (Figura 19). El resultado fue que desde el arranque de la aplicación necesitábamos tan solo 4 segundos para estar ya transmitiendo o grabando lo capturado por nuestra webcam.

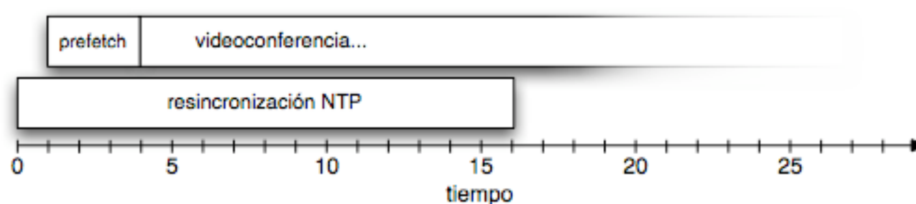

Figura 19 - Resincronización NTP en paralelo a la videoconferencia

Sin embargo, esta implementación provocaba que las grabaciones sufriesen desincronías aleatorias entre sus propios flujos de audio y video.

La única forma de solucionarlo fue volver a una versión más primitiva (Figura 20), en que se procedía secuencialmente a la resincronización NTP y posterior comienzo de la videoconferencia. Resultado: el usuario se ve obligado a esperar más de 18 segundos hasta que puede ver que la webcam está capturando.

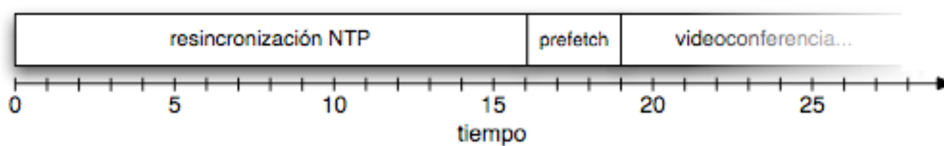

Figura 20 - Resincronización NTP previa a la videoconferencia

Problema de *frames* corruptos

Desde el principio del desarrollo de la aplicación de videoconferencia se habían encontrado algunos pequeños problemas con el uso normal de la captura de vídeo. Hablamos de problemas de poca importancia: eventual aparición de *frames* incompletos durante una videoconferencia, falta de *frames* I iniciales en los flujos de vídeo. Pero, como decimos, eran problemas de poca importancia: esporádicos, con pocas consecuencias, y fáciles de subsanar en el momento (por ejemplo, saludando a la cámara se consiguen variaciones en la imagen que fuerzan la aparición de *frames* I, por lo que surgió la costumbre de empezar las videoconferencias saludando a la cámara). Además este tipo de problemas se consideraron inevitables por la complejidad del sistema, ya que había una variedad de causas que podrían provocar ese tipo de problemas:

— podría tratarse de problemas esporádicos en la captura de vídeo desde la cámara USB, ya que a veces se encuentran errores similares en programas comerciales

— corrupción en la transmisión RTP (packetización o tránsito)

— problemas en el codificador nativo XVID (que durante buena parte del desarrollo de las aplicaciones era aún una versión beta)

— problemas en nuestro *wrapper* Java / JNI al codificador (aunque todo parecía funcionar correctamente aquí)

Lo que no se nos ocurrió es que podía deberse a problemas en el propio JMF (que, cómo no, acabó siendo el problema real). Esto lo descubrimos de nuevo al avanzar la implementación de prototipos que usaban clonado y varios reproductores; concretamente, el prototipo / aplicación de *videowall* antes comentada nos permitió ver que diferentes reproductores tenían diferentes problemas de calidad, pese a que todos deberían reproducir exactamente lo mismo, dado que todos compartían la misma fuente (captura en tiempo real o lectura desde un fichero).

Tuvimos que recurrir al código fuente del JMF para buscar el origen del problema. Los fuentes publicados por Sun no son la totalidad del JMF, pero por suerte nuestro problema si que estaba entre lo publicado; concretamente, en la parte del JMF encargada del clonado de flujos.

Funcionamiento interno del clonado

Como se vio en la explicación sobre el diseño del JMF, los *DataSources* son lo que conectan las diferentes etapas de procesado del JMF entre sí, a la manera de cables entre una fuente de señal y el consumidor de la señal.

Pero en realidad DataSource es sólo una clase abstracta en el API del JMF. Hay un número de clases que la extienden, según la funcionalidad específica que se deba prestar. Por ejemplo: hay *PushDataSources* y *PullDataSources*, según si la fuente de datos conectada al DataSource es de tipo *Push* (es decir, el servidor controla el flujo de datos, como un *broadcast* RTP o una capturadora de video) o de tipo *Pull* (es decir, el cliente controla el flujo de datos, como en el caso de una fuente accedida por HTTP o la lectura de un fichero). Así mismo hay otro tipo de *DataSources* especializado, que se usa para unir otros *DataSources* (*MergingDataSource*).

Para usar el clonado, se debe usar un DataSource que implemente el interfaz *SourceCloneable*. JMF se encarga de la conversión: con una llamada al método `Manager.createCloneableDataSource` se puede convertir cualquier DataSource en la clase equivalente que implemente el interfaz *SourceCloneable*.

Aquí se aísla al programador de bastante complejidad. La existencia de la clase Manager en el JMF y de sus métodos estáticos, como es el caso de `createCloneableDataSource`, nos permite no tener que implicarnos con la jerarquía de clases especializadas que heredan de la clase abstracta DataSource, y con su mapeo a las clases equivalentes clonables. No se trata de un mapeo inmediato, ya que implementar el interfaz *SourceCloneable* implica implementar nueva funcionalidad que necesita su propia "maquinaria especializada" (cada tipo de *DataSource* implementa de forma diferente el clonado, no sólo dependiendo de si se trata de un DataSource de tipo *push* o *pull* y basada en flujos o buffers, sino que además cada tipo de DataSource necesita una clase cloneable que será maestra de las clases clonadas). Y de hecho en la versión 2 del JMF, la jerarquía de *DataSources* normales ha sido implementada por Sun (con *packages* `com.sun.*`), mientras que la jerarquía de *DataSources* clonables ha sido implementada por IBM (*packages* `com.ibm.*`).

Este mismo mecanismo de usar un método estático en la clase *Manager* del JMF para crear *DataSources* especializados también se usa para convertir un conjunto cualquiera de *DataSources* en un *MergingDataSource*. De nuevo, se trata de aislar al programador de la complejidad implicada al reunir los *DataSources* y manejarlos simultáneamente, cumpliendo así con la encapsulación de funcionalidad esperable de un lenguaje orientado a objetos.

Volviendo al funcionamiento interno del clonado, una vez hemos hecho la conversión del DataSource original a un *CloneableDataSource* mediante los métodos estáticos de la clase utilitaria Manager del JMF, lo que tenemos es un nuevo DataSource (recordemos, clase abstracta antecesora de los *DataSources* especializados) que podemos clonar tantas veces como queramos, con una simple llamada al método

`createClone` del nuevo *DataSource* para crear cada nuevo clon. En este punto hay que tener en cuenta algunos puntos para asegurar el correcto funcionamiento del JMF:

— El DataSource original sigue existiendo, pero no debe ser usado para nada. El nuevo *CloneableDataSource* es el que ahora tomará su lugar a todos los efectos.

— Los clones que sean creados a partir del *CloneableDataSource* pueden tener diferentes características al *CloneableDataSource* o al DataSource original. Por ejemplo, si el DataSource original es un *PullDataSource* (es decir, el consumidor de los datos debe ir pidiéndolos), el *CloneableDataSource* también será un *PullDataSource*; pero, ¿qué sucede con los clones? Lógicamente, no pueden ser también *PullDataSources*, ya que cada uno podría pedir datos a diferentes ritmos y habría que buscar una estrategia adecuada para la situación: ¿se guardan los datos hasta que cada uno de los consumidores los haya pedido? ¿se produce un error? ¿se bloquea a los consumidores adelantados? Cada opción tendría sus problemas. Lo que hace el JMF es, en este caso, convertir los clones en *PushDataSources*: a medida que el *CloneableDataSource* pide datos, los clones los reciben como generados por la fuente: es decir, el *CloneableDataSource* actúa como maestro de sus clones, que son esclavos. No siempre tendría que darse ésta disparidad entre los *CloneableDataSource* y sus clones: si el DataSource original es un *PushDataSource* (es decir, el generador controla el flujo de datos), tanto *CloneableDataSource* como sus clones serán también *PushDataSources*.

