ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación:

INGENIERO DE TELECOMUNICACIÓN

Título del proyecto:

IMPLEMENTACIÓN DE UNA APLICACIÓN SOFTWARE PARA LA CREACIÓN Y GESTIÓN DE UNA ESCALETA DE VÍDEOS Y APLICACIONES MHP EN UN CANAL DE TELEVISIÓN DIGITAL

Dña. Elena Gadea Aransay

Dr. Mikel Sagüés García

Pamplona, 5 de Marzo de 2010

Agradecimientos

En primer lugar quiero agradecer a Mikel que siempre ha creído en este proyecto (hasta en los momentos en que a mí me entraban las dudas), muchas gracias por no dejar de animarme en ningún momento. Gracias también por estar siempre disponible, incluso desde las antípodas…, me he sentido muy bien trabajando contigo.

Una mención especial merece toda la gente de los CES, en primer lugar Raúl que ha sido mi "jefe" en este tiempo que he pasado por allí, muchas gracias por estar siempre dispuesto a ayudarme y a resolver todas las dudas que me iban surgiendo (que al principio eran muchas), ojalá que todos los jefes que me encuentre de aquí en adelante sean como tú. No puedo olvidarme de todos "mis compañeros de trabajo": Edurne, Rosa, Goretti, Kike,… porque me acogieron como a una más desde que aterricé en CEIN y han hecho que este proyecto sea mucho mas llevadero, siempre dispuestos a tomar un café (o unas cervezas) en los momentos de bloqueo.

Muchas gracias a mis amigas Cristina, Susana, Sara, Elena,… porque siempre están ahí, dispuestas a ayudarme a desconectar cuando llegan los momentos de estrés, y a celebrar los éxitos (porque los buenos momentos siempre quiero celebrarlos con ellas).

Mi agradecimiento más especial es para mi "compi" Aintzane, para mi has sido mucho más que una compañera de piso y más que una amiga, sin ti no lo hubiera conseguido. Gracias por los animos que siempre me das y por ayudarme siempre que lo he necesitado, pero sobre todo muchas gracias por aguantarme en los peores momentos de estrés….

No puedo olvidarme de mi familia, el pilar de mi vida y los que han hecho posible que haya llegado hasta aquí. Gracias a mis padres por todo el apoyo que he recibido siempre de ellos (hiciera lo que hiciera) y por todos los sacrificios que han tenido que hacer, espero que hayan merecido la pena…. Gracias a mi hermano y a Cari (para mí ya eres de la familia) por animarme a seguir adelante con esto y por estar siempre dispuestos a ayudarme, ya sabéis que sin vuestra ayuda no hubiera podido terminar este proyecto.

Muchísimas gracias a todos,

Elena

Índice

INTRODUCCIÓN

Este proyecto se ha desarrollado en el Centro de Excelencia Software Java - Open Source de CEIN (Centro Europeo de Empresas e Innovación de Navarra). El CEIN es un instrumento del Gobierno de Navarra al servicio de las PYMES y emprendedores, que persigue consolidar y diversificar el tejido industrial y económico de la Comunidad Foral. Para tal fin, fomenta el espíritu emprendedor, identifica, promueve y desarrolla proyectos empresariales e impulsa la innovación en las empresas navarras.

En este ámbito se establecen los Centros de Excelencia Software (CES), que son organismos sin ánimo de lucro impulsados por el Gobierno de Navarra y gestionados por el CEIN. El objetivo de estos centros es mejorar el conocimiento y la calidad de desarrollo de profesionales de empresas de software de Navarra. En concreto, el Centro Java – Open Source es un centro de difusión de tecnología y herramientas, tanto de desarrollo como de herramientas disponibles basadas en tecnologías Java y Open Source.

Una de las líneas de trabajo de este Centro se centra en la televisión digital terrestre por lo que dispone de un laboratorio de pruebas muy completo que permite estudiar la cadena completa de un sistema de transmisión-recepción de videos. Este laboratorio posibilita la transmisión de contenidos (tanto videos como aplicaciones) mediante una solución mixta de Hardware + Software. Por un lado, la parte de Software es un laboratorio multiplexor de vídeos y aplicaciones, en concreto, se trata del ***iMux Multiplexer*** de la empresa alemana MIT-Xperts, que permite el envío del *stream* de vídeo y datos a la velocidad correcta. Por otro lado, la parte de hardware, adapta y envía el *Transport Stream* generado por el software a un formato que pueda entender el *Set Top Box*. Este sistema hardware consiste en una tarjeta moduladora que envía la señal mediante un cable coaxial al STB. Además del sistema de transmisión, se dispone de varios decodificadores con distintas características que permite estudiar el comportamiento de distintos receptores a una misma transmisión. De este modo, se buscan soluciones lo más generales posible, es decir, que dependan lo menos posible del tipo de receptor que interprete la señal.

En esta línea de trabajo se ubica el proyecto propuesto por el Ayuntamiento del Valle de Aranguren a la UPNA. En este proyecto se pide una propuesta de soluciones para la inserción de un canal de televisión propio del Ayuntamiento en la red de cable del Valle de Aranguren y de posibles aplicaciones de televisión digital interactiva a desarrollar. Dicho proyecto se ha llevado a cabo en colaboración entre el departamento de Ingeniería

Eléctrica y Electrónica de la UPNA y los Centros de Desarrollo Software (CES) de Navarra que están integrados dentro del CEIN.

Para dar una solución a la propuesta de proyecto hecha por el Ayuntamiento del Valle de Aranguren y aprovechando el equipamiento ya disponible en CEIN, se pensó en usar el **iMux Multiplexer** como núcleo de la cabecera de transmisión para la red de cable del Valle de Aranguren. Aunque en CEIN se utiliza como laboratorio de pruebas, este equipamiento se puede emplear para difundir contenidos de forma real ya que es una herramienta profesional, testeada y de garantías.

Como ya se ha comentado el **iMux Multiplexer** es un laboratorio de multiplexación de vídeos y aplicaciones MHP. Este laboratorio supone un sistema de *playout* completo que tiene muchas ventajas:

- Permite crear servicios (incluyendo un *Transport Stream* con vídeo y audio) al que se añaden las aplicaciones MHP desarrolladas.
- Permite insertar y testear aplicaciones MHP en cualquier tipo de decodificador.
- Permite testear aplicaciones sincronizadas con la programación
- Permite insertar aplicaciones MHP de forma rápida y ágil.
- Dispone de una **Java API** bastante completa que permite la comunicación con el laboratorio desde una aplicación externa implementada en lenguaje Java.
- Dispone de una interfaz web (**iMux Manager**) que permite gestionar el laboratorio desde cualquier máquina con conexión a Internet.

Pero no sin inconvenientes:

- La precisión de la escaleta (*Schedule*) creada desde el **iMux Manager** es de un minuto. Esta precisión no sirve para la gestión de una escaleta de videos.
- No implementa ***Schedule Stream Events***. El usuario debe enviar los *Stream Event* de manera manual en los instantes en que se desee, de modo que si se quiere sincronizar alguna aplicación, debe haber alguien encargado de enviar los eventos en los momentos adecuados.

- No permite conocer el tamaño de los vídeos una vez que se han guardado en la base de datos del laboratorio.

- El **iMux Manager** no elimina los *Schedule* de aplicaciones una vez que han expirado. De modo que si el usuario no los borra, se irán acumulando en el laboratorio.

- El gestor de la escaleta de vídeos del **iMux Manager** provoca un *blackout* entre videos que interrumpe la continuidad de la reproducción de la escaleta.

En principio, esta opción parecía una solución sencilla y muy válida, pero al profundizar en el **iMux Manager** se comprobó que esta interfaz no cumple con las necesidades requeridas por el proyecto propuesto. Como ya se ha comentado en los inconvenientes del laboratorio, la precisión de un minuto ofrecida por esta interfaz no es suficiente para programar una escaleta de reproducción, ya que para que la reproducción sea continua se requiere al menos de una precisión de segundos. Además, es posible que se quieran programar videos de duración inferior a 1 minuto (por ejemplo, anuncios) lo cual no se puede realizar utilizando el **iMux Manager**.

Debido a estas limitaciones se planteó el objetivo de este proyecto final de carrera, que consiste en diseñar un software para la gestión de contenidos del canal de televisión. Dichos contenidos constan de la parte audiovisual, es decir los videos que se van a emitir en el canal, y también de aplicaciones interactivas como complemento a los contenidos audiovisuales. Esta herramienta también implementa la generación de *Stream Events* programados. Este software hace de intermediario entre el usuario y el laboratorio, solventando las limitaciones impuestas por este último. El software utiliza la Java *API* que proporciona el laboratorio para comunicarse con él y así poder implementar todas las funcionalidades necesarias para la correcta gestión del canal de televisión digital.

El software desarrollado en este proyecto está implementado en lenguaje Java, este lenguaje se creó con el propósito de que fuera independiente de la plataforma en que vaya a ejecutarse. Esto significa que un programa de Java, como el que se ha implementado en este proyecto, puede ejecutarse en cualquier máquina o plataforma. El único requisito necesario para ejecutar un software implementado en Java es que la máquina tenga implementada una JVM (Java Virtual Machine), es decir, un intérprete de Java. Otra característica de este lenguaje, y que por tanto también se puede aplicar a ese software, es que se trata de un lenguaje *multithreaded*, es decir, que puede soportar múltiples *threads* (puede ejecutar diferentes hilos al mismo tiempo). Esta característica es muy útil en este

caso, ya que el software requiere la implementación de varios hilos para su correcto funcionamiento. Este software se ha desarrollado como una aplicación de escritorio, cuya interfaz gráfica se implementa haciendo uso de ***Swing,*** uno de los paquetes gráficos de los que dispone Java.

En lo que se refiere a la estructura del software, se puede decir que se divide en tres bloques o acciones:

- Gestión de vídeos: Esta parte se dedica a gestionar tanto la lista de vídeos disponibles como la escaleta de reproducción. Al entrar en este bloque, el software obtiene los tamaños de todos los vídeos que hay guardados en la base de datos del laboratorio. Como se ha comentado antes, este dato no está disponible en el laboratorio de modo que se implementa en el nuevo software. Tras cargar los datos de los vídeos se entra en la gestión de vídeos, desde aquí se tiene acceso a funcionalidades básicas como son añadir o borrar vídeos. En este bloque también se encuentran las funcionalidades dedicadas a la configuración y reproducción de la escaleta de vídeo. Esta parte sustituye por completo el gestor de Schedule de vídeos del **iMux**, solucionando las dos grandes limitaciones del laboratorio: por un lado se consigue rebajar la precisión de la escaleta hasta los milisegundos. Por otro lado, se consigue la continuidad de la reproducción eliminando el *blackout* entre la reproducción de dos vídeos contiguos. Gracias a la propiedad *multithreaded* de Java, en este bloque se han implementado varios hilos que se encargarán de liberar al programa principal de las tareas más pesadas.

- Gestión de aplicaciones: Este bloque se encarga de las aplicaciones MHP del canal. Por un lado se gestiona la creación y la eliminación de las aplicaciones que están disponibles para el usuario. Por otro lado se gestiona la escaleta de aplicaciones propiamente dicha, en esta parte se implementan funcionalidades como la creación de *Schedule* de aplicaciones, o la actualización de la escaleta. Esta funcionalidad se crea para solventar la limitación del laboratorio comentada anteriormente, de tal forma que se borran todos los *Schedule* de aplicaciones que han expirado.

- Generación de *Stream Events*: En ese bloque se implementa una aplicación MHP que se sincroniza con los contenidos haciendo uso de los *Stream Events*. Esta aplicación se encarga de mostrar al inicio de cada vídeo un *banner* con el

nombre del vídeo que comienza a reproducirse. Para que este *banner* se muestre la aplicación debe recibir un *Stream Event* que le indique que comienza una nueva reproducción. El software que se ha desarrollado se encarga de enviar los eventos al inicio de cada reproducción de manera automática. Los *Stream Event* enviados son de tipo ***Do it now***, ya que el laboratorio no soporta el envío de ***Schedule Stream Event***, debido a la forma en que se ha implementado el software al crear la escaleta de vídeos, se está configurando el envío de *Stream Events* programados, ya que mientras haya una escaleta en reproducción, el software enviará un *Stream Event* cada vez que se inicie la reproducción de un vídeo nuevo. De esta forma se consigue implementar el envío de ***Schedule Stream Event*** aunque el laboratorio sólo permita la generación de *Stream Event* de tipo ***Do it now***.

Esta memoria consta de 4 capítulos, el primero de ellos es esta introducción. En un segundo capítulo se describe el código generado para la implementación de la aplicación. Como ya se ha comentado esta aplicación se ha desarrollado en lenguaje de programación Java. En este capítulo, denominado Descripción del código, se explica tanto la implementación de las distintas funcionalidades que se le han dado a la aplicación, como la parte de programación gráfica que describe cómo se ha creado la interfaz desde la cuál el usuario tendrá acceso a esas funcionalidades. Para esta parte de programación gráfica se ha utilizado la librería **Swing** de Java, que se explicará con más detalle a lo largo de este capítulo.

El capítulo 3 corresponde al Manual de usuario. En este apartado, se hace una descripción de la aplicación desde el punto de vista del usuario. En principio, la persona que vaya a hacer uso de esta aplicación no tiene porque conocer los detalles de la implementación a nivel de código de la aplicación. Tampoco es necesario que tenga conocimientos de transmisión de videos y aplicaciones a través de una red de distribución hasta llegar a los receptores. Por estos motivos, se ha pretendido que la aplicación tenga una interfaz amigable e intuitiva. Este manual de usuario, es una guía de los pasos que debe seguir el usuario para hacer un correcto uso de la aplicación.

Por último, se incluye un capítulo que contiene las conclusiones y las posibles líneas futuras del proyecto. En esta parte se hará una valoración general del proyecto y se comentarán las conclusiones técnicas a las que se ha llegado. También se comentarán las líneas futuras posibles que puede seguir este proyecto, es decir, cómo se puede mejorar lo ya hecho y el desarrollo de módulos complementarios a esta aplicación.

Además de estos 4 capítulos, la memoria incluye un anexo que completa la información detallada en la memoria. Este anexo ofrece una visión sobre la codificación de vídeos, y en concreto, las características que estos deben cumplir para que se puedan utilizar en esta aplicación y su reproducción sea la adecuada en los receptores. También se hace un estudio de las herramientas que se pueden usar para la conversión de distintos formatos de vídeo a *MPEG Transport Stream*, que es el utilizado por el software.

1.- INTRODUCCIÓN A LA PROGRAMACIÓN DE LA APLICACIÓN

En primer lugar se va a comentar como está estructurado el proyecto java en el que se ha desarrollado la aplicación. Este proyecto está dividido en tres paquetes:

- ***com.cein.aranguren.scheduler***: En este paquete se encuentra la clase principal desde la que se lanza la aplicación.
- ***com.cein.aranguren.scheduler.control***: En este paquete se guardan todas las clases que implementan las funcionalidades de la aplicación.
- ***com.cein.aranguren.scheduler.gui***: En este paquete se tienen todas las clases que implementan las partes graficas de la aplicación.

De esta forma, la distribución de las clases que componen la aplicación queda como se puede ver en el **Diagrama 1.1**.

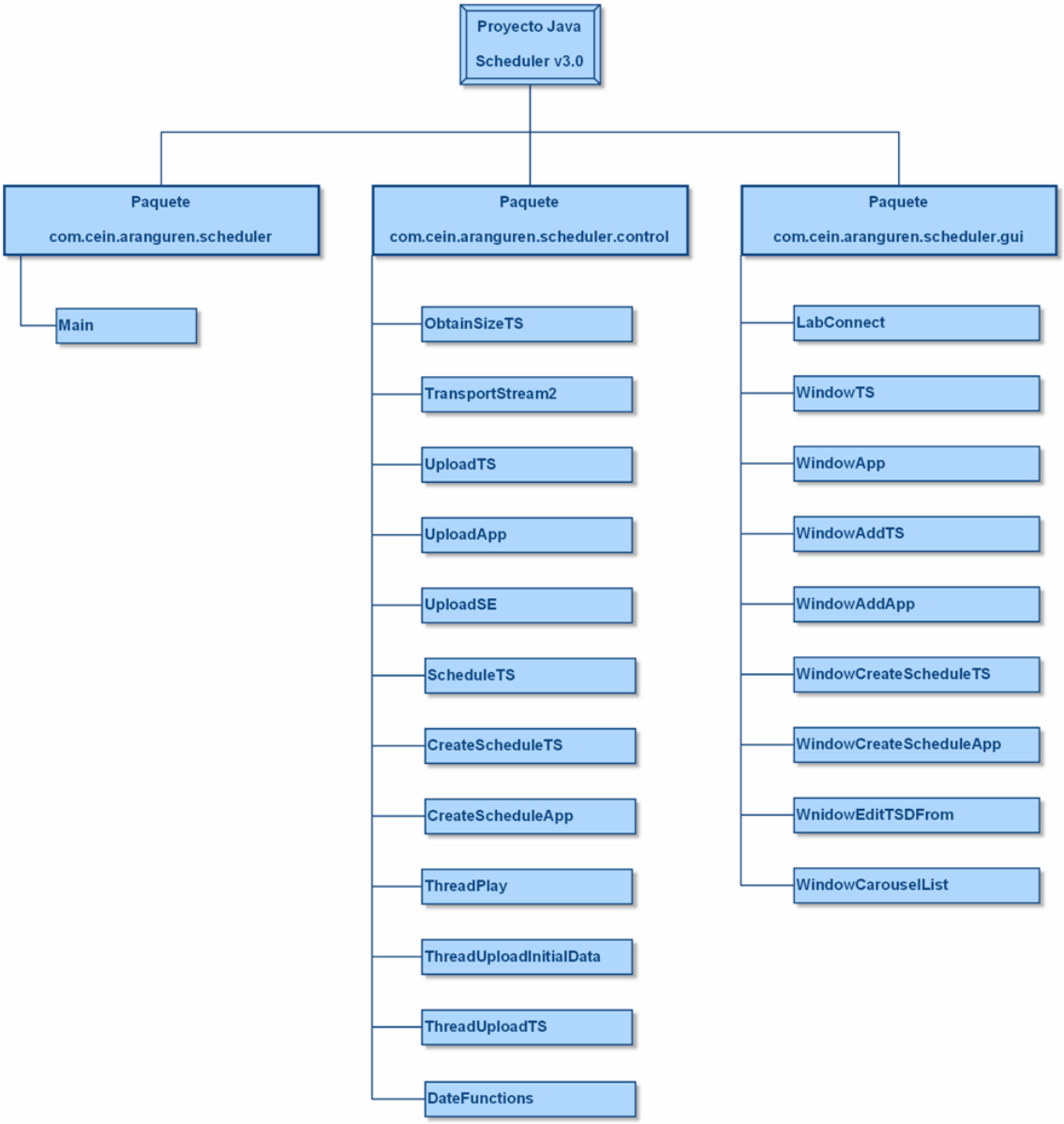


Diagrama 1.1.- *Esquema de clases del proyecto*

Para que todo funcione correctamente es necesario añadir al *Build Path* [1]del proyecto las siguientes librerías:

- ***imux-api.jar***
- ***imux-manager.jar***
- ***jsch-0.1.42.jar***
- ***pgsql-jdbc.jar***

Una vez que se conoce la estructura del proyecto se pasa a describir como se ha implementado la aplicación.