— El tercer punto está implícito en el anterior, pero por su importancia hay que remarcarlo: el *CloneableDataSource* puede estar actuando como maestro de sus clones, marcando el flujo de datos, por lo que hay que mantenerlo en marcha. En casos en los que un clon pueda estar dedicado a usos no inmediatos a su creación (transmisión RTP o grabación a partir de la activación de un control del usuario, por ejemplo), no es recomendable que ese clon sea el *CloneableDataSource*, a no ser que interese precisamente detener cualquier otro flujo de datos hasta que empiece esa tarea. La documentación del JMF llega al extremo de proponer una clase *"dummy"*

que se encargaría de mantener vivo el flujo de datos simplemente pidiendo más, aunque sin usarlos para nada. Aunque puede parecer una solución un tanto extrema, hay que tener en cuenta que la alternativa más simple (crear un player que se encargue de mantener viva de la misma forma la transferencia de datos) tendría precisamente la desventaja de que el player sí que procesaría esos datos, con el consumo correspondiente de recursos (memoria, procesador, etc.). La solución por la que hemos optado nosotros es usar el *CloneableDataSource* para implementar un player que permite al usuario monitorizar cómo es la captura en su webcam, aunque aún no esté transmitiendo o grabando. De esta forma matamos dos pájaros de un tiro, convirtiendo el requerimiento del funcionamiento del JMF en una funcionalidad interesante para el usuario.

Todos estos detalles sobre la forma de uso de los *CloneableDataSources* y sus clones ya puede ir dando una idea de cómo funciona internamente su implementación: el productor de los datos (el DataSource original) los dejará accesibles para el *CloneableDataSource*, que será el consumidor; y los clones verán estos datos inyectados en su flujo, convirtiéndose así en *PushDataSources*.

Dada la corrupción de *frames* que estábamos observando, nos vimos forzados a analizar todos los datalles del funcionamiento interno del JMF implicados en nuestros prototipos. Llegamos al punto en que tuvimos que analizar qué estaba pasando en la implementación del clonado, y encontramos problemas.

Productor y consumidor

El funcionamiento del *DataSource* original y del *CloneableDataSource* constituye un problema clásico de productor y consumidor de datos. En este tipo de problema, una entidad produce datos y los deja en un "lugar" (un buffer) accesibles por otra entidad que los consume, siendo ambas independientes. Hay que tener en cuenta los siguientes casos:

— Puede suceder que no haya datos disponibles en un determinado momento (si el consumidor es más rápido que el productor, por ejemplo); el consumidor deberá entonces esperar la llegada de nuevos datos. Sólo el productor puede dejar datos en el buffer.

— Puede suceder que el productor no pueda proveer datos nuevos porque los datos anteriores aún no ha sido consumidos; el productor deberá entonces esperar al consumidor. Sólo el consumidor puede retirar datos del buffer.

— Hay que evitar que consumidor y productor tengan acceso simultáneo al buffer; o, dicho de otra forma, cada uno debe tener acceso exclusivo al buffer. Si esto no se tuviese en cuenta, podría suceder que el productor crea que no puede proveer más datos porque el consumidor no ha leído los ya existentes, cuando en realidad acaba de leerlos; que el consumidor crea que no tiene datos nuevos disponibles, cuando en realidad el productor acaba de generarlos; y, en el peor de los casos, que el productor altere los datos que el consumidor está leyendo al mismo tiempo, provocando la corrupción de los datos.

En el caso particular de *DataSource* original y *CloneableDataSource* como productor y consumidor respectivamente hay además otro detalle que lo complica: no sólo hay varios consumidores (y el número puede variar dinámicamente durante la ejecución del programa), sino que además los consumidores no son todos iguales. El *CloneableDataSource*, si es un *PullDataSource*, podrá elegir cuándo recupera el dato siguiente; pero sus clones, convertidos en *PushDataSources*, verán el dato inyectado en su flujo.

La solución a este problema clásico es también clásica, y de hecho el tutorial oficial de Java publicado por Sun habla tanto de este problema como de cómo solucionarlo. De las diferentes formas en que se puede solucionar éste problema, la más natural en Java es el uso de los llamados monitores: infraestructura e instrucciones de sincronización para definir qué zonas de código (variables, métodos) son de uso exclusivo.

Al analizar la implementación de la solución en el mecanismo de clonado de flujos, nos encontramos con que los bufferes estaban sufriendo algún tipo de corrupción. Se puede decir que el problema tiene dos causas:

- La solución clásica para la situación del productor – consumidor cuenta con un buffer de varias posiciones donde alojar datos. Sin embargo, la implementación del JMF usa un buffer de una sóla posición.
- El uso de la librería nativa XVID implica una cierta infraestructura para la comunicación entre Java y el entorno nativo mediante el Java Native Interface. Parte de esta infraestructura mapea porciones de memoria (arrays de bytes) manejadas por la librería nativa para sus labores de codificación y decodificación con variables y arrays accesibles en Java; y lo hace de forma que en Java lo que se está manejando son referencias a la sección de memoria reservada por la parte nativa. La alternativa sería hacer una copia del contenido de la memoria nativa a una variable de Java, y viceversa. Pero dado que estas zonas de memoria contienen datos de video comprimido y descomprimido, se trataría de copiar secciones relativamente grandes de memoria decenas de veces por segundo, con las complicaciones que eso conlleva: carga extra del procesador, uso redundante de memoria, posiblemente incluso "contaminación" del espacio de memoria de Java con lo que el recolector de basura aún cargaría más el procesador. Claramente esto no es deseable, por lo que se eligió trabajar sólo con las referencias a la memoria ocupada por la información de imagen.

Que el buffer usado para el clonado (productor – consumidor) tenga una sola posición, y que el dato recogido por los consumidores (*CloneableDataSource* y sus clones) sea una referencia, estaba provocando que el contenido de la zona de memoria nativa usada por la librería XVID fuese reusada demasiado pronto: una vez el *CloneableDataSource* había llevado sus datos hasta el siguiente paso del procesado y procedía a leer un nuevo dato (como consumidor), la memoria referenciada era reusada con nuevos datos, de forma que los clones que aún no hubiesen entregado sus datos a sus siguientes pasos de procesado se encontraban con datos inconsistentes.

La forma en que hemos solucionado este problema ha sido cambiando la implementación del mecanismo de clonado. Con nuestra modificación, tanto el *CloneableDataSource* como cada uno de sus clones cuenta con su propio buffer circular de N posiciones. Cada vez que el productor genera un nuevo paquete de datos, es añadido al buffer de cada uno de los consumidores, de forma que las diferencias en la velocidad de procesado de cada clon, o incluso que una de las threads que implementa un clon tarde más de lo esperable en ejecutarse, no afectarán a los datos leídos. Concretamente, nuestra solución da un margen de tiempo para que los clones puedan recoger los datos y llevarlos al siguiente paso de la cadena de proceso; ya que el *CloneDataSource* tiene que recorrer las N posiciones del buffer circular para empezar a reutilizar zonas de memoria antiguas. Los clones tienen el tiempo correspondiente al procesado de esas N posiciones para leer y transportar los datos: para el caso de video, que es el que más problemas estaba causando, eso significa N veces el tiempo de *frame*.

Hemos hecho pruebas con más de 20 clones (con la ya comentada aplicación de *videowall*), algunos de ellos mostrándose en pantalla (lo cual implica la decodificación en tiempo real) y otros siendo grabados a fichero, y hemos encontrado que con N = 5 es suficiente para eliminar los problemas citados de corrupción de *frames*.

Los problemas de pérdida de *frames* I también se debían principalmente a este tipo de corrupción. Dado que la identificación de *frames* I se hace por la aparición de un marcador en el paquete de datos que ve el decodificados de la librería nativa Xvid, si la zona concreta donde debía estar situado este marcador había sido sobrescrita por datos de otro *frame*, el *frame* no era reconocido. Es un caso particularmente problemático, ya que el decodificador se mantiene inactivo al comienzo de un flujo de datos hasta que encuentra un *frame* I, ya que sólo éste tipo de *frames* lleva la información necesaria para decodificar los *frames* P y B que vendrán después. En la práctica esto se traduce en la ausencia de imagen, dejando claro al usuario que algo no está funcionando como debe; de hecho da la sensación de que la cámara no está funcionando, pese a que el resto del JMF está ocupado llevando el flujo de datos hasta el decodificador, que se limita a ignorarlos, manteniéndose a la espera de un *frame* I.

Este problema era también importante una vez el procesado de video había comenzado, pero su repercusión era mucho menor: el usuario sólo notaría normalmente una pérdida de calidad en forma de una imagen distorsionada en medio del flujo de video.

Puede ser interesante comentar que tuvimos que introducir la modificación comentada en las clases del JMF implementadas por IBM (`com.ibm.media.protocol.CloneableSourceStreamAdapter`), ya que ahí es donde se encuentran todas las clases dedicadas a la funcionalidad de clonado; y que para nuestra implementación pudimos usar un buffer circular disponible precisamente en el propio JMF... pero entre las clases implementadas por Sun (`com.sun.media.CircularBuffer`). ¿Quizás si ambas partes hubiesen sido del mismo implementador la funcionalidad de clonado hubiese sido más robusta? Desde luego, la documentación del API no dice nada al respecto, como suele suceder con las clases que se salen de los packages estándar de Java (`java.*`, `javax.*`). Al fin y al cabo los *packages* `com.ibm.*` y `com.sun.*` son particulares de la presente implementación del API del JMF, y se puede decir que normalmente sería ya de agradecer que tengamos acceso al código fuente de estas partes del JMF; pero dada la situación de abandono y poca fiabilidad que demuestra, encontramos que el código fuente se hace necesario para que JMF apenas empiece a cumplir lo que prometía.