Se trata de una aplicación de escritorio que permite al usuario gestionar un *Schedule* de contenidos en un canal de televisión digital. En estos contenidos se tienen por un lado los vídeos que se van a reproducir que componen la parte más visible de los contenidos. Por otro lado, como se trata de un canal digital, se tiene la posibilidad de incluir aplicaciones interactivas como un valor añadido a los contenidos audiovisuales. Por último, se ha incluido en el proyecto una demostración del uso de *Stream Events* para la sincronización de las aplicaciones interactivas con los contenidos audiovisuales. Esta demostración consiste en la implementación de una sencilla aplicación MHP que responde a la recepción de eventos en el receptor.

Por lo tanto, se puede decir que la aplicación se divide en tres bloques o acciones. Estos bloques son:

- Gestión de vídeos
- Gestión de aplicaciones
- Generación de *Stream Events*.

Esta aplicación trabaja sobre un laboratorio multiplexor, se trata del ***iMux Multiplexer*** (versión 1.4.0) de la empresa alemana MIT-Xperts. Es un laboratorio de multiplexación de vídeos y aplicaciones MHP. Este laboratorio permite crear servicios (incluyendo un *Transport Stream* con vídeo y audio) al que se le añaden las aplicaciones MHP desarrolladas. También permite insertar y testear aplicaciones MHP en cualquier tipo

[1] Desde el *Build Path* de un proyecto Java se establecen las clases que debe compilar, los recursos (directorios, ficheros, JAR, …) que debe tener en cuenta para compilarlo, etc.

de decodificador y, por supuesto, genera y envía *Stream Events* para posibilitar la sincronización de las aplicaciones.

El laboratorio se puede manejar vía Web con la interfaz de usuario de que dispone el propio laboratorio (***iMux Administrator***) o bien haciendo uso de la ***iMux Java API*** que se proporciona también con el laboratorio. En este proyecto, se utiliza esta última (iMux Java API) con la que se hace posible la comunicación entre la aplicación de escritorio desarrollada y el laboratorio.

Debido al uso de este laboratorio, antes de tener acceso a los distintos bloques de la aplicación se debe realizar la conexión con el laboratorio, para que éste sea manejable desde la aplicación de escritorio.

Como ya se ha comentado la aplicación se lanza desde la clase principal (***Main.java***), en esta clase lo único que se hace es definir su constructor y lanzar la clase.

```
/*
 * Constructor de la clase principal
 * */
public Main()
{
 lcEnvironement = new LabConnect();

 lcEnvironement.createWindow();
}
```

Desde este constructor se accede a la ventana de *log in* desde la que se realiza la conexión con el laboratorio. En el **Diagrama 1.2** se detalla el flujo de la operación de conexión.

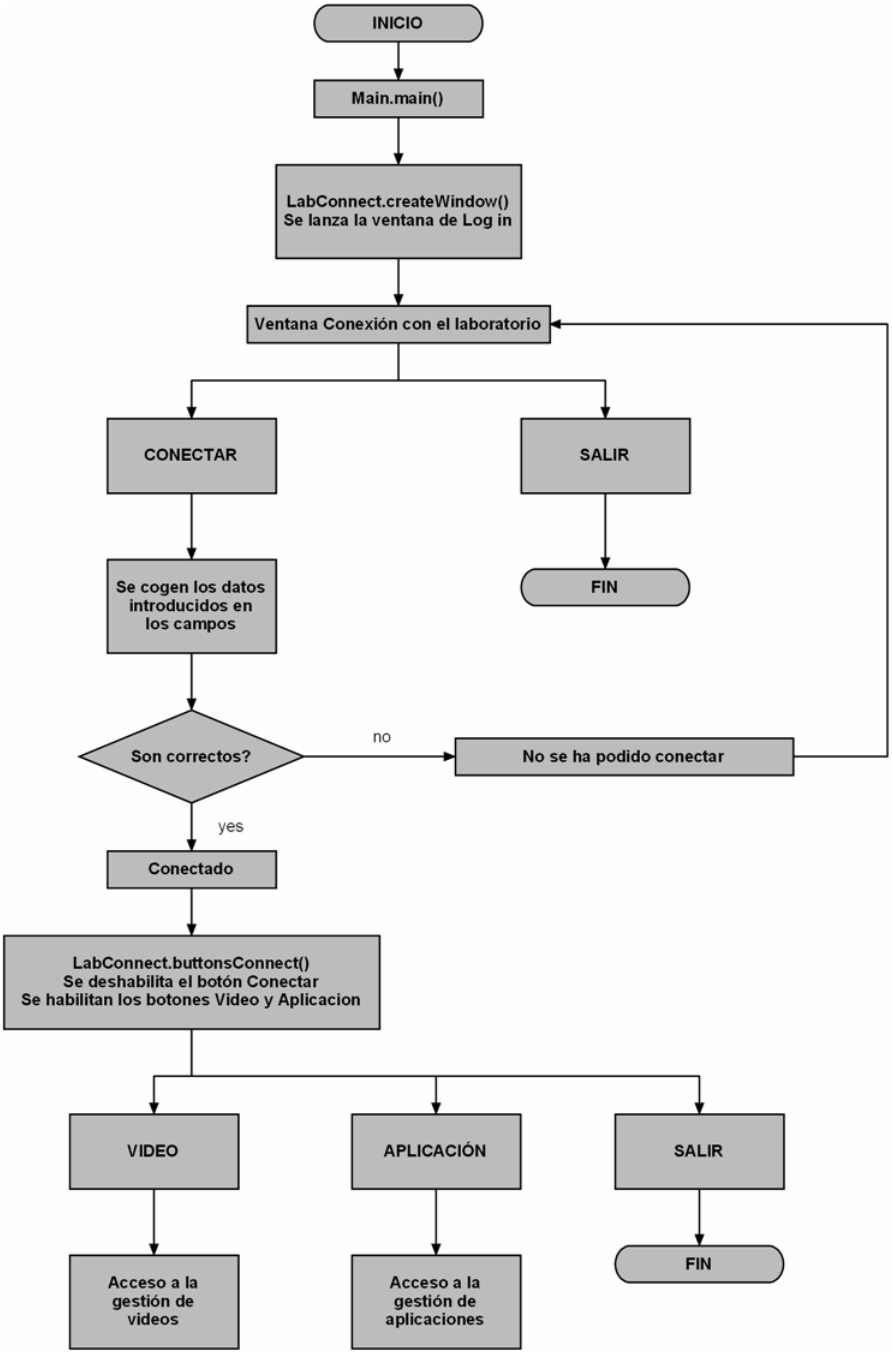


Diagrama 1.2.- *Diagrama de Flujo de la conexión con el laboratorio*

Figura 1.1.- *Pantalla de inicio antes de la conexión*

Una vez en la ventana de conexión con el laboratorio (**Figura 1.1**) se introducen los datos en los campos de texto. Después se presiona el botón Conectar, tras esta acción se comprueba que ningún campo está vacío. Si esto es así se llama al método conectar pasando como parámetros los datos que se toman de los campos. Si todo es correcto la conexión se realizará con éxito.

Para realizar la conexión con el laboratorio se utiliza el método ***connect()*** que proporciona la *API* del iMux:

```
imcConn = IMuxConnection.connect(txDirLab, txUser, txPass);
```

Los parámetros que requiere este método son la dirección IP del servidor donde esta alojado el laboratorio, el usuario y la contraseña.

```
if (imcConn != null)
{
      buttonsConnect();
      jlInfo.setText("La conexion se ha realizado con exito");
}
else
{
      jlInfo.setText("No se ha podido realizar la conexion.");
}
```

Una vez conectados con el laboratorio se habilitan los botones que dan acceso a la gestión de vídeos y de aplicaciones. En este punto se tienen dos opciones que son totalmente independientes y que se pueden manejar simultáneamente, la gestión de vídeos y la gestión de aplicaciones.

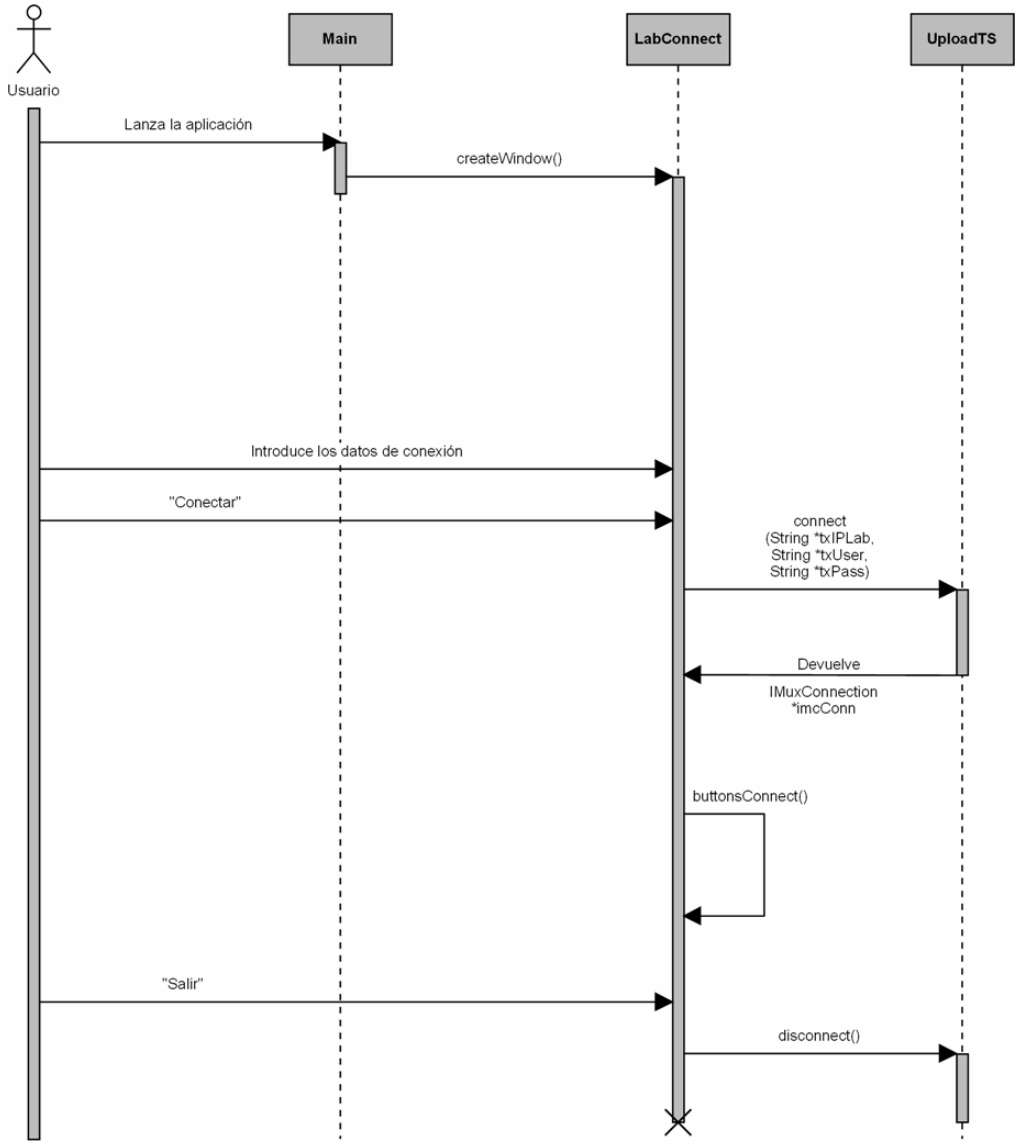


Diagrama 1.3.- *Diagrama de Secuencia de la conexión con el laboratorio*

En el **Diagrama 1.3** se puede ver la secuenciación de las acciones necesarias para realizar la conexión con el laboratorio.

Tras la conexión la ventana de inicio queda de la siguiente manera:

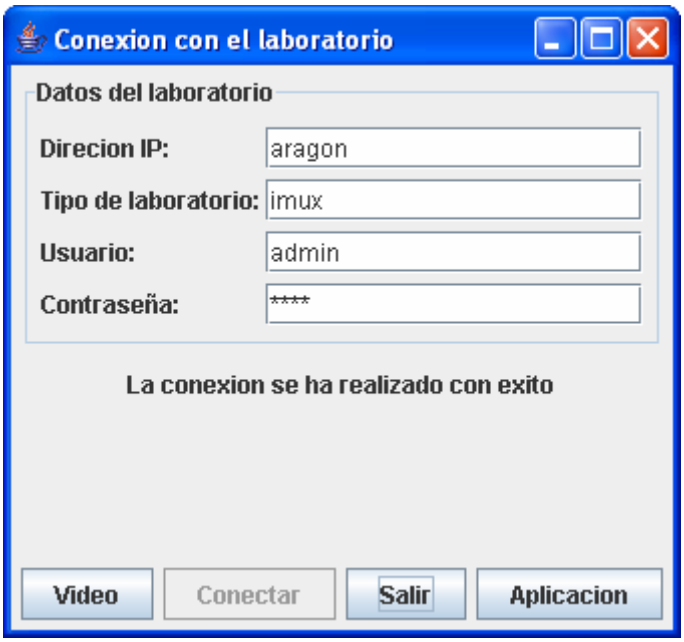


Figura 1.2.- *Pantalla de inicio tras la conexión*

Como se puede ver en la **Figura 1.2**, se ha deshabilitado el botón Conectar y se han habilitado los dos botones que dan acceso a la gestión de vídeos y de aplicaciones (Vídeo y Aplicación).

A continuación se pasan a describir los distintos bloques que componen la aplicación:

Gestión de vídeos

A través del botón Vídeo, se accede a la Gestión de vídeos. Este apartado a su vez se puede dividir en dos partes, por un lado se tiene la gestión de vídeos disponibles, es decir, la lista de vídeos que hay cargados en el laboratorio. Para poder añadir los vídeos a la escaleta, previamente se han debido cargar en el laboratorio, de modo que se tiene una lista de los vídeos cargados a la que se le pueden añadir vídeos nuevos o borrar los que ya no sean necesarios. Para estas funciones se dispone de una ventana de gestión independiente de la ventana de configuración de la escaleta. Esta ventana de configuración es la segunda parte del bloque de Gestión de vídeos, desde ella se crea la escaleta que se quiere reproducir. Desde esta ventana se tiene acceso a las distintas funciones que son necesarias para una correcta gestión de la escaleta. Por ejemplo, al añadir un nuevo vídeo a la escaleta, éste se coloca detrás del último vídeo, pero es posible que se quiera cambiar su posición, para ello se han implementado dos botones que posibilitan desplazar los vídeos

arriba y abajo una vez que han sido añadidos a la escaleta. Esta interfaz también ofrece la posibilidad de modificar la hora de inicio de los vídeos una vez que la escaleta está en reproducción. Es decir, cuando se pone la escaleta en reproducción, se calculan las horas de inicio y fin de cada vídeo a partir de la hora de inicio de la reproducción y de la duración de cada vídeo, pero es posible que interese cambiar la hora de inicio calculada por la aplicación.

Otra característica importante de este bloque es el uso de distintos hilos para implementar la concurrencia en el programa. A veces es necesario ejecutar tareas secundarias dentro de una aplicación que no interfieran con las tareas principales. Java da la oportunidad de realizar estas tareas de manera independiente. Esta aplicación corre sobre un hilo principal al cual se le pueden integrar más hilos que corren detrás de la aplicación.

La Máquina Virtual Java (JVM) es un sistema *multi-thread*, es decir, es capaz de ejecutar varias secuencias de ejecución (programas) simultáneamente. La JVM gestiona todos los detalles, asignación de tiempos de ejecución, prioridades, etc.… de forma similar a como un Sistema Operativo gestiona múltiples procesos. La diferencia básica entre un proceso de Sistema Operativo y un *thread* Java es que los *threads* corren dentro de la JVM, que es un proceso del Sistema Operativo y por tanto comparten todos los recursos, incluida la memoria y las variables y objetos allí definidos. A este tipo de procesos donde se comparten los recursos se les llama a veces procesos ligeros (*lightweight process*). [ECK 03] [BLO 01]

En el **Diagrama 1.4** se representa el ciclo de vida del un *thread* Java, este diagrama ilustra los distintos estados que puede tener un *thread* Java en cualquier momento de su vida. También ilustra las llamadas a métodos que provocan las transiciones de un estado a otro. Este no es un diagrama de estado finito pero da una idea general de las facetas más interesantes y comunes en la vida de un *thread*.

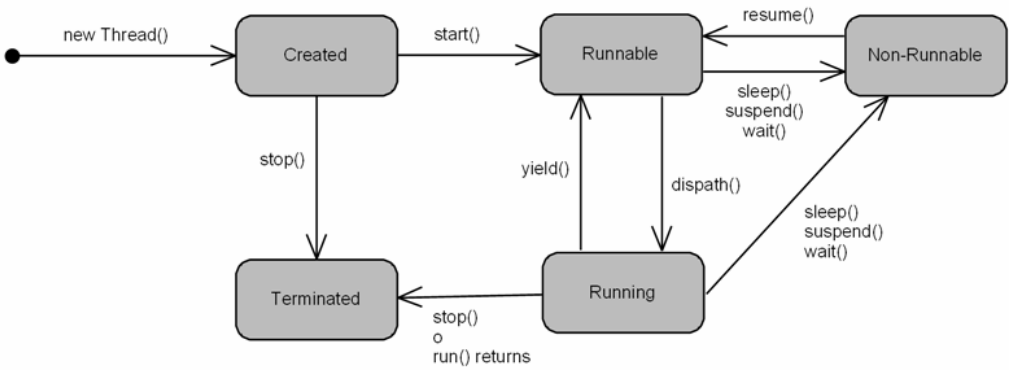


Diagrama 1.4.- *Ciclo de vida de un thread*

- Estado ***Created***: Cuando se crea un nuevo *thread* este no es arrancado, por lo tanto se queda en el estado ***Created***. Cuando un *thread* está en este estado, es sólo un objeto t*hread* vacío, no se le han asignado todavía recursos del sistema. Así, cuando un *thread* está en este estado, lo único que se puede hacer es arrancarlo o pararlo.

- Estado ***Runnable***: Una vez que el *thread* ha sido creado, para arrancarlo se llama al método ***start()***. Cuando el método ***start()*** crea los recursos del sistema necesarios para ejecutar el *thread*, programa el *thread* para ejecutarse y llama al método ***run()*** del *thread*, en este punto el *thread* está en estado ***Runnable***. Este estado se llama ***Runnable*** y no ***Running*** porque el *thread* todavía no ha empezado a ejecutarse cuando está en este estado.