Formatos de grabación

Encontrar problemas con el JMF ha sido la tónica durante el desarrollo de este Proyecto Fin de Carrera. Y ante cada problema ha habido que intentar definir en qué consistía exactamente el problema, hasta qué punto era importante, y si había posibilidades de corregirlo o al menos evitarlo.

Pero claro, llegados al punto en que estamos modificando el propio JMF, todo el trabajo hecho anteriormente queda prácticamente desfasado. Los antiguos problemas quizás ya no existen, o han cambiado (siempre esperando que a mejor). Pero aunque así sea, los problemas que habían sido "puenteados" mediante una solución no-óptima siguen siendo no-óptimos, o incluso pueden haber dejado de funcionar si se basaban

en el comportamiento erróneo del JMF que hemos eliminado. Y todo esto sin tener en cuenta la tremenda fragilidad del código que recurre al JMF, como ya se ha comentado.

En el mejor de los casos, las soluciones que ya habíamos desarrollado seguirán funcionando y *simplemente* será código dificil de explicar y mantener para usos futuros – suponiendo que una futura modificación del JMF no acabe haciendo que dejen de funcionar.

En nuestro caso, por ejemplo, llegados al punto en que el clonado de *DataStreams* funciona correctamente, al hacer algunas pruebas rutinarias encontramos que algunas áreas del JMF que antes daban enormes problemas ahora funcionaban sorprendentemente bien. Por ejemplo, recordemos el caso de los formatos contenedores AVI y MOV, que ya durante el desarrollo de la aplicación original de videoconferencia demostraron ser imposibles de usar. AVI siempre producía ficheros con las imágenes terriblemente corruptas y el sonido totalmente desincronizado; mientras que los ficheros MOV tenían relativamente buena calidad pero desincronías variables entre el audio y el video. Por todo ello se desarrolló el formato AVS, como se ha comentado ya, para solucionar estos problemas de calidad y desincronía.

Y sin embargo, tras las últimas modificaciones al JMF, encontramos que los formatos contenedores AVI y MOV funcionaban casi perfectamente.

Más adelante en el proyecto podría tener importancia el tipo de formato contenedor que usásemos. Por ejemplo, usar nuestro formato AVS nos obligaba a desarrollar el reproductor para las dos grabaciones de la videoconferencia en Java mediante el JMF; mientras que usar MOV, por ejemplo, no sólo nos permitiría usar reproductores estándar sino que nos haría considerar otras formas de implementar el reproductor final, como por ejemplo en forma de *skins* o personalizaciones para un reproductor Quicktime normal. En general siempre nos pareció más interesante usar un formato estándar: así, desarrollar nuestro propio reproductor es sólo una opción, en vez de una necesidad. Sin olvidar que nuestro formato AVS no es perfecto, ya que para asegurar la correcta sincronización de audio y video no sigue algunas de las recomendaciones

del JMF, y eso provoca algunas particularidades en la reproducción, como ya se ha indicado.

Así que el hecho de que ahora los formatos contenedores estándar del JMF funcionasen correctamente hicieron que volviésemos a empezar las pruebas de grabación, para comprobar si realmente el comportamiento era bueno.

Y una vez más encontramos nuevos problemas. Esta vez, aunque la calidad de la imagen era buena y la sincronía entre audio y video también (con fallos ocasionales típicos del JMF), encontramos un (nuevo) fallo serio: los multiplexores (es decir, las clases del JMF que implementan la grabación de flujos en los diferentes formatos contenedores) están pensados para funcionar desde el principio de un flujo. Cuentan con tener acceso a información transmitida con el flujo de datos, que pueden ser necesaria para escribir en disco correctamente la metainformación de los ficheros de video (es decir, información sobre los datos que se están grabando: especificación FOURCC de los formatos de video y audio, tamaño de cuadro, detalles de la base de tiempo, etcétera). Esto es razonable, pero esta implementación recorta la funcionalidad del JMF de forma arbitraria: los multiplexores sólo trabajan correctamente si han estado en funcionamiento desde el principio del flujo de datos.

— Cuando los multiplexores son instanciados, capturan un número de *frames* del flujo incluso aunque no hayan recibido la orden de empezar a grabar.

— Una vez activada la grabación, el multiplexor no grabará nada hasta que reciba un *frame* I o hasta que venza un *timeout* arbitrario definido en el propio multiplexor.

Ambos puntos funcionan satisfactoriamente si, como decimos, el multiplexor intenta grabar el flujo desde su mismo comienzo. Y, creemos que sospechosamente, ésa es la única funcionalidad de grabación que se ofrece en las aplicaciones de ejemplo y la documentación de Sun: empezarla a la vez que la captura o recepción del flujo multimedia.

Sin embargo para nuestra aplicación el flujo multimedia lleva rato empezado cuando activamos la grabación. Y en ese caso, puede suceder que el multiplexor haya

capturado *frames* anteriores a la activación de la grabación. Esto es malo por dos motivos: uno, que esos *frames* tienen lógicamente *timestamps* correspondientes al momento de su captura (anteriores por tanto al principio real de la grabación). Y otro, que entre esos *frames* puede haber un *frame* I.

Para entender lo que esto implica, hay que explicar qué es lo que se espera de un fichero grabado y, frente a eso, qué es lo que sucede cuando empieza una grabación más tarde que la captura: el multiplexor recibe la orden de comenzar la grabación y se encuentra con un flujo de *frames* que está en un punto arbitrario de su evolución. Por otra parte, un fichero grabado debería:

- Especificar la metainformación necesaria para decodificar correctamente los flujos que contiene
- Comenzar sus *timestamps* en un t=0
- Comenzar sus flujos de forma que puedan ser decodificados (por ejemplo, con un *frame* I si se trata de un flujo MPEG4)

Los *frames* I no sólo nos dan un punto en el que comenzar la decodificación, sino que nos dan la mayoría de la metainformación necesaria para especificar las características del flujo.

Así que el comportamiento esperable de un multiplexor sería, una vez llegada la orden de comenzar la grabación, esperar la llegada de un *frame* I (quizás incluso pidiendo explícitamente un *frame* I al codificador); y una vez recibida, tomar su *timestamp* como punto t=0, y corregir los *timestamps* de los siguientes *frames* para hacerlas coherentes con este nuevo “eje de tiempos”.

Los multiplexores incluídos en JMF, por su parte, cumplen estos puntos pero parece que suponiendo que están recibiendo los flujos desde su comienzo. Lógicamente los flujos comienzan con un *frame* I, así que si suponemos que el flujo acaba de empezar es lógico almacenar cuanto antes los primeros *frames*; puede representar una ventaja, por si (por ejemplo) estar ocupados con la preparación del fichero en disco nos hiciese

perder ese primer *frame*. Así después ya tendríamos la metainformación necesaria y la grabación podría empezar sin esperas y, por tanto, sin más pérdidas de tiempo; en el peor de los casos habría algún pequeño problema de calidad el los primeros pocos *frames* de la grabación, pero incluso eso podría quedar enmascarado por el propio comienzo de la reproducción y el lento arranque típico de las cámaras. En fin, se puede decir que es una decisión de diseño: en el desarrollo del JMF decidieron que era mejor un posible pequeño fallo de calidad en un momento difícil de detectar que una espera de quizás segundos cada vez que comenzase la grabación.

Además, grabar desde el comienzo nos ahorra los problemas de tratar con los *timestamps*; tan sólo hay que volcar los *frames* con sus *timestamps* originales al disco. Teniendo en cuenta la fragilidad de los programas que usan JMF, intentamos en nuestros prototipos dejar preparada la estructura necesaria de captura, clonación, procesadores y grabación para que, una vez comenzada la captura y videoconferencia, no hubiese que hacer nada más que activar la grabación. Pero fue entonces cuando encontramos que los multiplexores estaban insertando los primeros *frames* de la captura, mucho antes del comienzo pretendido de la grabación. Esto provocaba grandes problemas de calidad (puesto que no había relación alguna entre el *frame* I capturada al principio y los *frames* que le seguían) y sobre todo de temporización, puesto que siendo el primer *frame* de los primeros milisegundos, los siguientes *frames* (con *timestamps* muy posteriores) apenas habían sido corregidas: la grabación resultante parecía quedar congelada hasta llegar al momento en que se había comenzado la grabación.

Nuevamente recurrimos al código fuente de los multiplexores para intentar corregir estos problemas, pero esta vez no fue posible; no es un problema limitado a los multiplexores, sino que implica la forma de funcionamiento de los propios *DataSources*.

Barajamos la posibilidad de simplemente retirar la funcionalidad de empezar la grabación posteriormente a la captura, pero nos pareció una solución poco elegante. Al fin y al cabo se trata de una aplicación de videoconferencia, en la que la grabación es una funcionalidad extra: por eso ofrecemos la grabación al pulsar un botón, no desde el principio. Además esto permite a los usuarios prepararse para la grabación.

Y, principalmente: nada en el diseño del API del JMF indica que un multiplexor deba recibir un flujo en su integridad. La clonación de *DataSources* representa una gran flexibilidad que queda absurdamente recortada con esta implementación.

Finalmente, conseguimos encontrar una variación en la estructura de los prototipos para minimizar el problema: se prepara la captura, reproducción y transmisión como siempre; y sólo cuando se quiere comenzar a grabar se clona el DataSource adecuado y se conecta a un Processor que prepara el flujo para el formato contenedor elegido, que comienza la grabación inmediatamente. Es precisamente lo contrario a lo que la intuición dicta tras hablar de la fragilidad del JMF, ya que se trata de un cambio muy importante en el funcionamiento del programa cuando el JMF ya está en pleno trabajo; pero en este caso es la única forma de que funcionen los multiplexores sin grandes cambios internos en el JMF.

Y aún así, después de todo, los multiplexores incluidos en el JMF resultan no ser válidos para nuestros propósitos:

- El formato contenedor AVI, como ya se dijo, no soporta variable *framerate*. Se supone que los *frames* de audio y video llegan con una frecuencia fija, así que no se guarda información de *timestamps*, sino que al acabar la grabación se guarda en el fichero el tiempo medio de *frame* de audio y video. Y dado que el video que llega de la captura de webcam no cumple esta suposición, sino que llega con pequeñas variaciones, nos encontramos con que grabaciones largas en este formato desincronizan audio y video, pese a que grabaciones cortas (muy pocos minutos) pueden dar la sensación de funcionar correctamente. No obstante, de vez en cuando además aparecen fallos mucho más graves (audio y video sin ningún tipo de sincronía; por ejemplo, primero se oye todo el audio y luego aparece todo el video), y estos fallos sólo son detectables al acabar la grabación. Se debe concluir pues que éste multiplexor simplemente no es suficiente robusto, aparte de la debilidad del propio formato.

- El multiplexor para formato MOV, como el de AVI, tiene el código fuente repleto de comentarios y avisos en que los autores muestran sus dudas y

problemas cuando implementaban el multiplexor. Parte del problema parece ser que la especificación del formato publicada por Apple detalla muchos tipos de funcionalidad, y sólo la más importante ha sido implementada en el de/multiplexor; pero aún así hay puntos muy dudosos incluso en lo más básico, como es por ejemplo la grabación de los *timestamps*. Sin ir más lejos nos encontramos con que los flujos grabados en un formato contenedor MOV, al ser leídos por el propio JMF, tienen alterados sus *timestamps*. Y de nuevo las grabaciones cortas parecen ser correctas, pero al hacer pruebas de mayor longitud (decenas de minutos) encontramos que aparecen desincronías progresivas.

Así que finalmente volvemos al formato contenedor AVS, ya que, aunque no funciona completamente y requiere cuidado y pequeñas modificaciones en otras partes del JMF, al menos sabemos que funciona relativamente bien en su cometido. Incluso hemos ido arreglando alguno de los defectos que presenta: por ejemplo, hemos implementado la capacidad de seeking y conseguido minimizar los problemas de pre-captura de primeros *frames* que ve el Processor grabador.

La arquitectura del JMF en la que se encuadran los multiplexores no provee un método para que el multiplexor pueda pedir *frames* I al codificador correspondiente; lo solucionamos haciendo que sea el mismo programa que instancia codificador y grabador el que pida *frames* I inmediatamente después de activar la grabación.

Problemas anteriores ya solucionados

Para dar un poco de contexto sobre los problemas en el JMF ya solucionados anteriormente en el COMM, podemos nombrar fallos tan importantes como la creación de *timestamps* incorrectos en los paquetes de flujos RTP y RTCP. Esos problemas ya hubo que solucionarlos en su momento modificando el código fuente implicado del JMF.

Desincronía entre reproducciones

Una vez habíamos conseguido tener todas las piezas el proyecto en sus sitios (aplicación de videoconferencia maestra y esclava, desplegadas con Java Web Start y reproductor múltiple basado en JMF), encontramos que los flujos grabados durante una videoconferencia y posteriormente reproducidos en el reproductor múltiple no mantenían una buena sincronización entre ellos.

Durante el proceso de corrección de los problemas vistos hasta ahora, se procedió a instrumentar las clases del JMF que íbamos modificando para poder recoger información extra del funcionamiento en tiempo de ejecución. La práctica habitual cuando un programa presenta problemas de ejecución es usar un debugger para encontrar dónde aparece el fallo; pero usar un debugger implica cambiar el entorno de ejecución del programa, a veces de formas sutiles. En todo caso, para aplicaciones sensibles en temas de temporización (como es el caso de la multimedia en general), un debugger resulta inadecuado, ya que la forma típica de uso es detener el programa y ejecutarlo por pequeñas secciones dentro del debugger, con lo que cualquier temporización queda destruida. Por eso en este tipo de casos la práctica recomendada es, como decimos, instrumentar el código fuente; que consiste en prepararlo de forma que en puntos claves de su ejecución presente información sobre qué está sucediendo (entrada o salida de métodos, valor de variables, etcétera). Java dispone a este fin del conocido paquete Log4Java, y es el que hemos usado por sus buenas prestaciones (velocidad, flexibilidad y facilidad de uso, que lo hacen más recomendable que las mismísimas APIs de logging incluidas en Java a partir de su versión 1.4).

La información que extraemos de diferentes partes del JMF es el estado de variables, parámetros con los que son llamados algunos métodos y su resultado... concretamente, y que sean relevantes para la temporización, extraemos los *timestamps* y algunos flags de los paquetes de información que forman los flujos multimedia, a medida que son capturados, procesados, transportados y reproducidos; de forma que luego podemos hacernos una buena idea de qué ha ido sucediendo durante la ejecución de los programas.

Y ahora aprovechamos esta instrumentación ya preparada para recoger datos de la ejecución durante la reproducción de un fichero grabado en formato AVS con uno de nuestros prototipos, que simplemente clona el DataSource de captura y lo graba, tal como se ha explicado anteriormente. Para reducir las variables implicadas en el proceso, el prototipo utilizado es lo más básico posible: captura, clona los DataSources llegado el momento (cuando el usuario activa el GUI), reproduce y graba. Elimina el resto de funcionalidades: no transmite los flujos por RTP, por ejemplo. Además el sistema en que se realizó la grabación es un buen prototipo (incluso generoso) de lo que podría esperar encontrarse para nuestra aplicación de videoconferencia: Windows XP ServicePack 2, con procesador Pentium 4 y 1 GB de RAM, sin ejecutar nada en especial aparte del programa de grabación (incluso con salvapantallas y similares desactivados). Esta grabación consistió en 14 minutos de audio en formato GSM y video en formato XVID, con formato contenedor AVS.

A continuación podemos ver las gráficas obtenidas y algunas conclusiones importantes que se pueden extraer.

Para empezar, la resta del *timestamp* del *frame* reproducido menos el *timestamp* del momento real en que el *frame* se reprodujo nos da el retraso del *frame* respecto al momento en que se debió reproducir. Representaremos el resultado de esta resta con una línea roja en las siguientes gráficas. Idealmente este retraso se mantendría siempre muy próximo a cero; podemos ver a continuación la gráfica correspondiente a una reproducción correcta.

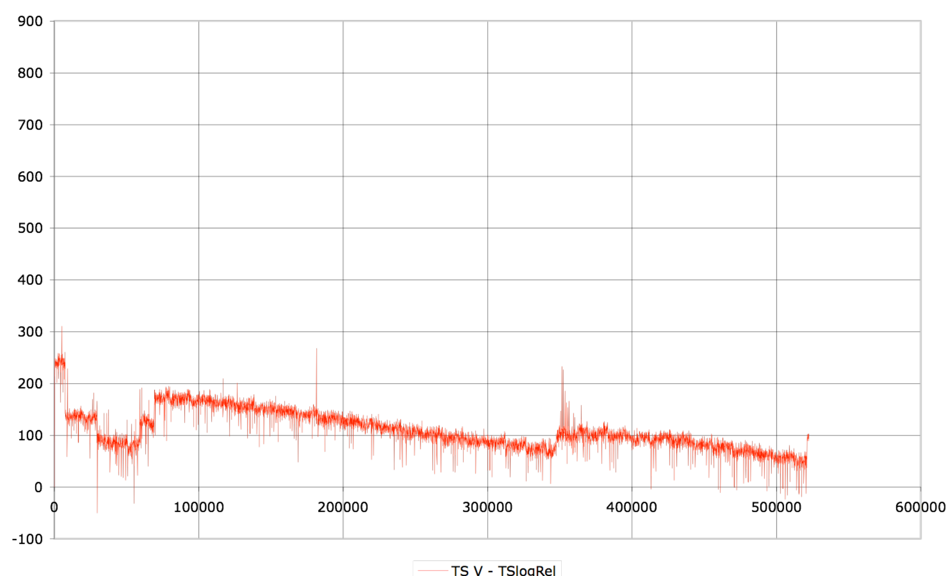

Figura 21 - Retardo de frame en una reproducción (relativamente) correcta con JMF

Podemos ver en la Figura 21 cómo el retardo de *frame* se mantiene razonablemente pequeño en toda la duración de la reproducción. Constituye un ejemplo de lo que se puede considerar una reproducción relativamente buena. Aun así, esta gráfica ya es alarmante para nuestro propósito de la reproducción simultánea de dos grabaciones, por dos razones:

— El retardo se mantiene alrededor de los 200 ms. durante minutos

— El retardo fluctúa durante toda la reproducción

Lo primero hará que la reproducción simultánea de dos grabaciones presente una desincronía equivalente potencialmente al doble de lo que presenta una de ellas. Es decir: en este ejemplo, puede suceder que un interlocutor haga una pregunta y la contestación llegue unos 400 ms después de lo normal; o, peor aún, 400 ms antes. 400 milisegundos es un retardo relativamente pequeño, y sería soportable; pero podemos recordar que un retardo mayor de unos 150 ms en telefonía se considera molesto. Además, en telefonía siempre se trataría de retraso, mientras que en el caso de la reproducción podría tratarse también de adelanto.