- Estado ***Non-Runnable***: Un *thread* entra en estado ***Non-Runnable*** cuando ocurre uno de estos cuatro eventos:

    - Alguien llama a su método ***sleep()***
    - Alguien llama a su método ***suspend()***
    - El *thread* utiliza su método ***wait()*** para esperar una condición variable.
    - El *thread* está bloqueado durante la I/O

  Para cada entrada en este estado existe una ruta de escape distinta y específica que devuelve el *thread* al estado ***Runnable***. Una ruta de escape sólo trabaja para su entrada correspondiente. Por ejemplo, si un *thread* ha sido puesto a dormir durante un cierto número de milisegundos deben pasar esos milisegundos antes de volverse ejecutable de nuevo, por tanto llamar al método ***resume()*** en un *thread* dormido no tiene efecto.

  A continuación se detalla en la siguiente lista la ruta de escape para cada entrada en el estado ***Non-Runnable***:

    - Si se ha puesto a dormir un *thread*, deben pasar el número de milisegundos especificados
    - Si se ha suspendido un *thread*, alguien debe llamar a su método ***resume()***
    - Si un *thread* está esperando una condición variable, siempre que el objeto propietario de la variable renuncie mediante ***notify()*** o ***notifyAll()***
    - Si un *thread* está bloqueado durante la I/O, cuando se complete la I/O

- Estado ***Terminated***: Un *thread* puede morir de dos formas, por causas naturales o siendo forzado a terminar. Una muerte natural se produce cuando su método ***run()*** termina normalmente. También se puede matar un *thread* en cualquier momento llamando a su método ***stop()***.

Hay muchos usos posibles para la multitarea, pero en general, se tendrá una parte de un programa en espera por un evento en particular o recurso, y no se quiere bloquear el resto del programa a causa de esto. Así es que se crea un hilo asociado con ese evento o recurso y se deja correr independientemente del programa principal. En este bloque se trabaja con vídeos, que por lo general son ficheros muy pesados, por lo que las operaciones que se realizan con este tipo de archivos suelen ser muy costosas, tanto en tiempo como en el uso de los recursos. Por ese motivo se decide implementar varios hilos de ejecución que permitan al usuario seguir utilizando el programa principal mientras estás operaciones se completan. En concreto, se han implementado tres hilos distintos:

- **ThreadUploadInitialData**: El primero de los tres hilos que se han implementado en este bloque se ejecuta nada más entrar en la parte Gestión de vídeos de la aplicación. Cuando el usuario presiona el botón Vídeo de la ventana de inicio, la aplicación tiene que cargar los datos de todos los vídeos que están cargados en ese momento en el laboratorio para confeccionar la lista de vídeos disponibles con todos los parámetros necesarios para que estos vídeos puedan ser manejados por la aplicación.

  Este primer hilo se encarga de obtener los datos de los vídeos que se encuentran cargados en el laboratorio en el momento en que se inicia la aplicación. En esta parte, se encontró un problema añadido por las limitaciones del laboratorio. Uno de los parámetros que se necesita conocer es el tamaño/duración de los vídeos que va a manejar la aplicación. En el caso de los vídeos que ya se encuentran cargados en el laboratorio, la *API* no proporciona la posibilidad de obtener este dato de los vídeos que se encuentran ya cargados. De modo que se implementa esta opción de una manera muy manual, para ello es necesario acceder al servidor donde se encuentra alojado el laboratorio mediante comandos de Shell, una vez en el servidor se debe buscar el fichero correspondiente y calcular su tamaño. Este proceso requiere de cierto tiempo, y este tiempo será mayor cuantos más vídeos haya que consultar.

- **ThreadUploadTS**: Como ya se ha comentado, en este bloque se tiene una lista de vídeos disponibles para ser programados. Para gestionar esta lista es posible añadir o quitar vídeos. Como los ficheros de los vídeos son bastantes pesados, la acción de cargar un nuevo vídeo en el laboratorio puede tardar varios segundos, por ese motivo se ha implementado un nuevo hilo que se encargue de realizar esta operación. Este hilo se ejecuta cada vez que se quiere añadir un nuevo vídeo al laboratorio. Esta operación se asemeja a copiar un fichero de un dispositivo a otro, y, como es lógico, esta operación durará más cuanto mayor sea el tamaño del fichero.

- **ThreadPlay**: Este tercer hilo se encarga de controlar la reproducción de los vídeos. El motivo por el cual se ha implementado este hilo es muy diferente a los motivos de los 2 hilos anteriores. Este *thread* se encarga de la reproducción de los vídeos, y por lo tanto, va a estar corriendo durante prácticamente todo el tiempo de vida de la aplicación. La existencia de este hilo es necesaria para que la aplicación tenga sentido, ya que es absurdo pensar que mientras se están reproduciendo los vídeos se bloquee el resto del programa. Este hilo va a ir reproduciendo un vídeo tras otro independientemente de lo que se esté haciendo en la aplicación principal. Sólo se morirá si el usuario pulsa el botón Stop de la ventana de configuración de la escaleta, o si se terminan los vídeos de la escaleta.

Gestión de aplicaciones

Este bloque se encarga de las aplicaciones MHP del canal. Al igual que ocurría en el bloque anterior, para poder programar estas aplicaciones primero deben ser cargadas en el laboratorio. Para ello se implementan dos funcionalidades, Añadir y Borrar aplicaciones del laboratorio. Una vez que se tienen las aplicaciones guardadas en el laboratorio, se puede configurar la escaleta de aplicación. Se debe tener en cuenta la diferencia del concepto escaleta dependiendo de si se está hablando de vídeos o de aplicaciones. En el primer caso, para los vídeos, la escaleta supone una lista de reproducción continua en la que sólo puede reproducirse un vídeo a la vez. En cambio, cuando se habla de aplicaciones, la escaleta no es más que una tabla que refleja las distintas programaciones que se han hecho con las aplicaciones, estas programaciones indican los momentos en que una determinada aplicación es accesible por el usuario, ya que puede interesar que una aplicación sólo sea visible en determinados momentos. Sobra decir que en este caso si es posible que dos aplicaciones estén disponibles al mismo tiempo, y que no se trata de una

reproducción continua, es decir, puede haber momentos en que no haya ninguna aplicación disponible, y otros en los que se tengan 4 ó 5 aplicaciones activas.

Generación de Stream Events

Por último se tiene la parte referente a los *Stream Events*. En este bloque se describe cómo se ha implementado una sencilla aplicación MHP sincronizada mediante el uso de *Stream Events*. Esta aplicación se encarga de mostrar al inicio de cada vídeo el nombre del mismo en un *banner* en la parte inferior de la pantalla. Para ello, el envío de los eventos se incluye en el código correspondiente a la gestión de vídeos ya que cada vez que se comience a reproducir un vídeo se debe enviar un evento. Estos *Stream Events* serán escuchados por la aplicación MHP que previamente ha tenido que ser cargada en el laboratorio y que debe estar disponible y corriendo en el receptor para que pueda escuchar los eventos cuando éstos lleguen al receptor.

Programación gráfica

Como se trata de una aplicación de escritorio pensada para que el usuario interactúe con ella es necesario tener una interfaz gráfica agradable e intuitiva para que su manejo sea lo más sencillo posible. Se pretende que el usuario no tenga que tener conocimientos específicos en el manejo tanto del laboratorio como de la implementación de la aplicación. Esta aplicación se compone de un sistema de ventanas que dan acceso a las distintas acciones o temas de la aplicación. Todo este sistema de ventanas está implementado en código Java, dentro del mismo proyecto que el resto de funcionalidades. Todas las clases que corresponden a partes de programación gráfica se recogen en el paquete ***com.cein.aranguren.scheduler.gui*** (ver **Diagrama 1.1**). Esta implementación se detalla más adelante en uno de los apartados de este capítulo.

Como se ha comentado más arriba, en primer lugar se muestra una ventana de inicio desde la que se realiza la conexión con el laboratorio (ver **Figura 1.1**). En el apartado de Gestión de vídeos se tienen dos ventanas principales:

- Por un lado la ventana Gestión de Vídeos que permite al usuario añadir o quitar videos de la lista de videos disponibles.
- Desde la ventana Gestión de Vídeos, también se accede a la otra ventana principal del bloque, la ventana Configuración de la escaleta de vídeos, desde la cual se gestiona la escaleta propiamente dicha. Es decir, se añaden videos a la programación, se modifica su posición de reproducción, etc.

Por último, se tiene la ventana principal del bloque correspondiente a la Gestión de aplicaciones. Desde esta pantalla, se controla tanto la lista de aplicaciones disponibles como el *Schedule* donde se programa la disponibilidad de las mismas.

La parte de generación de *Stream Events* no tiene parte gráfica, ya que se ha implementado a modo de demostración. En este caso se trata de una acción muy concreta que no requiere de la intervención del usuario, por este motivo no es necesario que tenga una parte visible. Los *Stream Events* se envían desde el código encargado de controlar la reproducción de los vídeos, ya que se debe enviar un evento cada vez que se inicie la reproducción de un video.

1.1.- Gestión de Vídeos

En la gestión de vídeos se pueden diferenciar dos partes, por un lado está la gestión de los vídeos disponibles, es decir, para poder configurar una escaleta de vídeos primero se tienen que tener estos vídeos cargados en el laboratorio. En esta primera parte se implementan funcionalidades básicas como añadir un nuevo vídeo o borrar un vídeo del laboratorio. Tan importante es cargar los vídeos, para que estén disponibles cuando se quieran incluir en la escaleta, como borrar los que ya no se necesiten para no sobrecargar el laboratorio.

Por otro lado se tiene la gestión de la escaleta propiamente dicha, en esta parte la interfaz permite configurar la escaleta deseada, es decir, se seleccionan los vídeos que se quieren reproducir y el orden en que van a ser emitidos y la aplicación se encargará de asignar las horas de inicio y fin de cada vídeo para que la reproducción sea continua.

Para la gestión de los vídeos se ha creado un objeto nuevo (***TransportStream2.java***), que se va a utilizar para guardar y consultar parámetros de los vídeos. Este objeto tiene 7 campos:

- ***String txName***: Se guarda el nombre del vídeo.
- ***double dSize***: Se guarda el tamaño del vídeo.
- ***double dBitrate***: Se guarda el *bitrate* con el que se ha codificado el vídeo.
- ***long lDuration***: Se guarda la duración del vídeo.
- ***Date dFrom***: Se guarda la hora de inicio, una vez que se ha incluido el vídeo en la escaleta
- ***Date dTo***: Se guarda la hora de fin, una vez que se ha incluido el vídeo en la escaleta
- ***String txText***: De momento no se utiliza, se creó por si alguna aplicación hecha a posteriori necesitaba el paso de alguna otra información sobre el vídeo.

En esta parte de gestión de vídeos se hace uso de dos estructuras de datos que se encargan de facilitar la reproducción de la escaleta. Estas estructuras son:

- Una cola FIFO de ***String***: En esta cola se guardan los nombres de los vídeos que componen la escaleta en orden de reproducción. La aplicación se va a alimentar de esta cola para realizar la reproducción.

- Un ***Set*** de ***TransportStream2***: Este ***Set*** se comporta como una base de datos volátil (porque se vacía cada vez que se cierra la aplicación). En él se guarda una variable ***TransportStream2*** por cada vídeo que se tiene en el laboratorio con los datos correspondientes a este vídeo.

Como ya se ha comentado antes, se accede a esta parte de gestión de vídeos presionando el botón Vídeo de la ventana de inicio.

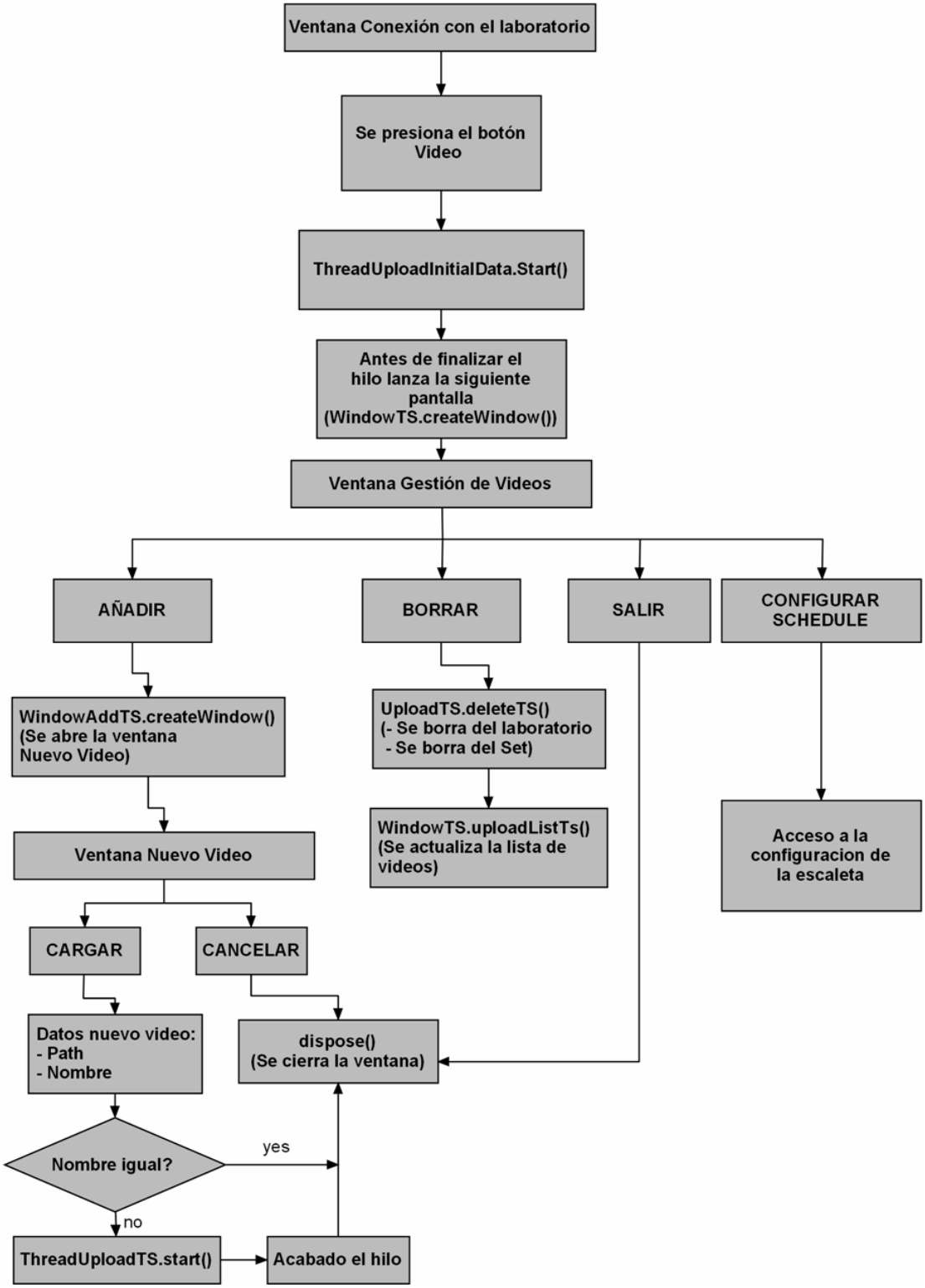


Diagrama 1.5.- *Diagrama de Flujo de la gestión de vídeos*

Como se puede ver en el **Diagrama 1.5,** en primer lugar se cargan los datos de todos los vídeos que ya están en el laboratorio. Para ello se lanza el hilo ***ThreadUploadInitialData***

```
//Se instancia un nuevo objeto Java
Thread thInitialData = new Thread( new
ThreadUploadInitialData(),"thInitialData" );

// Se arranca el hilo, para que comience su ejecución
thInitialData.start();
```

Ejecución del hilo:

Uno de los problemas que se han tenido con el laboratorio es la falta de acceso a algunos datos, por ejemplo en este caso no se podía conocer ni la duración ni el tamaño de los vídeos que ya estaban cargados en el laboratorio. Esto supone un serio problema porque se necesita conocer estos datos para poder crear la escaleta, ya que el cálculo de las horas de inicio y fin de los vídeos, que se verá más adelante, necesita de estos datos. Esta limitación en un principio obligaba a tener que cargar todos los vídeos con los que se iba a trabajar cada vez que se lanzaba la aplicación, pero esto resultaba poco práctico, además de sobrecargar el laboratorio ya que los vídeos que se cargaban en otras sesiones se iban acumulando en el laboratorio. Por este motivo se decidió implementar esta acción de obtener los datos de los vídeos que ya estaban cargados en el laboratorio.

Para la ejecución de este hilo se utiliza la librería ***JSch***, esta librería nos permite ejecutar comandos de consola desde la aplicación Java. ***JSch*** es una implementación pura de Java de SSH2. ***JSch*** permite conectar con un servidor *sshd* y utilizar redireccionamiento de puertos, transferencia de ficheros, etc., y se puede integrar su funcionalidad en programas Java. ***JSch*** se distribuye bajo una licencia BSD[2].

En el método ***run()*** de este hilo se llama al método ***getSizeTS()*** de la clase ***ObtainSizeTS*** de la aplicación.

```
ObtainSizeTS.getSizeTS();
```

[2] La licencia BSD es la licencia de software otorgada principalmente para los sistemas BSD (Berkeley Software Distribution). Es una licencia de software libre permisiva cómo la licencia de OpenSSL o la MIT License, permite la libre redistribución y la modificación.

En este método lo primero que se hace es intentar abrir una sesión con el servidor donde está alojado el laboratorio. Los parámetros necesarios (*user*, *password* y *port*) se le dan directamente por programa. Estos datos son identificativos de cada servidor, por lo que si se quiere trabajar con un laboratorio **iMux** alojado en otro servidor se deberán cambiar estos datos en el código de la aplicación. Si no se cambian estos datos la aplicación no podrá acceder al nuevo servidor porque está utilizando los identificativos erróneos.

```
Session session = jsch.getSession(user, host, 22);

session.connect();
```

Una vez se ha conectado con el servidor se leen todos los vídeos que hay guardados en el laboratorio y se guardan en un *array* para que se pueda trabajar con ellos mas tarde.

```
TStream[] arrTS;

arrTS = TStream.getAll(UploadTS.imcConn);
```

Para tomar todos los vídeos que están cargados en el laboratorio cuando se lanza por primera vez la aplicación se utiliza el método ***getAll()*** que hereda de la clase ***TStream*** y que se proporciona en la *API* del laboratorio.

Después de esto se entra en un bucle *for* donde se tratan todos los elementos del *array* de ***TStream***. Para cada vídeo se realiza el siguiente tratamiento: en primer lugar se calcula el tamaño con el comando:

```
String command = "wc -c /var/imux/admin/ts" + txId + ".trp |sed -e 's/
.*//'";
```

Donde ***txId*** es la ID del vídeo que se está tratando. Para comprender este comando se debe saber que el laboratorio guarda cada vídeo en una carpeta ubicada en la ruta que se especifica en el comando (**/var/imux/admin/**), esta carpeta se encuentra en el servidor en el cual se aloja el laboratorio. Además cada vídeo se guarda en forma de fichero **.trp** y su nomenclatura sigue una regla basada en la ID que identifica cada vídeo dentro del laboratorio. Esta ID se le asigna al vídeo en el momento en que éste es cargado en el laboratorio, y en ese mismo momento se crea el fichero **tsID.trp**, es decir, si a un vídeo se le asigna ID = 1, el archivo correspondiente a este vídeo será **ts1.trp**.

Para ejecutar este comando es necesario abrir un canal al cual se le pasa el comando.

```
Channel channel=session.openChannel("exec");
((ChannelExec)channel).setCommand(command);
channel.connect();
```

Para asegurar que los tamaños de los vídeos se calculan correctamente se ha añadido un *sleep* de 1 segundo en cada vuelta del *for*. De esta forma el tiempo que se tarda en cargar todos los vídeos que se encuentran en el laboratorio será de tantos segundos como vídeos haya.

```
/*
* Se añade un sleep de 1 segundo para asegurarnos de que los
* tamaños se calculan correctamente.
* */
try{Thread.sleep(1000);}catch(Exception ee){}
```

Una vez que se ha obtenido el tamaño del vídeo se calcula la duración del mismo dividiendo el tamaño entre el *bitrate* que se obtiene de la variable ***TStream*** que se está tratando. Todos estos datos junto con el nombre del vídeo se guardan en una variable ***TransportStream2*** que se añade al ***Set***.

```
TransportStream2 ts2Add = new TransportStream2();
ts2Add.setDSize(lFsize);

tsTs = arrTS[j];
String txNameTS = tsTs.getName();
ts2Add.setTxName(txNameTS);
double dBitrate = tsTs.getBitrate();

ts2Add.setDBitrate(dBitrate);

double dDuration = lFsize/dBitrate;
long lDurationMili = (long) (dDuration * 1000);
//Se guarda la duracion en milisegundos porque esta se puede sumar
//directamente
ts2Add.setLDuration(lDurationMili);

/*
 * Se guarda el ts en el hashset
 * */
UploadTS.hsTS.add(ts2Add);
```

Para calcular la duración de los vídeos se cambia el formato de los datos a ***double*** de esta forma se evita que se redondeen los decimales y así se gana precisión en el cálculo de la duración.

Cuando se han tratado todos lo vídeos y se ha salido del *for* se desconecta la sesión y se sale del método ***getSizeTS()***.

Por último se lanza la ventana Gestión de Vídeos desde la que se maneja la lista de vídeos disponibles. Y con esto se termina la ejecución del hilo ***ThreadUploadInitialData***.

En el **Diagrama 1.6** se ha representado la secuenciación de las acciones necesarias para acceder al bloque Gestión de vídeos.

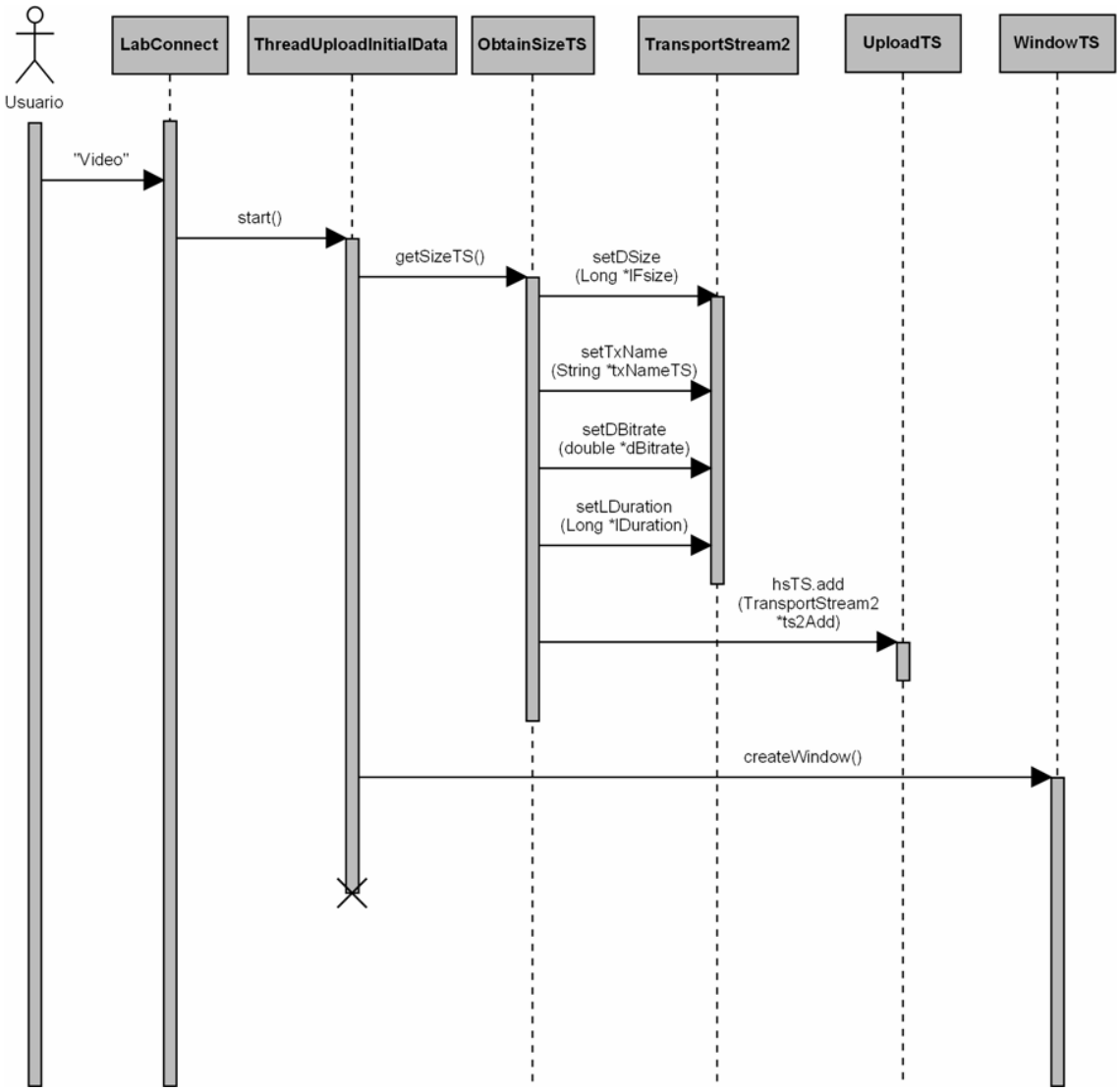


Diagrama 1.6.- *Diagrama de secuencia de acceso al bloque Gestión de Vídeos*

Desde la ventana Gestión de Vídeos (**Figura 1.3**) se tienen 4 acciones disponibles:

- Añadir un nuevo vídeo al laboratorio
- Borrar un vídeo del laboratorio
- Salir de la ventana
- Acceder a la configuración de la escaleta

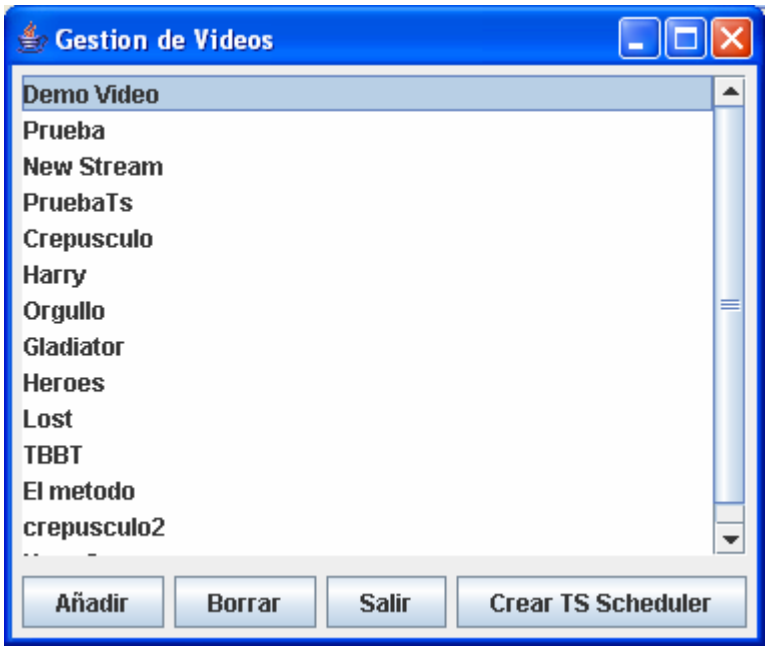


Figura 1.3.- *Ventana Gestión de Vídeos*

1.1.1.- Añadir un nuevo vídeo

El primer paso para crear una escaleta de reproducción es tener vídeos que reproducir. Por ello es necesario añadir los vídeos al laboratorio para que estén disponibles cuando se quieran incluir en la escaleta.

Cuando se presiona el botón Añadir de la ventana de gestión de vídeos se lanza la ventana de añadir un nuevo vídeo (**Figura 1.4**).

```
WindowAddTS wats = new WindowAddTS();
wats.createWindow();
```

Esta ventana tiene 3 campos (*Path* del archivo, Nombre del vídeo y tamaño del archivo), desde aquí se tiene acceso a dos acciones: Cargar el vídeo o salir de la ventana.

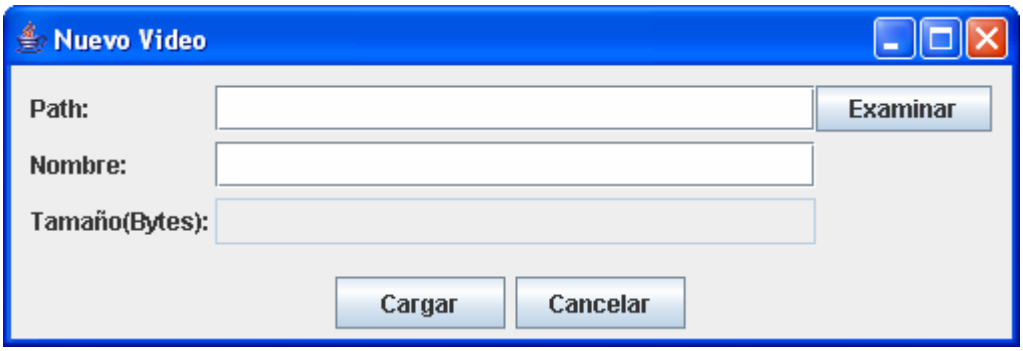


Figura 1.4.- *Ventana Nuevo Vídeo*

Si se presiona el botón Cargar se lanza el hilo ***ThreadUploadTS***, que se encarga de subir el vídeo al laboratorio.

Ejecución del hilo:

Se toma la lista de todos los vídeos cargados en el laboratorio, para ello se llama al método ***TStreamObtention()*** de la clase ***UploadTS*** de la aplicación.

```
arrTxTs = UploadTS.TStreamObtention();
```

Este método devuelve un *array* de ***String*** con los nombres de todos los vídeos que están guardados en el laboratorio. Con esta lista se comprueba que no hay ningún vídeo con el mismo nombre que el que se quiere cargar.

```
if (arrTxTs.length != 0)
{
      while (i<arrTxTs.length &&
            !arrTxTs[i].equals(WindowAddTS.txNameFile))
      {
            i++;
      }
}
```

Si no existe se crea una nueva variable ***TransportStream2*** con los datos del nuevo vídeo que se va a subir (Nombre, Tamaño, *Bitrate* y Duración). Para ello se llama al método ***addTS()*** de la clase ***UploadTS***.

```
boolean boUploadTS;
UploadTS uts = new UploadTS();
boUploadTS = uts.addTS();
```

En este método se carga el vídeo en el laboratorio y se guarda la variable en el ***Set***. Este método devuelve un valor ***booleano*** que indica si el vídeo se ha cargado satisfactoriamente. Una vez que se comprueba que el vídeo se ha cargado correctamente se refresca la lista de vídeos disponibles con el método ***uploadListTs()*** de la clase ***WindowTS***.

```
if (boUploadTS)
{
      WindowTS wts = new WindowTS();
      wts.uploadListTs();
}
```

Por último, se cierra la ventana y se vuelve a la pantalla anterior.

Un resumen de la secuenciación de esta operación se puede ver en el **Diagrama 1.7** ue representa el diagrama de secuencia de la acción: Añadir un nuevo vídeo.

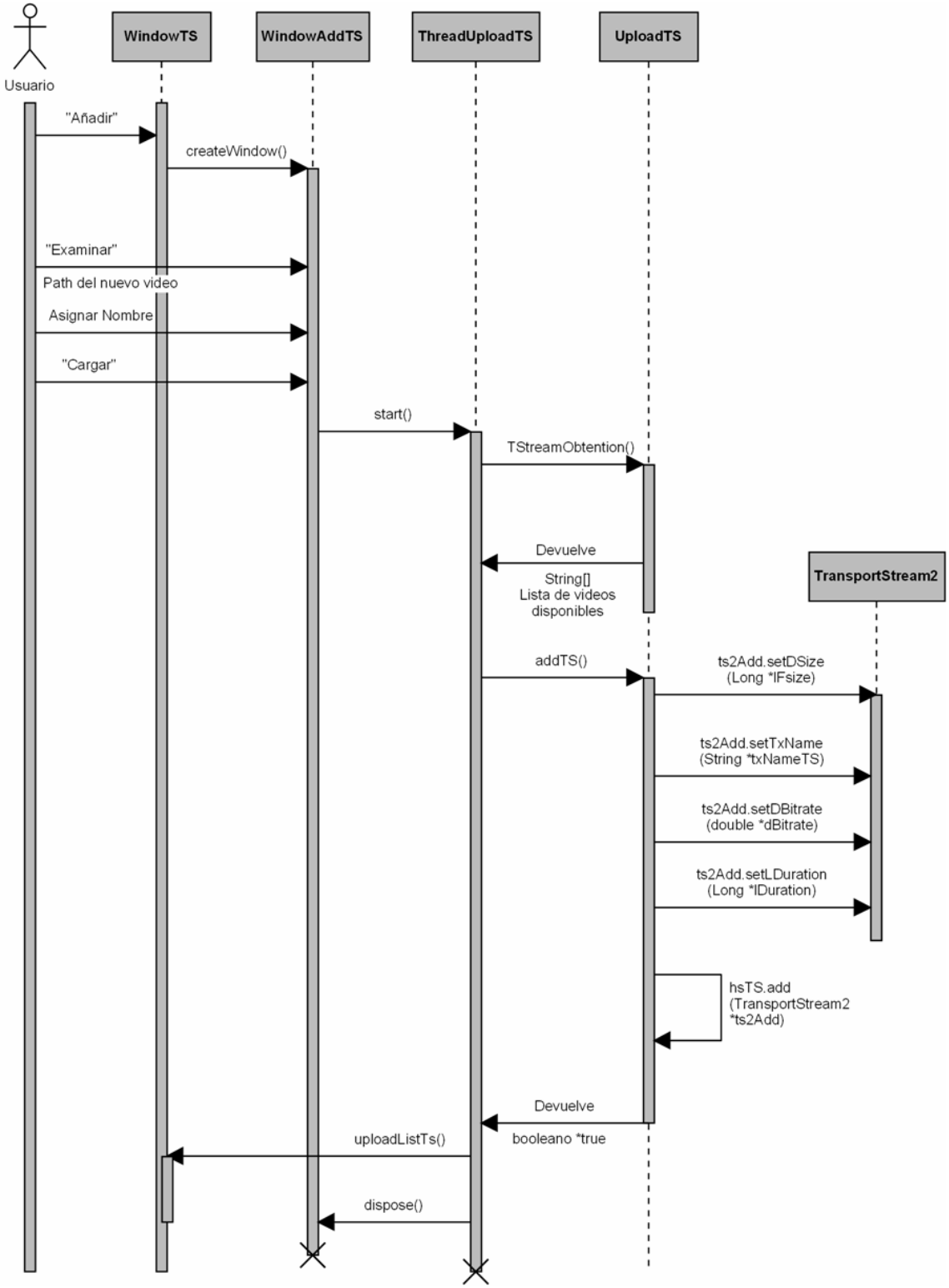


Diagrama 1.7.- *Diagrama de secuencia de la acción añadir un nuevo vídeo*

A continuación se describen los diferentes métodos que intervienen en la ejecución de esta funcionalidad. Estos métodos han sido mencionados en la descripción de la acción Añadir vídeo, pero ahora se va a describir con más detalle cómo se ha implementado cada uno de ellos.

Método: ***String[] TStreamObtention()***

- *Parámetros de entrada:* Ninguno.
- *Salida:* ***String[]*** → *Array* de ***String*** con los nombres de los vídeos cargados en el laboratorio

En primer lugar, se obtienen todos los vídeos guardados en el laboratorio con el método ***getAll()*** de la clase ***TStream*** proporcionada por la *API* del laboratorio. Ese método devuelve un *array* de ***TStream***, una clase que se define también en la *API* del laboratorio.

Se crea un *array* de ***String*** del mismo tamaño que el *array* que devuelve el método ***getAll()***

```
arrTS = TStream.getAll(imcConn);
String[] arrTxTs = new String[arrTS.length];
```

Después, mediante un bucle *for* se recorre el *array* de ***TStream*** y se copian los nombres en el *array* de ***String***.

```
for (int i=0; i<arrTS.length; i++)
{
      arrTxTs[i] = arrTS[i].getName();
}
return arrTxTs;
```

Método: ***booleano addTS()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***booleano*** → Indicador de si el fichero se ha cargado correctamente

Se crea un objeto de tipo ***File*** con el *Path* que se ha introducido en el campo del mismo nombre de la ventana "Nuevo Vídeo". También se crea una nueva variable ***TranportStream2***, en ella se guarda el tamaño del nuevo fichero, este dato se consulta en el nuevo objeto ***File*** creado, como el dato a la salida de la consulta se da en *Bytes* se transforma en *bits* antes de guardarlo en la variable ***TransportStream2***.

```
File fTs = new File(WindowAddTS.txPathFile);

TransportStream2 ts2Add = new TransportStream2();

Long lFsize = new Long(fTs.length());
//Como se obtiene el tamaño en Bytes se pasa a bits
lFsize = lFsize * 8;
//Se guarda el tamaño del archivo en la variable ts2 que
//despues se guardara en el hashset
ts2Add.setDSize(lFsize);
```

A continuación se crea una variable ***TStreamPackets*** (objeto definido en la *API* del laboratorio), para cargar el nuevo vídeo en el laboratorio. De esta nueva variable se obtiene el *bitrate* del vídeo que se está cargando y este dato se guarda en la variable ***TransportStream2***.

```
TStreamPackets tspPack;

[…]

String txNameTs = WindowAddTS.txNameFile;
//Se guarda el nombre del TS en la variable ts2
ts2Add.setTxName(txNameTs);

tspPack = TStream.createTStreamPackets(imcConn, txNameTs);

/*
 * Se carga el ts en el laboratorio
 * */
tspPack.update(fTs, true);

tspPack.setBitrate(tspPack.getDetectedBitrate());
```

Por último, se calcula la duración del vídeo del mismo modo que se ha descrito en el ***ThreadUploadInitialData*** y se guarda la variable ***TransportStream2*** en el ***Set*** con todos los datos obtenidos.