Y lo segundo, la fluctuación del retardo, provocaría que la variación (el retardo o adelanto) vaya cambiando durante la reproducción. Puede verse que la variación

tiende a cero en este ejemplo concreto, pero puede saltar. Se puede decir en realidad que esta fluctuación es peor que si la variación fuese constante, ya que no permite al espectador acostumbrarse.

Y sin embargo, como decimos, esa era una gráfica correspondiente a una reproducción relativamente correcta. Veremos ahora cuál es el caso típico en la reproducción con JMF.

En la siguiente gráfica, volvemos a representar el retraso de *frame* con una línea roja. Podemos ver que se aleja limpiamente de 0. De hecho se puede aproximar linealmente como una pendiente. Para nuestra grabación de 14 minutos la reproducción dura unos 16 minutos; el retraso final es de unos 115 segundos, lo que representa más de un 10% del tiempo grabado; y curiosamente, podemos decir que aun así es suficientemente pequeño como para que no sea apreciable en grabaciones cortas y que no se comparen con ninguna base de tiempos. Pero resulta devastador para nuestros propósitos.

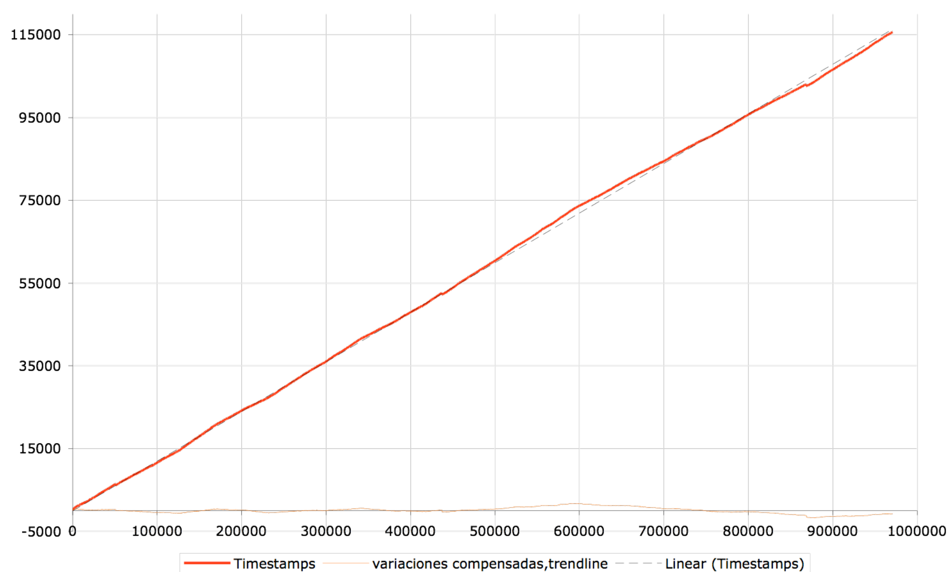

Figura 22.- Retardo de frame en una reproducción normal con JMF

Como se ha indicado, la simple existencia de esa línea roja con una pendiente diferente de 0 implica que la reproducción en el JMF no está cumpliendo la velocidad de reproducción que debería. Una primera idea que puede surgir es que podemos

intentar arreglarlo; al fin y al cabo que sea aproximable linealmente (como demuestra la línea negra punteada) significa que la reproducción mantiene un determinado número de *frames* por segundo casi constante. Esto significaría que dos grabaciones, aunque no cumplan su tiempo de reproducción, podrían sincronizarse entre sí si las dos van a la misma velocidad.

Pero se puede observar que hay pequeñas fluctuaciones en la línea roja que la llevan arriba y abajo de la línea punteada. En efecto, si restamos el retraso de *frame* (línea roja) de su aproximación lineal (línea punteada) tenemos lo que podríamos llamar un retraso de segundo orden:

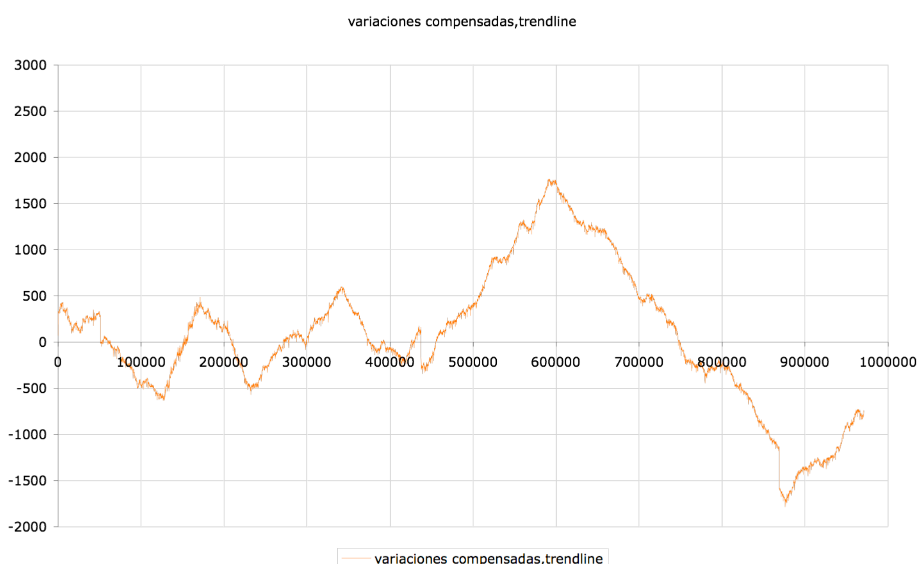

Figura 23.- Detalle del retardo de frame "de 2º orden" de una reproducción normal con JMF

Aquí podemos ver que esas fluctuaciones que apartan la reproducción de una velocidad constante pueden tener amplitudes de 1.5 segundos. Y si tenemos en cuenta que estas fluctuaciones estarían también presentes en la otra reproducción, hay que tomar entonces no la fluctuación de pico, sino posiblemente la “de pico a pico”, que es de 3 segundos; lo cual concuerda perfectamente con lo observado en las reproducciones simultáneas llevadas a cabo en el laboratorio.

¿Podría suceder que al menos estas variaciones sean siempre iguales, con lo que podríamos contar con ellas para conseguir la sincronización de alguna forma?

Para responder podemos observar qué sucede en otra reproducción de exactamente el mismo fichero, en exactamente la misma máquina, inmediatamente después de la primera reproducción:

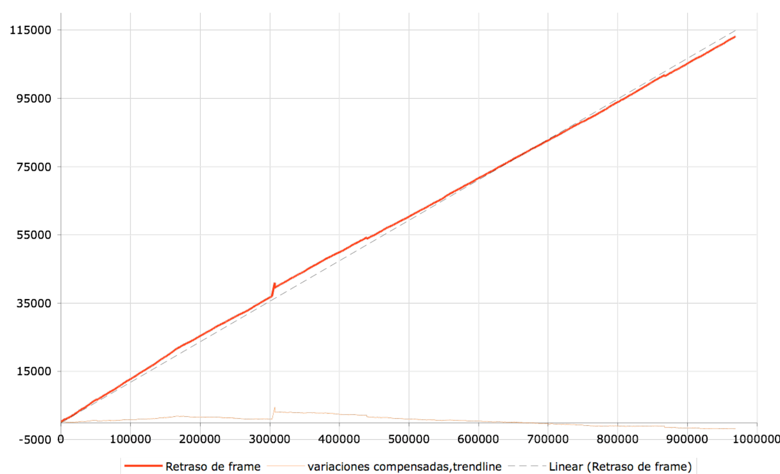

Figura 24.- Otro ejemplo de retardo de frame, a partir del mismo fichero

La respuesta es "no, no es siempre igual". Aquí las fluctuaciones son totalmente diferentes; la única constante es que siguen existiendo. De hecho, obsérvese cómo durante los primeros 30 segundos de reproducción el retraso aumenta más rápidamente que la aproximación lineal; de repente hay un parón en la reproducción que dura más de un segundo, e inmediatamente después la reproducción continua con una velocidad mayor que antes (y casi constante esta vez).