Al cargar los vídeos de esta forma en el laboratorio, en muchos casos se produce un error cuando se intenta activar la configuración. Este error se debe a que, al cargar el vídeo se conservan la PIDs originales del vídeo, a menudo una de estas PIDs es 0 lo cual da problemas porque el laboratorio no admite como válido este valor para una PID. Para solucionar este problema se borran todas las PIDs originales excepto las que pertenecen a los paquetes de audio y vídeo. De esta forma, antes de salir de este método, se borran todas las PIDs que no interesan y se cambia el tipo de PID a "*unreferenced*".

```
TStreamPacketsPID[] arrTsp = tspPack.getTStreamPacketsPIDs();
if (arrTsp != null)
{
      for (int i=0; i<arrTsp.length; i++)
      {
            int iPid = arrTsp[i].getPidType();

            if (iPid != TStreamPacketsPID.PID_TYPE_AUDIO_MPEG1 &&
                iPid != TStreamPacketsPID.PID_TYPE_AUDIO_MPEG2)
            {
                 if (iPid !=
                     TStreamPacketsPID.PID_TYPE_VIDEO_MPEG1 &&
                     iPid !=
                     TStreamPacketsPID.PID_TYPE_VIDEO_MPEG2)
                 {
                         arrTsp[i].setDestinationPID(-1);

                         arrTsp[i].setPidType(
                                TStreamPacketsPID.PID_TYPE_UNREFERENCED);
                 }
            }
      }
}
```

Por último, si todo se ha ejecutado correctamente se le da a la variable ***boUploadTS*** el valor ***true*** que se devuelve como parámetro de salida para indicar que la carga del archivo de vídeo se ha completado con éxito. En caso contrario, se devolverá en esta variable el valor ***false***.

```
boolean boUploadTS;
[…]
boUploadTS = true;
```

Método: ***void uploadListTs()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

Primero, se recupera en un *array* de ***String*** la lista de todos los vídeos disponibles en el laboratorio con el método ***TStreamObtention()***.

```
arrTxTs = UploadTS.TStreamObtention();
```

Después se limpian las listas antiguas, tanto de la ventana Gestión de Vídeos como de la ventana Configuración de la escaleta de vídeos (de la que se hablará más adelante) ya que si las dos ventanas están abiertas se actualizan ambas listas de vídeos disponibles.

```
if (dlmTs.size() != 0)
{
      dlmTs.clear();
}
if(WindowCreateSchduledTS.dlmListaTS.size() != 0)
{
      WindowCreateSchduledTS.dlmListaTS.clear();
}
```

Tras limpiar las listas se comprueba que el *array* obtenido con los nombres de los vídeos cargados en el laboratorio no esté vacío. Si es así, se rellenan las listas con los nuevos datos a través de un bucle *for*.

```
if (arrTxTs != null)
{
      for (int i=0; i<arrTxTs.length; i++)
      {
            dlmTs.addElement(arrTxTs[i]);
            WindowCreateSchduledTS.dlmListaTS.addElement(arrTxTs[i]);
      }
}
```

1.1.2.- Borrar un vídeo

Al presionar el botón Borrar de la ventana Añadir Vídeo se crea una variable ***String*** donde se guarda el nombre del vídeo que se ha seleccionado para ser borrado.

```
      String txDelTS;
      txDelTS = jlTs.getSelectedValue().toString();
```

Esta variable se utiliza como parámetro de entrada en la llamada al método ***deleteTS()*** de la clase ***UploadTS***. Este método borra el vídeo seleccionado tanto del laboratorio como del ***Set***. Al igual que en el método ***addTS()***, este método devuelve un valor booleano que indica si el vídeo se ha borrado correctamente.

Si el método ***deleteTS()*** devuelve ***true*** como parámetro de salida se refresca la lista de vídeos disponibles con el método ***uploadListTs()***. Si el valor devuelto es ***false***, el vídeo no se ha podido borrar correctamente y la lista de vídeos disponibles se queda como estaba.

Para una mejor comprensión de esta operación se puede ir al **Diagrama 1.8**, que representa el diagrama de secuencia de esta operación.

```
boolean boDeleteTS;
boDeleteTS = uts.deleteTS(txDelTS);
if (boDeleteTS)
{
 uploadListTs();
}
```

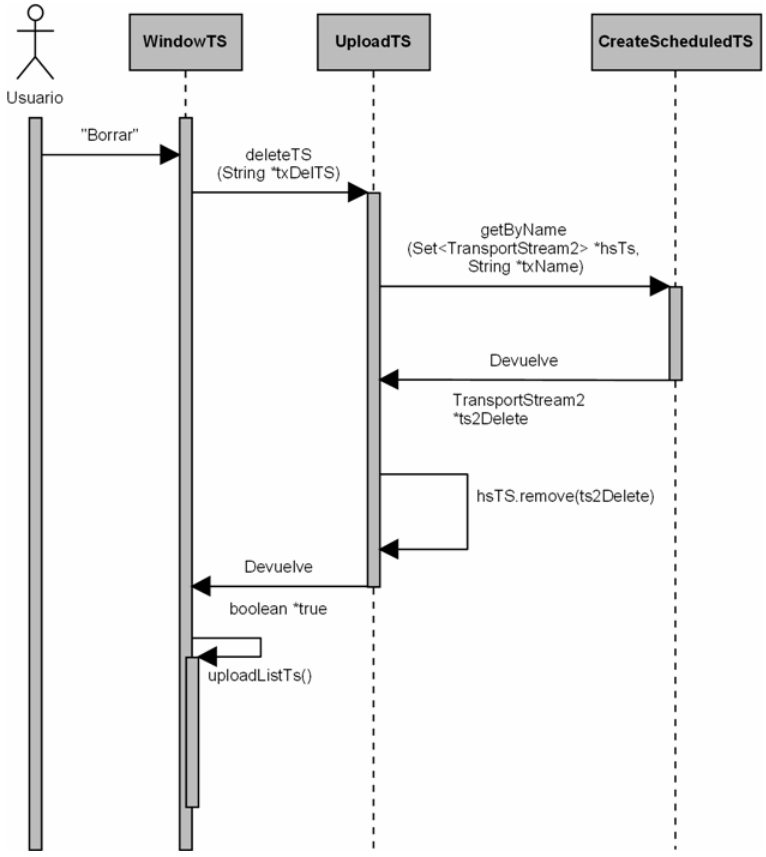


Diagrama 1.8.- *Diagrama de secuencia de la acción Borrar un vídeo*

Método: ***booleano deleteTS(String *txDelTS)***

- *Parámetros de entrada:*
  String *txDelTS → Nombre del vídeo a borrar
- *Salida:* ***booleano*** → Indicador de si el vídeo se ha borrado correctamente

Este método se divide en dos acciones, en primer lugar se borra el vídeo del laboratorio y después de borra del ***Set***.

Se recupera el vídeo que se quiere borrar en una variable ***TStream*** con el método ***getByName()*** de la clase ***TStream*** proporcionada por la *API* del laboratorio.

```
TStream[] arrTsDelTS;
arrTsDelTS = TStream.getByName(imcConn, txDelTS);
```

Se comprueba que realmente existe dicho vídeo y se borra del laboratorio con el método ***delete()*** de la clase ***TStream***.

```
TStream tsDelTstream = null;
if(arrTsDelTS.length == 0)
{
	System.out.println("No se ha encontrado ningun TS con ese
		nombre.");
}
if(arrTsDelTS.length > 1)
{
	System.out.println("Advertencia: Hay mas de un TS con ese
		nombre.");
	tsDelTstream = arrTsDelTS[0];
}
if(arrTsDelTS.length == 1)
{
	tsDelTstream = arrTsDelTS[0];
}
if(tsDelTstream != null)
{
	tsDelTstream.delete();
	TransportStream2 ts2Delete =
		CreateScheduledTS.getByName(hsTS, txDelTS);
	UploadTS.hsTS.remove(ts2Delete);
	boDeleteTS = true;
}
```

Por otro lado, se recupera el vídeo que se quiere borrar en una variable ***TransportStream2*** con el método ***getByName()*** de la clase ***CreateScheduledTS***. A partir de esta variable se borra el vídeo del ***Set*** con el método ***remove()***.

Método: ***TransportStream2 getByName(Set<TransportStream2> *hsTs2, String *txName)***

- *Parámetros de entrada:*
  Set<TransportStream2> *hsTs2 → ***Set*** en el que se está buscando.
  String *txName → Nombre del objeto que se está buscando
- *Salida:* ***TransportStream2*** → Objeto que se está buscando.

En primer lugar se crea un objeto ***Iterator*** que se utiliza para recorrer el ***Set*** en busca del objeto que se quiere localizar.

```
Iterator<TransportStream2> iteratorTS2 = hsTs2.iterator();
```

Después se recorre, con ayuda de un bucle ***while***, el ***Set***, y se compara en cada vuelta del bucle si el elemento que se está tratando es el que se está buscando. Si es así, se guarda ese elemento en una variable ***TransportStream2*** que se retorna a la salida del método.

```
TransportStream2 ts2 = null;

while (iteratorTS2.hasNext())
{
	TransportStream2 ts2Aux = iteratorTS2.next();
	if(ts2Aux.getTxName().equals(txName))
	{
		ts2 = ts2Aux;
	}
}
return ts2;
```

1.1.3.- Configurar la escaleta de vídeos

Se accede a la pantalla Configuración de la escaleta de vídeos (**Figura 1.5**) desde la que se va a gestionar la escaleta de vídeos.

```
WindowCreateSchduledTS wcsts = new WindowCreateSchduledTS();
wcsts.createWindow();
```

En el **Diagrama 1.9** se pueden ver las 8 acciones posibles que se pueden realizar desde esta ventana.

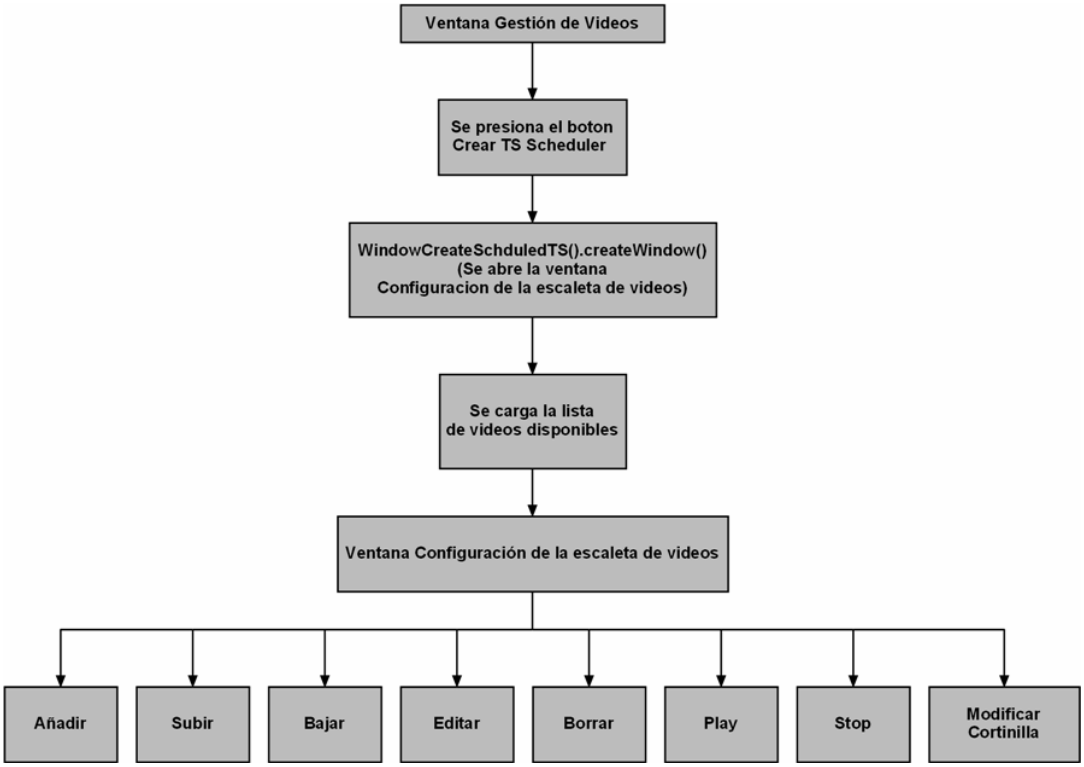


Diagrama 1.9.- *Diagrama de flujo de la configuración de la escaleta de vídeos*

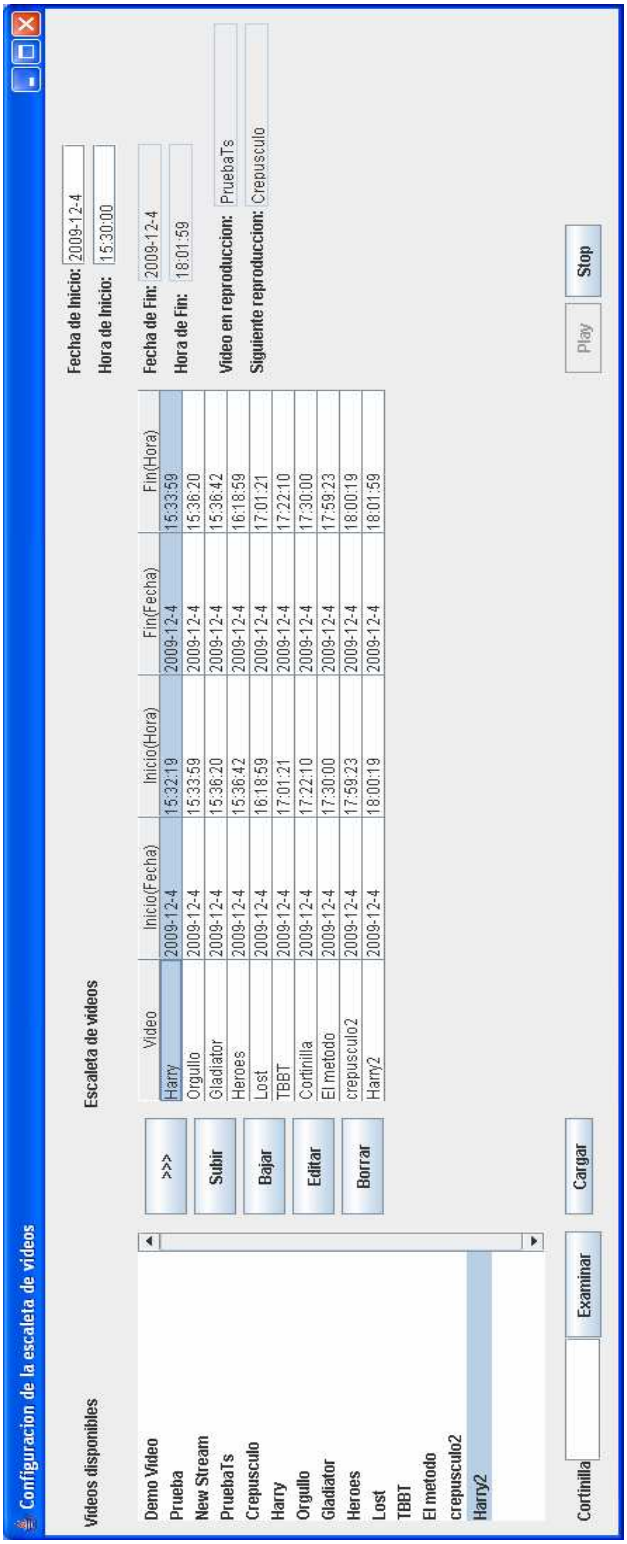


Figura 1.5.- Ventana Configuración de la escaleta de vídeo

Desde la ventana mostrada en la **Figura 1.5** se tiene acceso a las siguientes acciones:

- Añadir un nuevo vídeo a la escaleta.
- Subir una posición el vídeo en la escaleta.
- Bajar una posición el vídeo en la escaleta.
- Editar la hora de inicio de un vídeo de la escaleta.
- Borrar un vídeo de la escaleta.
- Play (Iniciar la reproducción de la escaleta).
- Stop (Detener la reproducción en curso).
- Modificar el vídeo de la cortinilla.

Como se puede ver en la **Figura 1.5**, en esta ventana se tiene, por un lado, una lista con todos los vídeos de que se dispone para configurar la escaleta. Como también se ha comentado antes, esta lista de vídeos se ira actualizando a medida que se vayan borrando o añadiendo vídeos desde la ventana anterior. Por otro lado, en la parte central de la ventana, se tiene una tabla dinámica donde se refleja la escaleta que se esta gestionando, tanto durante la reproducción como antes de ella.