Por completitud, veamos de nuevo la resta del retraso de *frame* menos su aproximación lineal:

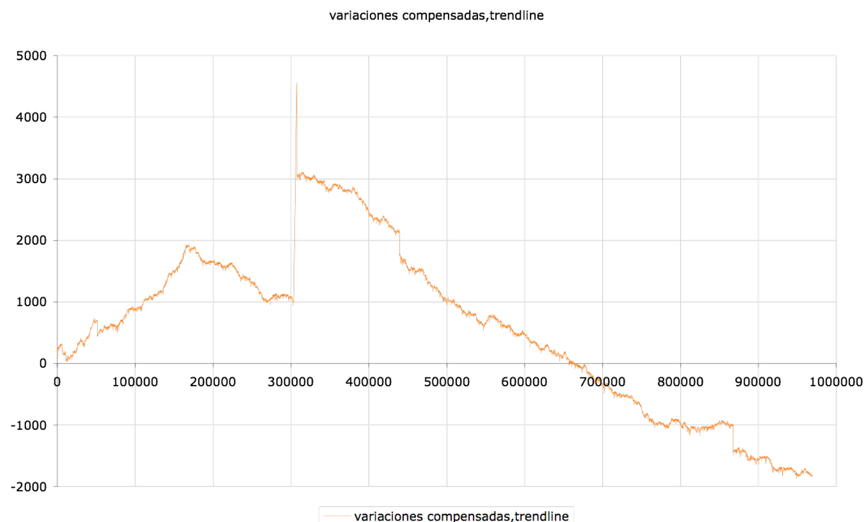

Figura 25.- Detalle del retardo de frame "de 2º orden" del 2º ejemplo

Otra pregunta que surge, naturalmente, es: ¿cómo es posible que no hayamos detectado esa variación de más de un 10% en la velocidad a simple vista? (o incluso "a simple oído"). Sin embargo, puede ser interesante considerar que la industria cinematográfica comúnmente ralentiza las películas más de un 4% al pasar del estándar PAL al estándar NTSC (por el proceso llamado *telecine*), y al contrario, las acelera también más de un 4% al pasar de formatos de 24 *frames* por segundo (cine) a los 25 fps del estándar PAL; y poca gente es capaz de notarlo, aunque ello influye en el tono de la música, por ejemplo (y por supuesto en la duración). **[2] [9] [10]**

Al menos todas estas experiencias nos pueden dar ideas de qué es lo que está fallando en el JMF. Por ejemplo, en esta última gráfica vemos que al principio la reproducción es "relativamente lenta", luego sufre un parón y luego sigue "relativamente rápida". ¿Podría deberse a que la memoria de la máquina virtual estaba demasiado llena y el Garbage Collector se ha visto forzado a actuar? Las medidas tomadas durante las ejecuciones de nuestros prototipos muestran que el Garbage Collector (mediante el uso de parámetros de debugging de la JVM como `-XX:PrintGCTimeStamps` o `-verbose:gc`) provoca pausas frecuentes (incluso varias por segundo) de hasta 3 ms, y eventuales de hasta 150 ms (!). Pero, incluso suponiendo que el Garbage Collector sea el culpable y que podamos evitar sus pausas, podemos ver que, aunque después del escalón el rendimiento es mejor, aún es deficiente: sigue sin concordar con la velocidad esperada de la grabación, y sigue sin ser siquiera una velocidad constante.

Si seguimos buscando posibles causas, otra que se nos presenta en plataformas Windows es la variabilidad de las interrupciones de reloj, que por defecto tendrían una periodicidad de 15 o 10 ms, y pueden configurarse a 1 ms de forma extremadamente simple. Las APIs de Windows permiten que *una aplicación cualquiera* aumente la frecuencia de estas interrupciones para conseguir más resolución; no permite reducirla. Pero el mayor número de interrupciones provocará que el rendimiento baje, y *afectará a todos los procesos* de la máquina. La frecuencia de interrupciones podría volver a su valor normal cuando la aplicación que hizo la petición se cierre, o cuando el sistema entre en reposo o sea reiniciado **[11]**. Todo esto concuerda con nuestras observaciones sobre la aparición esporádica de reproducciones con JMF a la velocidad correcta; pero implicaría que debemos tomar el control sobre la precisión de temporización en las máquinas Windows en las que se ejecuten nuestras aplicaciones. Además, pruebas preliminares muestran que, efectivamente, simplemente añadiendo el pequeño cambio de 2 líneas en alguno de nuestros prototipos (según se indica en **[11]**) conseguimos mejoras consistentes en la velocidad de reproducción y en la grabación, llegando a 1 segundo de desincronía en 35 minutos de grabación a 15 *fps*.

Otra posibilidad es que JMF no parece tener la "inteligencia" necesaria para reconocer que se está quedando atrasado en la reproducción y que debería acelerar, quizás saltando la decodificación o presentación de *frames*, para volver al momento adecuado de la reproducción. En eso consiste la funcionalidad de *framedropping* presente en muchos reproductores multimedia, desde Quicktime hasta mplayer; y la evidencia anecdótica apunta a eso, a que JMF no es capaz de ello; ya que hay ocasiones en que, si por alguna razón el video ha quedado muy retrasado (si se carga puntualmente al máximo el procesador y luego se libera, por ejemplo), para recuperar el momento actual lo que hace es acelerar la reproducción de todos los *frames*, en vez de simplemente ignorarlos para saltar hasta el momento actual.

Pero esto son simplemente ideas sobre lo que *podría* estar funcionando incorrectamente. El problema general es : ¿Valdría la pena estudiarlo y buscar en qué punto del código fuente del JMF se debería estar teniendo en cuenta cada uno de esos problemas, siempre suponiendo que sean ésos los problemas? ¿Cuánto costaría de

implementar, si es que eso lo arreglaría? Incluso, ¿se puede implementar con alguna garantía de buen funcionamiento?

Como se ha podido ver durante todo el desarrollo de este Proyecto Fin de Carrera, resolver un problema del JMF suele mostrar una infinidad de nuevos problemas. Hemos conseguido solucionar alguno de ellos, de forma que funcionalidad que antes no podía ser usada ahora es útil, lo cual ya ha posibilitado un número de mejoras en las aplicaciones actuales en uso. Pero hemos encontrado un número de problemas suficientemente serios como para impedir el uso del JMF en otras aplicaciones, principalmente relacionados todos ellos con irregularidades en el procesado de *timestamps* de los flujos multimedia; lo cual es quizás irónico, siendo que la razón de ser de una arquitectura multimedia es precisamente la habilidad de manejar conjuntamente un número de flujos basándose en sus *timestamps*.

Veremos en el siguiente capítulo algunas sugerencias sobre un posible camino a seguir para continuar delimitando los problemas de JMF y poder seguir usándolo o incluso modificándolo, para poder asegurar las funcionalidades que más nos interesan.

Capítulo 6

Conclusiones y trabajo futuro

Resumen

El COMM dispone de un sistema de videoconferencia desarrollado con Java y JMF (Java Media Framework). Se pretende añadir nuevas funcionalidades al sistema, concretamente grabación y reproducción de los flujos multimedia; pero dado el historial de crecientes dificultades presentado por el JMF, se hace imperativo estudiar la viabilidad de seguir usando el JMF y definir qué funcionalidades son practicables o no.

El presente Proyecto Fin de Carrera estudia por tanto las posibilidades de grabación multimedia con Java y Java Media Framework, con vistas a añadir esa funcionalidad al sistema de videoconferencia del COMM.

Para ello comenzamos estudiando las posibles configuraciones de equipos y red posibles. Procedemos después a desarrollar diversos prototipos de aplicación que implementen las funcionalidades deseadas en las dos configuraciones consideradas más interesantes, hasta llegar a una aplicación de videoconferencia con grabación para cada configuración, y otra aplicación para reproducir adecuadamente el par de grabaciones generadas en una videoconferencia.

Todo el proceso de desarrollo de estas aplicaciones es paralelo al proceso de perfilado de la aplicabilidad y problemática de JMF para cada tipo de funcionalidad,

requiriendo el recurso al limitado código fuente del framework publicado por Sun para corregir sus bugs y problemas de diseño. Continuamos además el desarrollo de extensiones al JMF que se desarrollaron anteriormente en el COMM, para circundar alguno de los problemas más graves del JMF y poder por ejemplo grabar y reproducir video y audio sincronizadamente.

Finalmente, encontramos que hay problemas de base suficientemente importantes en el JMF como para impedirnos reproducir sincronizadamente entre sí los pares de grabaciones que conseguimos con nuestras aplicaciones. Ello nos hace recurrir a reproductores de terceras partes (Quicktime, mplayer) para implementar un reproductor doble, capaz de reproducir de forma simultánea, sincronizada y unificada el par de grabaciones generado en una videoconferencia.

Para seguir delimitando las posibilidades del JMF, reaprovechamos lo aprendido y desarrollado para implementar la grabación de capturas de pantalla, también como funcionalidad a añadir al sistema de videoconferencia original. Desafortunadamente, encontramos que las extensiones del JMF desarrolladas en el COMM para la grabación han sido diseñadas para grabar desde dispositivos de video y deberán ser reorganizadas para esta nueva fuente de imagen.