1.1.4.- Añadir un nuevo vídeo a la escaleta

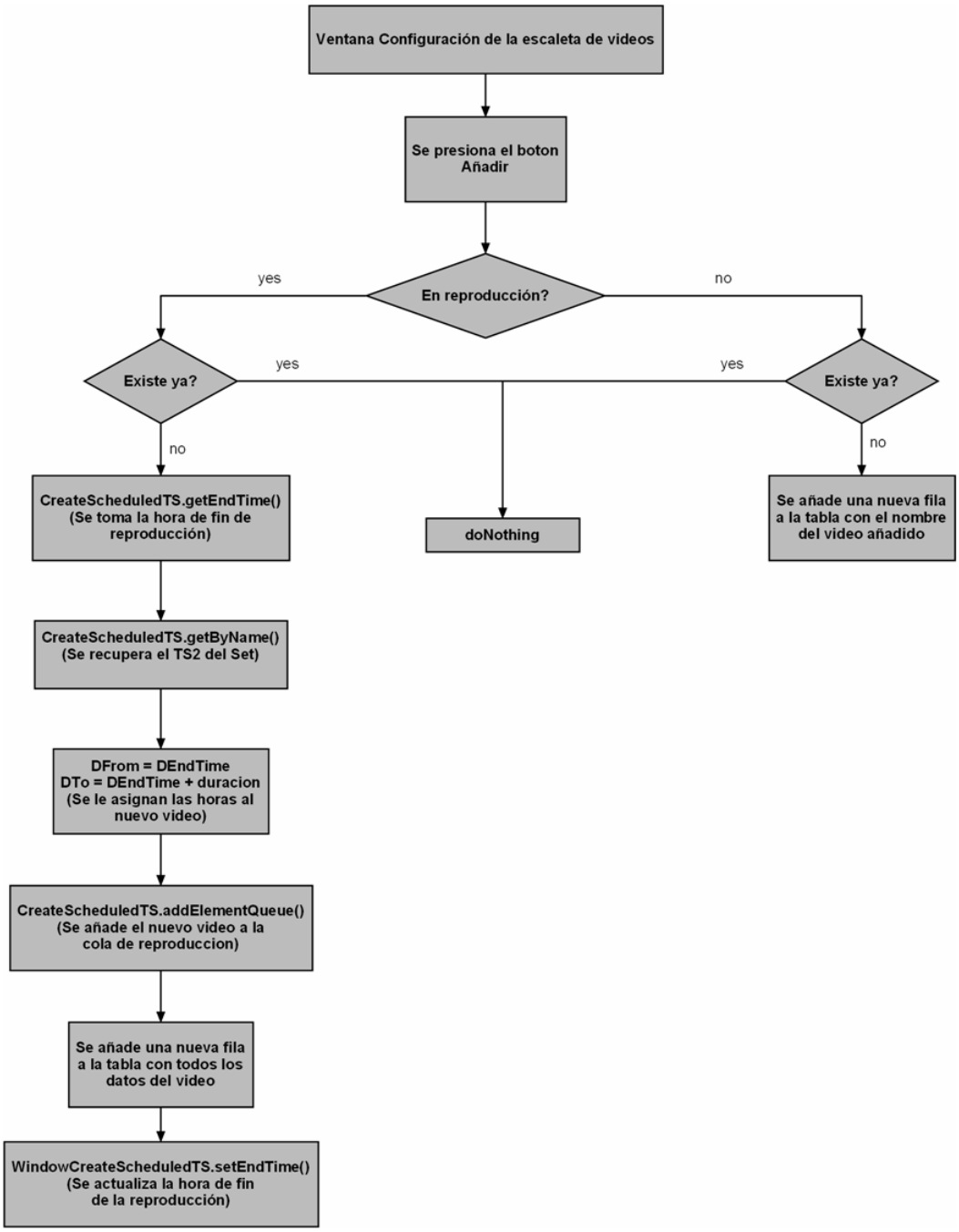


Diagrama 1.10.- *Diagrama de flujo de la acción Añadir un vídeo a la escaleta*

Como se puede ver en el **Diagrama 1.10** se tienen dos casos excluyentes:

- No se ha iniciado la reproducción:

Se comprueba si ya se ha añadido este vídeo a la escaleta que se está configurando. Para ello, si no es el primer elemento que se añade a la tabla, se va recorriendo la tabla con un bucle *while* comparando los vídeos que ya se han cargado con el que se está tratando en este momento. Si a la salida del bucle se comprueba que se ha pasado por todas la posiciones de la tabla sin encontrar ningún vídeo igual, se procede a añadirlo a la escaleta. Para añadir un vídeo a la escaleta, primero se crea la nueva fila que se va a añadir a la tabla, para ello se crea un *array* de ***String*** con el nombre del nuevo vídeo en la posición [0] y las demás celdas se dejan vacías.

```
if (jtScheduler.getRowCount() != 0)
{
      while (i<jtScheduler.getRowCount() &&
            !dtmScheduler.getValueAt(i, 0).toString().equals
             (jlistListaTs.getSelectedValue().toString()))
      {
            i++;
      }
}
if (i == jtScheduler.getRowCount())
{
     String txNameNewTs =
            jlistListaTs.getSelectedValue().toString();
     String[] arrTxNewRow = {
                              txNameNewTs,
                              null,
                              null,
                              null,
                              null,
      };
      dtmScheduler.addRow(arrTxNewRow);
}
```

- Ya se ha iniciado la reproducción:

Igual que en el caso anterior en primer lugar se comprueba si existe algún otro vídeo en la tabla con el mismo nombre. Si no existe ninguno se procede a subir el vídeo. Se coge la hora de fin de reproducción que está en los campos destinados a ello. Esta hora será el ***DFrom*** del nuevo vídeo. Para obtener la hora de fin de reproducción se utiliza el método ***getEndTime()*** de la clase ***CreateScheduledTS***.

```
Date dEndTime;
dEndTime = CreateScheduledTS.getEndTime();
```

Se recupera el vídeo del ***Set*** en una variable ***TransportStream2*** y se le asignan las nuevas horas de inicio y de fin. Como ya se ha comentado antes el ***DFrom*** del nuevo vídeo es la hora de fin de reproducción y el ***DTo*** es esta misma hora más la duración del vídeo.

```
TransportStream2 ts2New;
ts2New = CreateScheduledTS.getByName(UploadTS.hsTS,
      jlistListaTs.getSelectedValue().toString());
ts2New.setDFrom(dEndTime);
Date dTo = new Date(dEndTime.getTime() + ts2New.getLDuration());
ts2New.setDTo(dTo);
```

A continuación, se añade el nuevo vídeo a la cola de reproducción, para ello se utiliza el método ***addElementQueue()*** de la clase ***CreateScheduledTS***. Por último, se añade la nueva fila a la tabla de la escaleta, para crear la nueva fila que se va a añadir se usa el método ***rowScheduler()*** de la clase ***SchedulerTS***, y se actualiza la hora de fin de la reproducción con el método ***setEndTime()*** de la clase ***WindowCreateScheduledTS***.

```
CreateScheduledTS.addElementQueue(jlistListaTs.
      getSelectedValue().toString());
String[] arrTxNewRow;
arrTxNewRow = SchedulerTS.rowScheduler(ts2New);
dtmScheduler.addRow(arrTxNewRow);
setEndTime(dTo);
```

En el **Diagrama 1.11** se representa el caso en el que la escaleta está en reproducción.

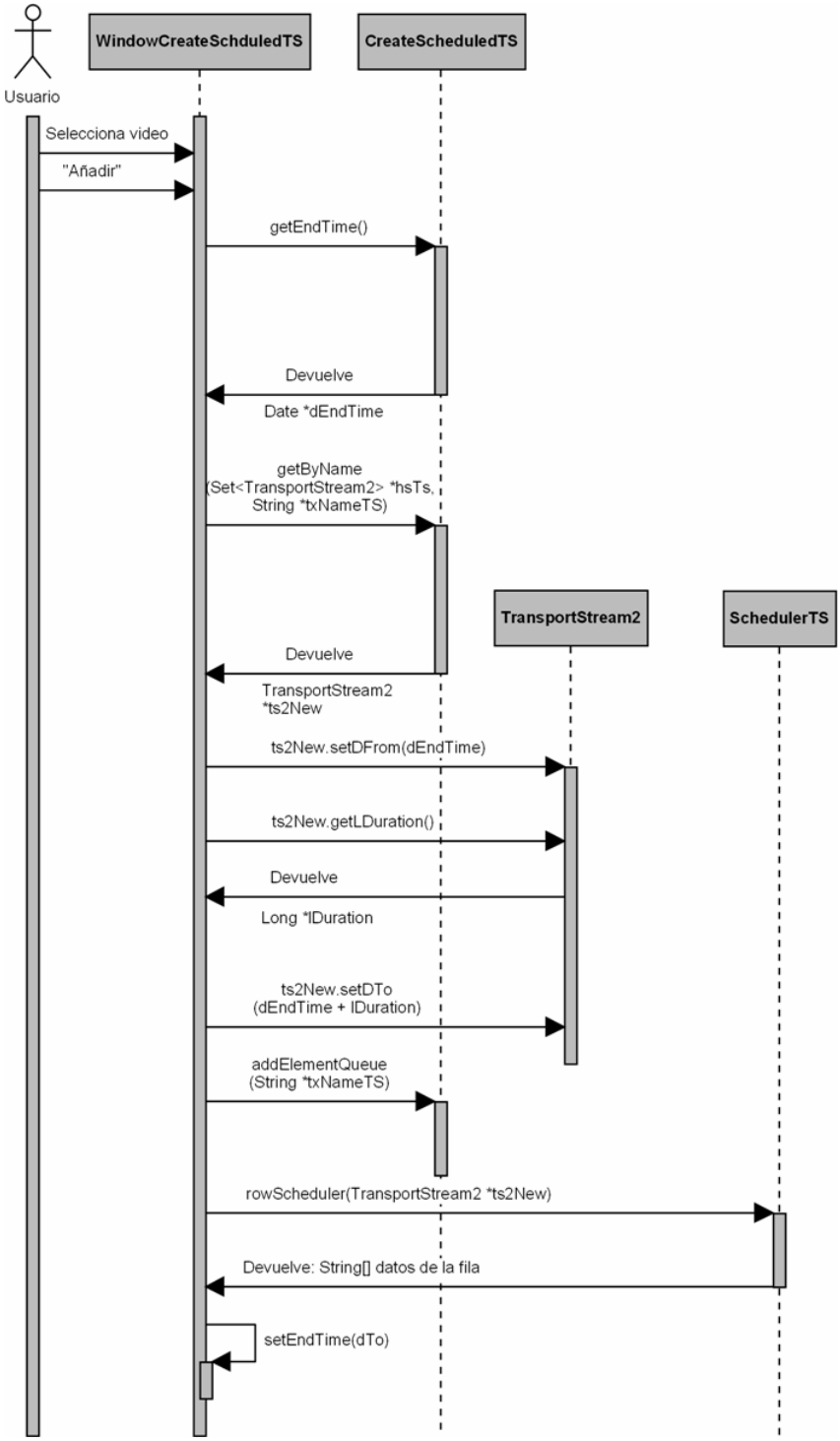


Diagrama 11.- *Diagrama de secuencia de la acción Añadir un nuevo vídeo a la escaleta*

Método: ***Date getEndTime()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***Date*** → La hora de fin de reproducción que se está consultando

Lo que se hace en este método es leer la fecha y la hora del fin de la reproducción de los campos asignados para ello en la ventana de configuración de la escaleta, y se convierte en un objeto ***Date*** que se devuelve a la salida del método.

Como ya se ha comentado antes, lo primero que se hace es leer por un lado la fecha y por el otro la hora y se guardan en dos variables ***String***.

```
String txEndDate;
txEndDate = WindowCreateSchduledTS.jtfEndDate.getText();

String txEndTime;
txEndTime = WindowCreateSchduledTS.jtfEndTime.getText();
```

Ahora lo que se tiene que hacer es cambiar el formato, es decir, las dos variables ***String*** se convierten en una variable de tipo ***Date***. Para ello se llama al método ***stringToDate()*** de la clase ***CreateScheduledTS***.

```
String[] arrTxEndTime = {
            txEndDate,
            txEndTime
};

Date dEnd;
dEnd = stringToDate(arrTxEndTime);
```

Como se puede ver en las líneas de código especificadas arriba, se crea un *array* de ***String*** con las dos variables ***String*** que contienen la fecha y la hora de fin de reproducción leídas de la aplicación, este *array* se pasa como parámetro de entrada al método ***stringToDate()***.

Método: ***Date stringToDate(String[] *arrTxDateTime)***

- *Parámetros de entrada:*
  String[] *arrTxDateTime → *Array* de ***String*** con la fecha y la hora que se quiere convertir
- *Salida:* ***Date*** → Objeto ***Date*** resultante de la conversión

El porqué de este método se explica si se tiene en cuenta la diferencia entre los formatos de una misma hora dependiendo de si se utiliza a nivel interno de programa o si se quiere representar en la interfaz visible de la aplicación. Así por ejemplo si se quiere guardar este dato en un campo de una variable ***TransportStream2*** se necesita que sea un objeto ***Date***, por el contrario, en la interfaz de la aplicación se ve representada en dos campos de texto (uno para la fecha y el otro para la hora propiamente dicha).

En primer lugar se crean dos variables ***String*** donde se recuperan la fecha y la hora que se pasa como parámetro de entrada.

```
String txDate;
txDate = arrTxDateTime[0];
String txTime;
txTime = arrTxDateTime[1];
```

A continuación se procede a realizar la conversión. Como se conoce el formato en que se lee tanto la hora como la fecha:

- Fecha → Año-Mes-Día
- Hora → Hora:minutos:segundos

Con ayuda de un bucle *for* se divide cada variable ***String*** en tres componentes, de manera que de la variable ***txDate*** se obtiene por separado el año, el mes y el día, respectivamente, de la variable ***txTime*** se obtiene la hora, los minutos y los segundos.

```
int j = 0;
int k = 0;
int iIndexDate = 0;
int iIndexTime = 0;
String[] arrTxDate = new String[3];
String[] arrTxTime = new String[3];

for (int i=0; i<2; i++)
{
      iIndexDate = txDate.indexOf("-", j);
      arrTxDate[i] = txDate.substring(j, iIndexDate);
      j = iIndexDate + 1;

      iIndexTime = txTime.indexOf(":", k);
      arrTxTime[i] = txTime.substring(k, iIndexTime);
      k = iIndexTime + 1;
}
arrTxDate[2] = txDate.substring(j, txDate.length());
arrTxTime[2] = txTime.substring(k, txTime.length());
```

Como se puede ver estos *substring* se guardan en dos *array* de ***String*** de longitud 3. Una vez que se han separado los 6 componentes se crea el objeto ***Date*** con el método ***getDate()***

```
dDate = getDate(arrTxDate[0], arrTxDate[1], arrTxDate[2],
      arrTxTime[0], arrTxTime[1], arrTxTime[2]);
```

Método: ***Date getDate(String *txYear, String *txMonth, String *txDay, String *txHour, String *txMin, String *txSec)***

- *Parámetros de entrada:*
  - ***String *txYear*** → Campo año de la fecha
  - ***String *txMonth*** → Campo mes de la fecha
  - ***String *txDay*** → Campo día de la fecha
  - ***String *txHour*** → Campo hora de la fecha
  - ***String *txMin*** → Campo minutos de la fecha
  - ***String *txSec*** → Campo segundos de la fecha
- *Salida:* ***Date*** → objeto ***Date*** resultante de los parámetros de entrada

Lo primero que se hace en este método es transformar los parámetros de entrada en objetos ***int*** para poder trabajar con ellos.

```
iYear = Integer.parseInt(txYear);
iMonth = Integer.parseInt(txMonth)-1;
iDay = Integer.parseInt(txDay);
iHour = Integer.parseInt(txHour);
iMin = Integer.parseInt(txMin);
iSec = Integer.parseInt(txSec);
```

Como se puede ver, al campo mes se le resta 1 tras la transformación. Se trata de un ajuste provocado por la configuración de la clase ***Calendar*** que se usa para hacer el cambio de formato, en esta clase Enero corresponde al mes 0, por lo tanto si se quiere poner una fecha de mes de Mayo que la forma natural de definirlo es el mes 5 para la clase ***Calendar*** este mes corresponde al cuarto mes.

Como ya se ha comentado, se va a utilizar un objeto ***Calendar***, así que se hace una instancia al objeto y se le pasan los valores.

```
Calendar cCalendar = Calendar.getInstance();
cCalendar.set(iYear, iMonth, iDay, iHour, iMin, iSec);
```

Por último se crea el objeto ***Date*** que retorna el método, para ello se toma el valor en milisegundos de ***cCalendar***, y este dato se usa como parámetro de entrada en la creación del objeto ***Date***.

```
Long lCalendar;
lCalendar = cCalendar.getTimeInMillis();
Date dCalendar = new Date(lCalendar);
```

Los dos métodos que se acaban de describir se encuentran en la clase ***DateFunctions*** que contiene las funciones auxiliares que se utilizan para facilitar el manejo de las horas y las fechas en la configuración de las escaletas. Esta clase se completa con el método ***dateToString()***, que realiza la función opuesta al método ***stringToDate()***, como se puede deducir de sus nombres.

Método: ***String[] dateToString(Date *dDate)***

- *Parámetros de entrada:*
  Date *dDate → Fecha que se quiere cambiar de formato.
- *Salida:* ***String[]*** → *Array* de ***String*** de tamaño 2 con la fecha y la hora dadas en el parámetro de entrada

En este método se recibe un objeto ***Date*** como parámetro de entrada, este objeto se quiere convertir en dos ***String*** (uno con la fecha y otro con la hora), que se devuelven en un *array* .

En este caso también se utiliza un objeto ***Calendar*** como intermediario entre los dos formatos. En primer lugar se instancia el objeto y se le da el valor del parámetro de entrada.

```
Calendar cTimeEnd = Calendar.getInstance();
cTimeEnd.setTime(dDate);
```

Después se obtienen los distintos campos de la fecha con el método ***get()*** que proporciona la clase ***Calendar***.

```
int iYear = cTimeEnd.get(Calendar.YEAR);
int iMonth = cTimeEnd.get(Calendar.MONTH);
int iDay = cTimeEnd.get(Calendar.DAY_OF_MONTH);

int iHour = cTimeEnd.get(Calendar.HOUR_OF_DAY);
int iMinute = cTimeEnd.get(Calendar.MINUTE);
int iSecond = cTimeEnd.get(Calendar.SECOND);
```

Una vez se han separado los 6 campos de la fecha se agrupan en dos *array* de ***String***, uno con la fecha y otro con la hora. En el caso del *array* que representa la hora, al pasar los campos a ***String***, se completan los valores menores de 10 con un cero adicional delante del valor. Esto último es una cuestión de estética, se ha decidido que el formato de las horas contenga dos dígitos en cada campo, de este modo si se quiere representar la una y cinco, quedaria 01:05:00, en lugar de 1:5:0.

```
String[] arrTxDate = {
            Integer.toString(iYear),
            Integer.toString(iMonth + 1),
            Integer.toString(iDay)
};

int[] arrIntTime = {
            iHour,
            iMinute,
            iSecond
};
String[] arrTxTime = {
            Integer.toString(iHour),
            Integer.toString(iMinute),
            Integer.toString(iSecond)
};

/*
 * Se completa el formato de la hora añadiendo ceros delante de
 * los numeros menores de 10
 * */
for(int i=0; i<arrIntTime.length; i++)
{
      if(arrIntTime[i]<10)
      {
            String txAux;
            txAux = "0" + Integer.toString(arrIntTime[i]);
            arrTxTime[i] = txAux;
      }
}

String txDate = new String(arrTxDate[0] + "-" + arrTxDate[1] + "-" +
      arrTxDate[2]);

String txTime = new String(arrTxTime[0] + ":" + arrTxTime[1] + ":" +
      arrTxTime[2]);
```

Con los elementos de los *array* se forman los ***String*** con los formatos correspondientes a cada tipo de dato (la fecha separada por "-" y la hora separada por ":")

Por último, se agrupan los dos ***String*** en el *array* que se retorna como parámetro de salida del método.

```
String[] arrTxDateTime = {
            txDate,
            txTime
};

return arrTxDateTime;
```

Método: ***void addElementQueue(String *txNewVideo)***

- *Parámetros de entrada:*
  String *txNewVideo → Nombre del nuevo vídeo
- *Salida:* ***void***

Este método añade el vídeo que se está cargando a la cola de reproducción. Como ya se ha comentado antes, para la reproducción los vídeos se guardan en una cola de la que se alimenta la aplicación para seguir correctamente el orden de reproducción. Cada vídeo que se añade a la escaleta se debe incluir en la cola de reproducción para que la aplicación se entere de que se debe reproducir cuando llegue el momento. Los vídeos añadidos se colocan al final de la escaleta así que como se trata de una cola FIFO se añade directamente al final de la cola.

Al entrar en el método se añade el objeto ***String*** que se pasa como parámetro, para ello se usa el método ***add()*** que proporciona la clase ***Queue*** de Java.

```
Boolean boAdd = qTsScheduled.add(txNewVideo);
```

Método: ***String[] rowScheduler(TransportStream2 *ts2Row)***

- *Parámetros de entrada:*
  String *ts2Row → Elemento al que pertenecen los datos que contiene la fila de la tabla de reproducción
- *Salida:* ***String[]*** → *Array* de objetos ***String*** que componen la nueva fila de la tabla de reproducción

Este método, dado un objeto ***TransportStream2***, se encarga de configurar los datos que se incluyen en una fila de la tabla que representa la escaleta de vídeos. Cada fila de esta tabla contiene 5 datos: nombre del vídeo, fecha de inicio, hora de inicio, fecha de fin y hora de fin.

Todos los datos necesarios para configurar la fila se obtienen del parámetro de entrada ****ts2Row***, en este tipo de objetos tanto el inicio como el final del vídeo se guardan como variables de tipo ***Date*** por lo tanto se usa el método ***dateToString()*** para obtener dos ***String*** que es lo que se necesita.

```
txRowNameTS = ts2Row.getTxName();

/*
 * Se obtiene la hora de inicio del video
 * */
Date dFromRow;
dFromRow = ts2Row.getDFrom();
String[] arrTxDFromRow;
arrTxDFromRow = DateFunctions.dateToString(dFromRow);
txRowInitDate = arrTxDFromRow[0];
txRowInitHour = arrTxDFromRow[1];

/*
 * Se obtiene la hora de fin del video
 * */
Date dToRow;
dToRow = ts2Row.getDTo();
String[] arrTxDToRow;
arrTxDToRow = DateFunctions.dateToString(dToRow);
txRowEndDate = arrTxDToRow[0];
txRowEndHour = arrTxDToRow[1];
```

Por último, se agrupan todos los datos en un *array* de ***String***.

```
String[] arrTxRowScheduler = {
            txRowNameTS,
            txRowInitDate,
            txRowInitHour,
            txRowEndDate,
            txRowEndHour
};
return arrTxRowScheduler;
```

Método: ***void setEndTime(Date *dEnd)***

- *Parámetros de entrada:*
  String *dEnd → Nueva hora de fin de la reproducción
- *Salida:* ***void***

Con este método se actualiza la hora de fin de la reproducción cada vez que se modifica la escaleta de vídeos.

Como la hora de fin de la reproducción está representada por dos campos de texto, se debe convertir el parámetro de entrada ****dEnd*** en dos variables ***String***. Estas dos variables se pasan como parámetros del método ***setText()*** para asignar los nuevos valores a los campos de texto.

```
String[] arrTxDateTime;
arrTxDateTime = DateFunctions.dateToString(dEnd);

String txDate = arrTxDateTime[0];
String txTime = arrTxDateTime[1];

jtfEndDate.setText(txDate);
jtfEndTime.setText(txTime);
```

1.1.5.- Subir una posición el vídeo en la escaleta

En muchos casos tras añadir un vídeo a la escaleta puede interesar cambiar de posición algún vídeo, para eso se han añadido estas funciones de subir y bajar vídeos que permiten cambiar el orden de reproducción, tanto antes de que comience la reproducción como durante la misma. En el **Diagrama 1.12** se puede ver el diagrama de flujo de esta operación.

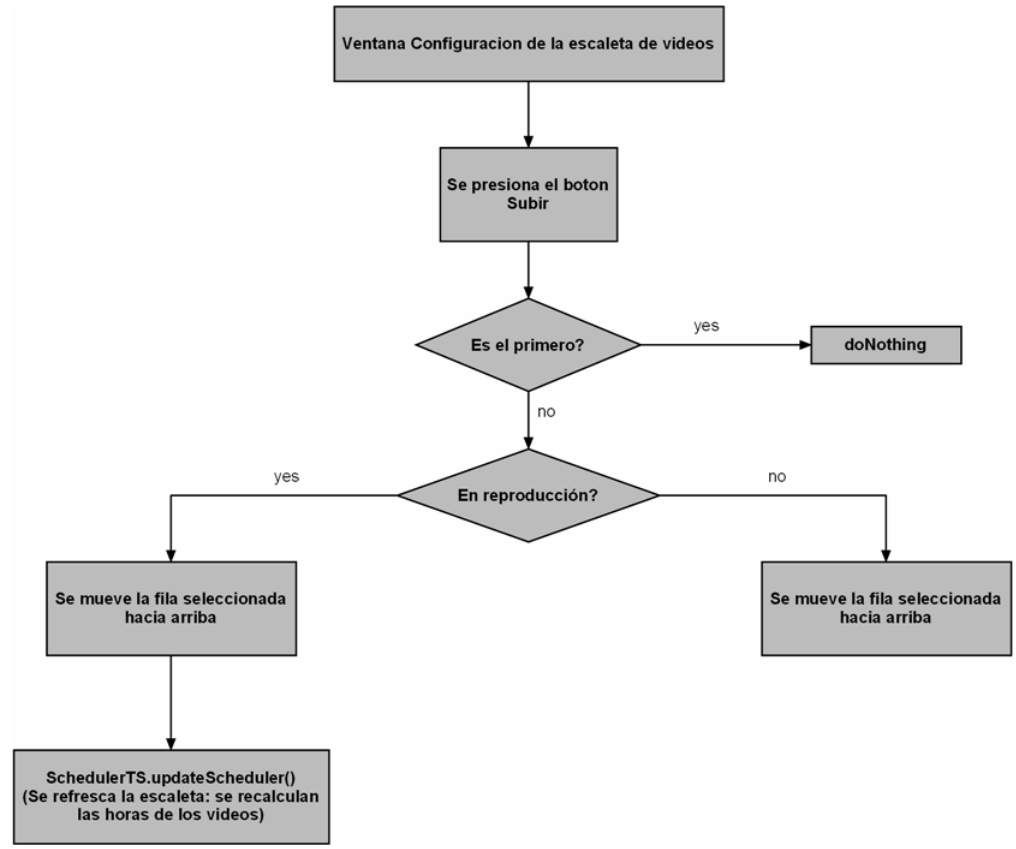


Diagrama 1.12.- *Diagrama de flujo de la acción Subir una posición el vídeo en la escaleta*

Tras seleccionar en la tabla el vídeo que se quiere subir, se presiona el botón Subir, y cuando el *listener* recibe el evento se llama a la función ***upTS()*** de la clase ***SchedulerTS***.

```
if (arg0.getSource().equals(jbUp))
{
      int iIndexUp;

      iIndexUp = jtScheduler.getSelectedRow();
      SchedulerTS.upTs(iIndexUp);
}
```

A continuación, se puede ver en el **Diagrama 1.13** el diagrama de secuencia de la acción subir una posición el vídeo en la escaleta.

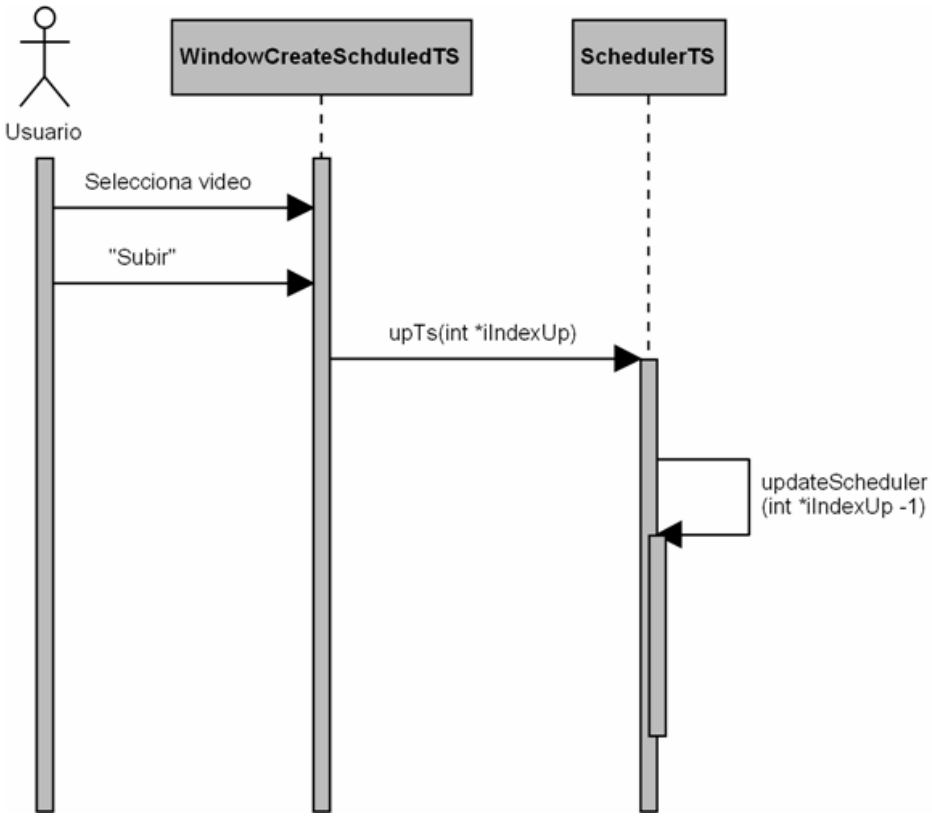


***Diagrama 1.13**.- Diagrama de secuencia de la acción Subir una posición el vídeo en la escaleta*

Método: ***void upTs(int *iIndexTS)***

- *Parámetros de entrada:*
  int *iIndexTS → Índice del vídeo que se va a subir
- *Salida:* ***void***

En primer lugar se comprueba que el vídeo seleccionado no sea el primer vídeo de la tabla, ya que si es así no tiene ningún sentido subir el vídeo puesto que no puede situarse más arriba de la primera posición.

Si no es el primer vídeo de la tabla, igual que en el caso anterior, se tiene dos opciones:

- *No se ha iniciado la reproducción:*

Se mueve la fila seleccionada una posición hacia arriba. Para ello se usa el método ***moveRow()*** que pertenece a la clase ***DefaultTableModel*** de Java.

- *Ya se ha iniciado la reproducción:*

Se mueve la fila seleccionada una posición hacia arriba y se refresca la escaleta, es decir, se recalculan las nuevas horas de inicio y fin. Para realizar el refresco de la escaleta se llama al método ***updateScheduler()*** de la clase ***SchedulerTS***.

```
if (iIndexTS != 0)
{
	//Si la escaleta no esta en reproduccion
	if(WindowCreateSchduledTS.jbSave.isEnabled())
	{
		WindowCreateSchduledTS.dtmScheduler.moveRow(iIndexTS,
							iIndexTS, iIndexTS - 1);
	}
	//Si la escaleta esta en reproduccion
	else
	{
		WindowCreateSchduledTS.dtmScheduler.moveRow(iIndexTS,
							iIndexTS, iIndexTS - 1);

		updateScheduler(iIndexTS -1);
	}
}
```

Método: ***void updateScheduler(int *iIndexEdit)***

- *Parámetros de entrada:*
  int *iIndexEdit → Índice a partir del cual se actualiza la escaleta
- *Salida:* ***void***

Este método actualiza la escaleta de vídeos tras una edición, cambiar la posición de un vídeo, un borrado,…

En primer lugar se obtiene la hora de inicio auxiliar, esta hora se va a utilizar para calcular a partir de ella las nuevas horas de inicio y fin de cada vídeo.

Si se está refrescando la escaleta desde la primera posición, la hora de inicio auxiliar se toma del ***DTo*** del vídeo que se representa en el campo de siguiente reproducción. Para el resto de los casos la hora de inicio auxiliar se toma del vídeo que se encuentra en la posición inmediatamente anterior. En ambos casos se toma el nombre del vídeo del que se va a consultar el ***DTo*** y con el método ***getByName()*** de la clase ***CreateScheduledTS*** se recupera del ***Set***, la consulta se realiza con el método ***getDTo()*** definido para los objetos de tipo ***TransportStream2***.

```
Date dInitAux;
if(iIndexEdit == 0)
{
	String txNextTs;
	txNextTs = WindowCreateSchduledTS.jtfTsNext.getText();
	TransportStream2 ts2NextTs;
	ts2NextTs = CreateScheduledTS.getByName(UploadTS.hsTS,
		txNextTs);
	dInitAux = ts2NextTs.getDTo();
}
else
{
	String txBeforeTS;
	txBeforeTS = WindowCreateSchduledTS.dtmScheduler.getValueAt
		(iIndexEdit - 1, 0).toString();
	//si el vídeo anterior es la cortinilla se debe tomar como
	//hora de inicio la del video
	//anterior porque la cortinilla no cuenta y se va a
	//eliminar.
	if (txBeforeTS == txNameDefVideo)
	{
		txBeforeTS =
			WindowCreateSchduledTS.dtmScheduler.getValueAt
			(iIndexEdit - 2, 0).toString();
		iIndexEdit = iIndexEdit -1;
	}


	TransportStream2 ts2Before;
	ts2Before = CreateScheduledTS.getByName(UploadTS.hsTS,
		txBeforeTS);
	dInitAux = ts2Before.getDTo();
}
```

Se puede apreciar que se está considerando un caso especial cuando el vídeo anterior del que se pretende obtener la hora de inicio auxiliar es una cortinilla (estas cortinillas se añaden como consecuencia de un ajuste en la hora de inicio de algún vídeo, más adelante se hablará con más detalle de ellas). En este caso se obvia la fila que corresponde a la cortinilla y se toma el ***DTo*** del vídeo anterior a esta. También se resta 1 al valor del índice, esto es un reajuste necesario ya que cuando se quita la fila correspondiente a la cortinilla el vídeo desde el que se esta recalculando la escaleta sube una posición, de modo que su índice se debe reducir en 1.

A continuación se repasan todas las posiciones de la tabla de reproducción que se van a refrescar para localizar las cortinillas y se guarda su posición en un *array* que se utiliza para borrarlas con ayuda de un bucle *for*. Aunque se tienen guardados los índices de las cortinillas en un *array*, a la hora de borrar estas filas se necesita hacer un reajuste en el

valor de los índices, ya que cada vez que se borra una cortinilla, las filas posteriores se desplazan una posición hacia arriba,

```
int j=0;
int[] arrIntIndexDefVideo = new
      int[WindowCreateSchduledTS.jtScheduler.getRowCount()];
int k = 0;
for (int i=0; i<arrTxTsSelec.length; i++)
{
      String txNameTsAux;
      txNameTsAux =
            WindowCreateSchduledTS.dtmScheduler.getValueAt
            (i, 0).toString();
      if (txNameTsAux != txNameDefVideo)
      {
            arrTxTsSelec[j] = txNameTsAux;
            j++;
      }
      else
      {
            arrIntIndexDefVideo[k] = i;
            k++;
      }
}

int c = 0;
for(int i=0; i<k; i++)
{
      if (arrIntIndexDefVideo[i] >= iIndexEdit)
      {
            WindowCreateSchduledTS.dtmScheduler.removeRow
                                          (arrIntIndexDefVideo[i] - c);
            c++;
      }
}
```

En los casos que no se trate de una cortinilla se guarda el nombre de los vídeos en otro *array* de ***String*** que se usa para tratar los vídeos y asignarles las nuevas horas de reproducción.

Después de la preparación de los datos se procede a asignar las nuevas horas de inicio y fin de reproducción de cada vídeo. Para ello se usa un bucle *for* que recorre la tabla desde el vídeo modificado hasta el final de la tabla. El nuevo valor del ***DFrom*** será el ***DTo*** del vídeo anterior, y el ***DTo*** se calcula sumando la duración. Una vez que se han asignado las nuevas horas de inicio y fin de la reproducción en el objeto ***TransportStream2*** se actualiza este vídeo en el ***Set*** con el método ***editTS()*** de la clase ***CreateScheduledTS***. Además de modificar el ***Set*** también se debe actualizar los datos de la tabla de reproducción ya que es la parte visible de la reproducción, para ello en cada vuelta del

bucle se compone la fila correspondiente con el método ***rowScheduler()*** de la clase ***SchedulerTS***. A la salida del *bucle* se actualiza la hora de fin de reproducción y se le da como valor el ***DFrom*** del último vídeo de la tabla.

```
int iLengthTsSelect = j;
if(iLengthTsSelect != 0)
{
      String txTsName;
      Date dFromAux;
      Date dToAux;
      dFromAux = dInitAux;
      for(int i=iIndexEdit; i<iLengthTsSelect; i++)
      {
            txTsName = arrTxTsSelec[i];
            TransportStream2 ts2;
            ts2 = CreateScheduledTS.getByName(UploadTS.hsTS,
                  txTsName);
            if(ts2 != null)
            {
                  dToAux = new Date (dFromAux.getTime() +
                        ts2.getLDuration());
                  ts2.setDFrom(dFromAux);
                  ts2.setDTo(dToAux);
                  dFromAux = dToAux;
            }
            CreateScheduledTS.editTS(UploadTS.hsTS, ts2);

            String[] arrTxRowSchedulerData =
                  SchedulerTS.rowScheduler(ts2);
            WindowCreateSchduledTS.dtmScheduler.setValueAt
                  (arrTxRowSchedulerData[1], i, 1);
            WindowCreateSchduledTS.dtmScheduler.setValueAt
                  (arrTxRowSchedulerData[2], i, 2);
            WindowCreateSchduledTS.dtmScheduler.setValueAt
                  (arrTxRowSchedulerData[3], i, 3);
            WindowCreateSchduledTS.dtmScheduler.setValueAt
                  (arrTxRowSchedulerData[4], i, 4);
      }
      String txLastTS;
      txLastTS = arrTxTsSelec[iLengthTsSelect - 1];
      TransportStream2 ts2Last;
      ts2Last = CreateScheduledTS.getByName(UploadTS.hsTS,
            txLastTS);
      Date dEnd;
      dEnd = ts2Last.getDTo();
      CreateScheduledTS.editTS(UploadTS.hsTS, ts2Last);
      WindowCreateSchduledTS.setEndTime(dEnd);
}
```

Por último, se refrescan los valores de la cola de reproducción. Primero limpia la cola para eliminar la configuración, y se vuelve a llenar con la nueva composición de la

escaleta. Este último paso se realiza con un bucle *for*, en el que en cada iteración se añade un elemento a la cola con el método ***addElementQueue()*** de la clase ***CreateScheduledTS***.

```
CreateScheduledTS.qTsScheduled.clear();
for (int i=0; i<iLengthTsSelect; i++)
{
      String txNameAux;
      txNameAux = arrTxTsSelec[i];
      CreateScheduledTS.addElementQueue(txNameAux);
}
```

Método: ***void editTS(Set<TransportStream2> *hsTs2, TransportStream2 *ts2Edit)***

- *Parámetros de entrada:*
  Set<TransportStream2> *hsTs2 → ***Set*** sobre el que se está trabajando
  TransportStream2 *ts2Edit → Objeto ***TransportStream2*** que se quiere editar
- *Salida:* ***void***

Se trata de un método auxiliar que se encarga de editar un objeto ***TransportStream2*** que se tiene guardado en el ***Set***. Se trata de una función muy sencilla que se compone únicamente de dos instrucciones, en primer lugar se borra el elemento antiguo del ***Set*** y posteriormente se reemplaza por el objeto nuevo. De manera que el método queda de la siguiente forma:

```
TransportStream2 ts2 = ts2Edit;

hsTs2.remove(ts2);

hsTs2.add(ts2Edit);
```

La variable ****ts2*** se define para que no se pierda el elemento que se está guardando al borrar el antiguo del ***Set***.