Podemos entonces resumir los resultados de este PFC indicándolos en ***negrita cursiva*** en la tabla que se vió al final del capítulo 3:

Funcionalidad de JMF	**Status en JMF original**	**Status en COMM**
Captura	OK	En uso
De/codificación	Pocos codecs	Añadido MPEG4 (Xvid)
Transmisión RTP	*Timestamps* incorrectos	Corregido
Recepción RTP	OK	En uso
Reproducción	OK	En uso **(con reservas)**
Clonación de flujos	***Problemas de calidad y estabilidad***	***Corregido***

Funcionalidad de JMF	**Status en JMF original**	**Status en COMM**
Grabación	***Desincronía entre pistas***	***Corregido***

Tabla 2.- Resumen de resultados respecto al JMF

Recordando que las mejoras a investigar debían acabar dando forma las nuevas funcionalidades del sistema existente de videoconferencia, también podemos ahora reflejar los resultados en otra tabla.

Funcionalidad sistema videoconferencia	**Status**
Grabación de videoconferencia	***Usable***
Sincronización de grabaciones	***Usable***
Grabación captura de pantalla	***No implementable por problemas en wrapper Xvid / JMF***
Reproducción simultánea sincronizada	***Implementada; no usable por desincronías en JMF***

Tabla 3.- Resumen de resultados respecto al Sistema de Videoconferencia

Además del éxito de las implementaciones prácticas, creemos importante la información que hemos obtenido sobre los límites de aplicación de JMF. De esta experiencia podemos extraer un número de conclusiones y direcciones de trabajo futuro, tanto para seguir mejorando las funcionalidades que ya tenemos como para conseguir nuevas funcionalidades.

Trabajo futuro: sugerencias para continuar mejorando JMF

El principal resultado de este Proyecto Fin de Carrera es la información que hemos conseguido sobre cómo trabajar con el JMF: qué se puede y qué no se puede hacer, y

sobre todo qué hacer cuando algo no funciona como se espera. Hay que recordar que JMF no ha sido actualizado en más de 6 años, y la documentación que Sun ofrece sobre su uso es poca y en ocasiones incorrecta. La liberación del código fuente ayuda, pero ni siquiera se ha liberado todo el código fuente; con lo que hay que tener mucho cuidado al hacer modificaciones. En todo caso se puede afirmar sin miedo a equivocarnos que si no llega a ser por el acceso al código fuente, JMF hubiese sido descartado como una opción viable hace mucho tiempo.

De todas formas, JMF sigue siendo una alternativa diríamos que muy interesante para ciertos usos. Sin olvidar que hay un número de aplicaciones desarrolladas con Java y JMF que están funcionando, demostrando que ciertas funcionalidades sí que son usables. Además, recopilando: una arquitectura multimedia como es el JMF, usable en entornos 100% Java pero capaz de aprovechar codecs instalados nativamente en plataformas Windows, capaz de ejecutarse en Máquinas Virtuales de Java 1.1, con (parte del) código fuente publicado, y basada en *plug-ins* desarrollables por nosotros mismos para poder crear nuestros propios formatos de fichero o incluir funcionalidad nativos, es francamente un regalo del cielo para los intereses del COMM, que suelen incluir dispositivos portátiles como teléfonos móviles, PDAs, etc.

Entonces: ¿qué es posible con el JMF, y qué debería descartarse? Incluso, ¿hasta qué punto vale la pena seguir alterando el propio JMF para arreglar las deficiencias que presenta? ¿Son bugs, que permitirían un arreglo más o menos delimitado, o problemas de diseño, que implicarían grandes cambios?

Creación de herramientas

La primera propuesta sería estudiar en profundidad la evolución de un flujo multimedia a lo largo de diferentes puntos del JMF. Instrumentar el código fuente es muy útil, pero se hace necesaria alguna herramienta que nos permita comprobar sistemáticamente que el comportamiento de los flujos es el esperado. Por ejemplo, sería muy interesante disponer de extensiones al JMF que permitan inyectar un flujo multimedia sintético, de características perfectamente conocidas, al JMF; o de seguir la pista de un flujo por diferentes puntos del API hasta llegar a su reproducción,

comprobando su integridad. Esto eliminaría dudas sobre la fiabilidad de la calidad o los *timestamps* de los flujos en cualquier punto del proceso. Por ejemplo:

— A veces los *frames* mostrados por el reproductor muestran en sus *timestamps* periodicidades extrañas. ¿Es por el procesado en el JMF o es que la captura desde la webcam ya es así? ¿cómo de irregular es la aparición de *frames* en el interfaz de captura de video o audio del JMF con el Sistema Operativo?

— Sabemos que una webcam USB puede eventualmente no entregar un *frame* (por falta de ancho de banda en el bus, por ejemplo). Cuando esto sucede, ¿cómo reacciona JMF? Sabemos también que JMF es especialmente sensible a la temporización del audio, con lo que averiguar si sucede lo mismo en la captura de audio sería muy interesante.

— Habría que aclarar qué sucede ante algunos casos concretos de secuencias de *timestamps*, y cómo de comunes son esos casos. Paquetes de audio demasiado próximos; dependencia de si son flujos RTP (lo que podría afectar nuestro formato AVS)...

Algunas de estas preguntas podrían responderse profundizando en el código fuente, o aprovechando el código que ya hemos instrumentado para extraer logs del comportamiento; pero actualmente extraer los datos conseguidos en cada ejecución requiere un tiempo de procesado manual, y el hecho de que los fallos del JMF sean esporádicos multiplica el número de intentos necesarios. Seguramente valdría la pena entonces desarrollar herramientas que nos permitan saber en cualquier momento (incluso durante la ejecución del programa) qué está pasando y si ya han aparecido errores. Compárese con el proceso actual de

— grabación de un mínimo de 10 minutos extrayendo logs,

— reproducción de la grabación extrayendo logs (muchas veces repetida para asegurarnos de que hay o no hay problemas y cómo de frecuentes son), estando pendientes durante toda la duración para observar si se mantiene correctamente la sincronía,

— limpieza de los logs e importación en programas que permitan el análisis (como Matlab o Excel),

— análisis propiamente dicho.

Teniendo en cuenta que las modificaciones en el JMF o incluso en nuestros prototipos forzosamente han de ser iterativas y de pequeño alcance, nos encontramos con que cada pequeño cambio introducido necesita más de una hora de atenta dedicación para comprobar sus efectos. Añádase que algunos efectos no se notarán hasta que se acumulen un número de cambios (como fue el caso de ajustar el número de *frames* capturados antes de comenzar la grabación y empezar la grabación posteriormente a la captura), y nos encontramos con que la metodología seguida hasta ahora difícilmente permitirá seguir trabajando a este nivel con el JMF.

Nótese que JMF incluye algo ya ligeramente aproximado a las "extensiones de chequeo automático" que proponemos; pero lo que hace lo incluido en el JMF es mostrar la arquitectura interna de un player que está en funcionamiento. Seguramente se podría extender esta funcionalidad para que la información que provee sea más exhaustiva.

Aumentando la velocidad y la repetibilidad de las iteraciones de mejora del JMF sería mucho más fácil llegar a soluciones realmente fiables, con muchas menos incertidumbres; e incluso permitiría afrontar cambios más radicales en el JMF, que ahora mismo no nos atrevemos a emprender por la inseguridad de cómo reaccionaría el sistema en su conjunto.

Se trata, resumidamente, de aplicar la filosofía de *Unit Testing* (testeos automatizados, constantes y previos a la propia implementación) al desarrollo del JMF y de las aplicaciones que lo usan; porque no podemos suponer que las funcionalidades que ofrece estén funcionando como deben.

Documentación escasa y deficiente

La documentación oficial existente para el JMF es:

— Poca: limitada a la especificación del API, a una guía del usuario, y a una serie de ejemplos;

— Y deficiente: en el cambio de la versión 2.1 del JMF a la 2.1.1, parte del API para transmisión y recepción RTP fue hecha obsoleta para introducir nuevas clases y métodos. No sólo es un cambio inexplicable para una actualización de tercer orden (ya que tradicionalmente los cambios de API deberían ser como mínimo en el segundo número de versión, y posiblemente incluso en el primero), sino que la documentación no siempre lo refleja, con lo que es contradictoria en diferentes partes: la guía dice unas cosas, los ejemplos otras, y el API no se pronuncia.

Adicionalmente, existen un foro y una lista de correo oficiales sobre JMF, pero el tráfico es prácticamente nulo y se limita a estudiantes pidiendo ayuda para resolver prácticas en las que tienen que implementar un player con el JMF (lo cual por otra parte es uno de los ejemplos). **[5]**

Así que se hace prácticamente vital desarrollar una documentación propia del COMM en que se documenten los problemas encontrados, los rodeos o soluciones aplicados, los bugs corregidos, las formas de uso que encontremos que permiten que las aplicaciones funcionen como deben... etcétera. Si no, nos arriesgamos a que trabajo de semanas o meses se desaproveche por falta de "conocimiento histórico", e incluso que alguna solución sea redescubierta o reimplementada independientemente.

Y hay que afrontar que la principal fuente de información muchas veces será el propio código fuente. La forma de trabajo normal con un Framework sería investigar las guías o el API para resolver problemas o dudas; no así con el JMF, sino que tras un primer vistazo e intento de solución con API, guías y ejemplos, no hay que dudar en recurrir al código fuente. Por supuesto que esto implica mucho más trabajo y problemas, pero es la única forma de llegar a alguna conclusión, incluso aunque ésta sea que determinada funcionalidad no es usable. Por contra, quedarse en hacer pruebas a un nivel superior (limitándonos a recurrir a la guía y el API) nos dejará

seguramente con conclusiones y soluciones con problemas intermitentes, y por tanto nada robustas.

Se podría pensar que esta forma de trabajo es similar a la que se espera del código libre (open source). En realidad, el caso con JMF es peor; ya que el código ha sido publicado, pero no es libremente modificable. Mientras que un código realmente libre hubiese podido ser mantenido por terceros, JMF está efectivamente congelado, y el código es, en teoría, no más que otra forma de documentación. Por ello hay desarrolladores que opinan que Sun debería liberar el código oficialmente, en vez de limitarse a publicarlo.

Refinar y estandarizar trabajo ya hecho

Este punto podría incluirse en el anterior, pero es suficientemente importante como para tomarlo aparte.

Dado el muchas veces caótico código del JMF y la fragilidad del código que lo tiene que usar, se hace vital que documentemos perfectamente todo lo que toquemos; a ser posible, de forma que el propio código sea autoexplicativo. De hecho, en muchos casos podemos encapsular funcionalidades de los prototipos desarrollados en este Proyecto Fin de Carrera para que cualquier aplicación pueda implementarlas de forma prácticamente inmediata, y posiblemente sin problemas. Por ejemplo: preparar la captura de audio y video, reunir y comprimir los flujos multimedia y obtener los flujos resultantes puede ser relativamente complicado; y sin embargo toda esa funcionalidad es encapsulable en un sólo método estático, que podría ser inmune a cambios en el resto del programa. De esta forma obtenemos todas las ventajas del JMF y el trabajo ya hecho, y ninguno de los inconvenientes.

Otros casos de uso encapsulables serían la grabación en un fichero, la transmisión RTP, la clonación...

Además, estos casos de uso servirían a la vez “ejemplos vivientes” de cómo afrontar ciertas tareas con JMF con garantías con de éxito.

Por otro lado, hay un número de soluciones implementadas en nuestras modificaciones del JMF y nuestros prototipos que no son generales, sino bastante específicos para los problemas que pretendían solucionar en su momento. Y sin embargo esas mismas soluciones harán falta cuando las condiciones cambien ligeramente. Se haría pues necesario, si no generalizar esas soluciones, sí al menos documentar qué casos específicos se han cubierto y cuáles aún encontrarán problemas; así, cuando aparezcan esos problemas, se sabrá que la solución está casi implementada ya, y que no se trata de algo nuevo que investigar.

Ése es precisamente el caso del codificador XVID y del packetizador XVID/RTP, donde hemos podido ver por qué es necesario este tipo de documentación. Codificador y packetizador pueden ser usados perfectamente por separado en fuentes de video provenientes de una cámara, pero dan problemas cuando capturamos desde otras fuentes, como es la captura de pantalla. En este caso, el codificado y la packetización inmediata funcionan, pero no hay forma de instanciar por separado codificador y packetizador. El problema está en que la implementación del codificador no acaba de seguir la forma canónica del API, sino que tiene pequeñas particularidades para su instanciación, y el packetizador contiene ese conocimiento; con lo que si se delega su instanciación al packetizador, funcionará, pero no de otra forma.

Formatos

Una vez más hay que hablar de los formatos de grabación. MOV parece ser el formato con más futuro: es un estándar y parece sufrir de relativamente pocos problemas. Sin embargo AVS nos permite hacer grabaciones individuales de larga duración sin aparición de retardos. ¿Qué opción tendría más futuro: seguir adaptando poco a poco un formato propio como AVS, que da un 90% de la funcionalidad que necesitamos pero es básicamente un hack para obligar al JMF a hacer lo que queremos y que por tanto puede o podría llegar un momento en que se haga insostenible; o afrontar la mejora del multiplexor MOV?

Incluso estaría la opción de recurrir a otras librerías para acceder a ficheros en otros formatos; desde Quicktime for Java hasta implementaciones nativas o incluso en Java de las librerías comunes en el mundo open source (ffmpeg, VLC, ...).

Como mínimo, nuevamente habría que documentar los problemas que hay en cada formato. ¿MOV funcionaría para grabar sólo video o sólo audio? ¿Y AVS? MOV se comporta perfectamente para seeking y reproducción repetitiva, no así AVS. ¿Se podría arreglar, o no es tan importante esa funcionalidad?

Investigar la temporización de la JVM

Como se ha comentado en el capítulo de Detalles de Implementación, hemos encontrado en la conclusión de este PFC información relativamente reciente sobre cómo interactúan la temporización de la JVM y la de las APIs de Windows. Pruebas preliminares muestran que efectivamente podemos conseguir que JMF aumente la precisión en la reproducción, llegando a velocidades que parecen correctas; pero parece ser dependiente de versiones de la JVM y de equipos. En todo caso, la solución propuesta en **[11]** es suficientemente simple e inocua como para ser implementada por defecto en cualquier aplicación que quiera beneficiarse de buena resolución temporal en Java. Sería enormemente interesante hacer las pruebas necesarias para comprobar hasta qué punto esta solución realmente lo es. Concretamente, la grabación no mantiene una velocidad constante, sino que puede fluctuar continuamente entre valores de 12 y 20 *fps* en menos de un minuto para una grabación que debería efectuarse en teoría a 20 *fps*. Nótese que esto no debería ser un gran problema en realidad mientras se graben los timestamps adecuados, como se supone que hace MOV (si el multiplexor de JMF cumple correctamente la especificación); pero es una buena razón por la que AVI no puede funcionar, ya que no guarda timestamps sino duraciones medias. Además, hay que recordar que la fluctuación puede ser debida hasta cierto punto por irregularidades en el *framerate* entregado por la webcam.

Estudiar integración con otras librerías, frameworks, etc.

Por último, y de forma más radical, hay que plantearse si, ya que se habla de otras librerías, no tendría sentido dejar JMF y usar alguna de las alternativas. Las hay desde pure Java (FMJ, diseñado además para ser un substituto directo de JMF) hasta wrappers Java alrededor de librerías nativas (Quicktime for Java); pasando incluso por librerías nativas pero altamente portables (ffmpeg). Dependiendo del entorno esperado en las aplicaciones y de la funcionalidad que se necesite, podría valer la pena hacer el cambio. Por ejemplo, algunas de las limitaciones serían que Quicktime for Java sólo existe para Windows y Mac; que muchas de las librerías alternativas no tienen soporte para captura; otras no funcionarían en alguno de los sistemas que nos interesan.

En todo caso, es un tema que considerar seriamente. Si la meta inicial del uso de JMF era disponer de una infraestructura multimedia que permitiese el desarrollo rápido de aplicaciones, JMF necesita serios cuidados para serlo.

Capítulo 7

Bibliografía

[1] *« Essential JMF»*, Prentice-Hall
http://www.amazon.com/Essential-JMF-Java-Media-Framework/dp/0130801046/ref=si3_rdr_bb_product/002-0578034-4702457

[2] Documentación del reproductor open source Mplayer.
http://www.mplayerhq.hu/DOCS/HTML/en/menc-feat-dvd-mpeg4.html#menc-feat-dvd-mpeg4-preparing-encode-material

[3] *«Java Media Framework 2.0 Guide»*, Sun Microsystems, Noviembre 1999
http://java.sun.com/products/java-media/jmf/2.1.1/guide/index.html

[4] *«JMF Solutions»*, Sun Microsystems Website,
http://java.sun.com/products/java-media/jmf

[5] *«JMF Interest List Archive»*, Sun Microsystems Website,
http://archives.java.sun.com/archives/jmf-interest.html

[6] Documentación de `System.nanoTime()` en el API de Java 5
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/System.html#nanoTime()

[7] Bug report: *System.currentTimeMillis() granularity too coarse*
http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4423429

[8] *«My kingdom for a good timer!»* , JavaWorld.com
http://www.javaworld.com/javaqa/2003-01/01-qa-0110-timing_p.html

[9] *«Telecine: Frame rate differences»,* Wikipedia
http://en.wikipedia.org/wiki/Telecine#Frame_rate_differences

[10] *«Temporal rate conversion: The nature of video and how it's displayed»*, Microsoft.com, Windows Hardware Developer Central
http://www.microsoft.com/whdc/archive/TempRate.mspx#E3

[11] *«Inside the Hotspot VM: Clocks, Timers and scheduling events»,* David Holmes' Blog, Sun Microsystems.
http://blogs.sun.com/dholmes/entry/inside_the_hotspot_vm_clocks