1.1.6.- Bajar una posición el vídeo en la escaleta

El diagrama de flujo de esta operación (**Diagrama 1.14**) es muy similar al que se tiene para la acción de Subir descrita anteriormente (**Diagrama 1.12**).

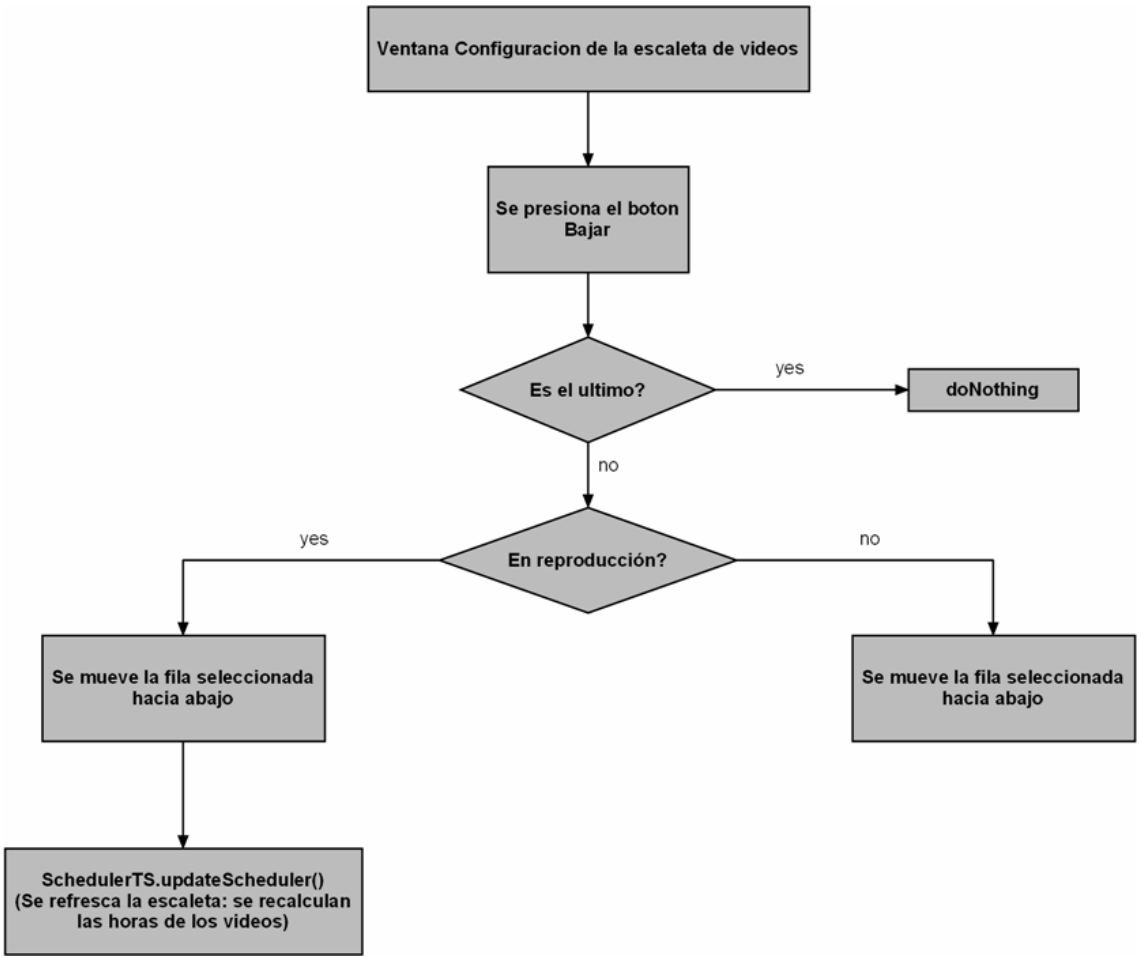


Diagrama 1.14.- *Diagrama de flujo de la acción Bajar una posición el vídeo en la escaleta*

Esta funcionalidad es prácticamente igual que la descrita anteriormente (Subir), en este caso cuando la aplicación recibe el *listener* que indica que se ha presionado el botón Bajar de la ventana de configuración de la escaleta de vídeos se llama al método ***downTs()*** de la clase ***SchedulerTS***.

```
if (arg0.getSource().equals(jbDown))
{
      int iIndexDown;

      iIndexDown = jtScheduler.getSelectedRow();
      SchedulerTS.downTs(iIndexDown);
}
```

El diagrama de secuencia de esta operación se puede ver en el **Diagrama 1.15** que se incluye más abajo.

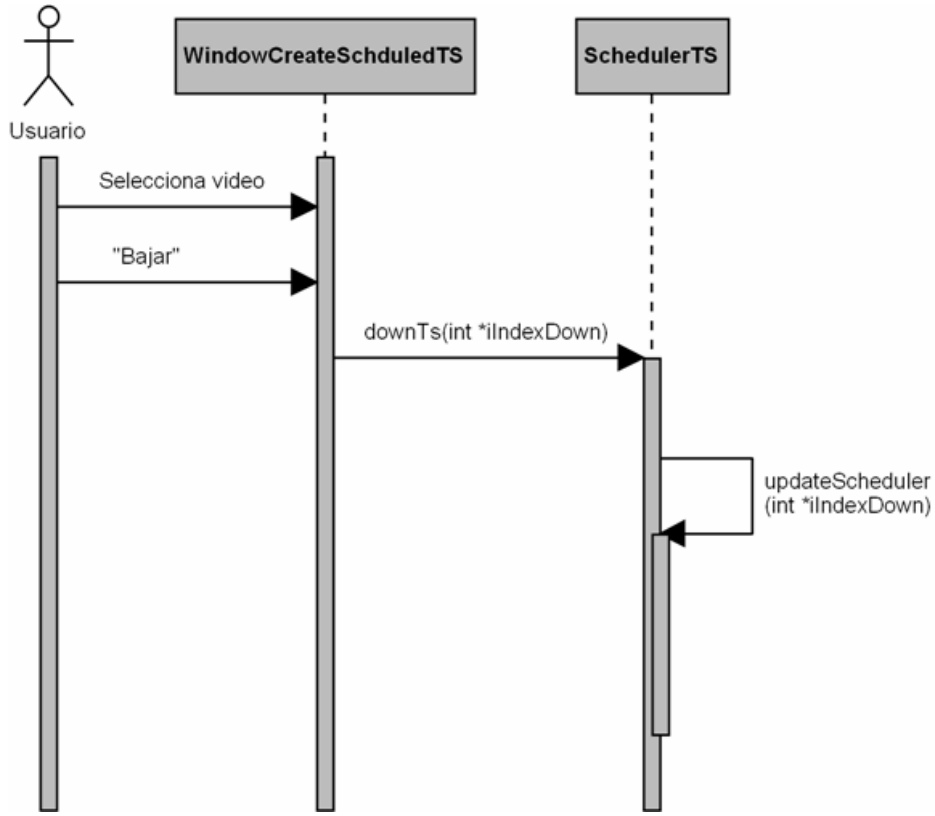


Diagrama 1.15.- *Diagrama de secuencia de la acción Bajar una posición el vídeo en la escaleta*

Método: ***void downTs(int *iIndexTS)***

- *Parámetros de entrada:*
  int *iIndexTs → Índice del vídeo que se va a bajar
- *Salida:* ***void***

La estructura de este método es igual que la del método ***upTs()*** que se ha descrito para la funcionalidad anterior. En este caso en lugar de mover las filas hacia arriba se mueven hacia abajo.

```
if (iIndexTS != WindowCreateSchduledTS.jtScheduler.getRowCount() - 1)
{
      //Si la escaleta no esta en reproduccion
      if(WindowCreateSchduledTS.jbSave.isEnabled())
      {
            WindowCreateSchduledTS.dtmScheduler.moveRow(iIndexTS,
                  iIndexTS, iIndexTS + 1);
      }
      //Si la escaleta esta en reproduccion
      else
      {
            WindowCreateSchduledTS.dtmScheduler.moveRow(iIndexTS,
                  iIndexTS, iIndexTS + 1);
            updateScheduler(iIndexTS);
      }
}
```

1.1.7.- Editar la hora de inicio de un vídeo

Otra de las opciones implementadas en la aplicación se encarga de modificar la hora de inicio de los vídeos. Como ya se ha comentado, cuando se añade un nuevo vídeo a la escaleta éste se añade al final de la misma y toma como hora de inicio la hora de fin del vídeo anterior. Ya se ha visto que se puede variar la posición de los vídeos con los botones Subir y Bajar, pero ¿que ocurre si la posición está bien pero se quiere cambiar la hora de inicio?, esto es muy frecuente en el mundo de la televisión, se quiere que un programa empiece a las 15:00 (por ejemplo), pero no a las 14:55 que es la hora calculada por la aplicación, para estos casos se ha implementado esta funcionalidad.

Esta función únicamente se encuentra disponible cuando la escaleta está en reproducción, así que hasta que no se comienza la reproducción el botón Editar se encuentra deshabilitado. Esto es así debido a la forma en que se ha implementado la configuración de la escaleta. Antes de que comience la reproducción, la escaleta consiste en una lista de vídeos ordenados según se quiera que se reproduzcan, pero en la que los campos de inicio y fin de cada vídeo están vacíos. Una vez que se pulsa el botón Play, se calculan todas estas horas, esto se explica con más detalle más adelante cuando se describa la reproducción de la escaleta. De modo que antes de que comience la reprodución, no se tienen las horas de inicio y por tanto no se pueden modificar.

Esta funcionalidad se puede dividir en dos partes, por un lado se atrasa la hora de inicio del vídeo, y para asegurar la continuidad de la reproducción de la escaleta se rellena el *gap* que queda entre el final del vídeo anterior y el inicio del vídeo editado con una

cortinilla. Esta cortinilla, como se verá más adelante, es un vídeo auxiliar que se reproduce de forma cíclica hasta que llega la hora de inicio del siguiente vídeo. Se debe tener en cuenta que en esta aplicación sólo se permite atrasar la hora de inicio de los vídeos, el motivo por el cual no se puede adelantar esta hora es porque un adelanto en la hora de inicio de un vídeo supondría un recorte en el vídeo anterior, ya que al tratarse de una reproducción contínua la hora de inicio de un vídeo coincide con la hora final del vídeo anterior. Así que si se quiere adelantar la hora de inicio de un vídeo 10 min y el vídeo anterior es por ejemplo una película, se estarían recortando los últimos 10 min de la película.

A continuación se incluye el diagrama de flujo de la operación Editar la hora de inicio de un vídeo (**Diagrama 1.16**).

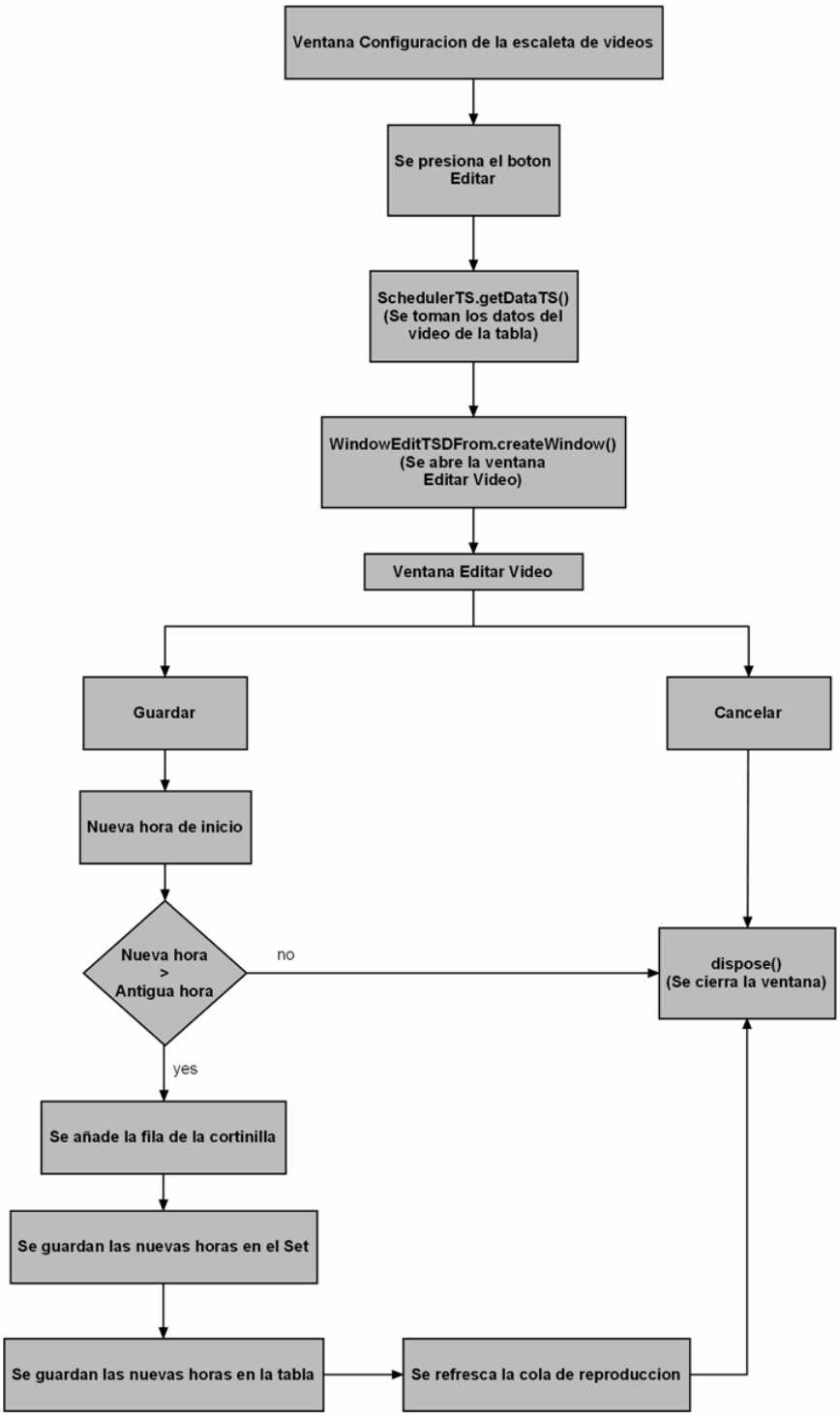


Diagrama 1.16.- *Diagrama de flujo de la acción Editar la hora de inicio de un vídeo*

Para describir esta función se asume que la escaleta se encuentra en reproducción, ya que de lo contrario no es posible que se accione el botón Editar. Cuando la aplicación recibe el evento que indica que se ha accionado el botón se lanza el método ***getDataTS()*** de la clase ***SchedulerTS***. Este método se encarga de consultar los datos del vídeo que se quiere editar (nombre del vídeo, Fecha y hora de inicio y fecha y hora de fin), estos datos se utilizan para completar los campos de la ventana de edición.

```
if (arg0.getSource().equals(jbEdit))
{
      int iIndexEdit;

      iIndexEdit = jtScheduler.getSelectedRow();
      SchedulerTS.getDataTS(iIndexEdit);
}
```

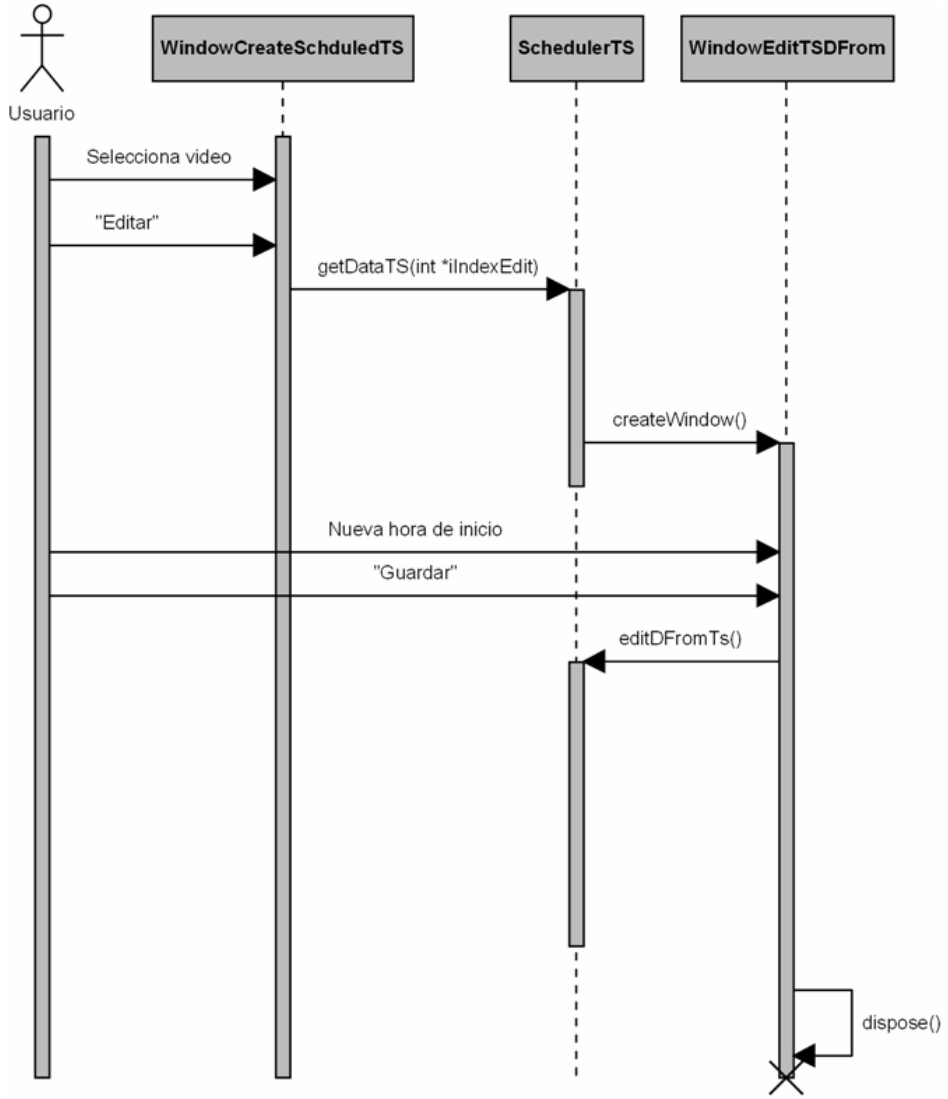


Diagrama 1.17.- *Diagrama de secuencia de la acción Editar la hora de inicio de un vídeo*

Método: ***void getDataTS(int *iIndexTS)***

- *Parámetros de entrada:*
  int *iIndexTS → Índice del vídeo que se quiere editar
- *Salida:* ***void***

Como se acaba de comentar este método se encarga de tomar los datos del vídeo que se está editando de la tabla y se los asigna a los campos de la ventana que se lanza un vez que se tienen todos los datos. Esta ventana se abre al ejecutar el método ***createWindow()*** de la clase ***WindowEditTSDFrom***.

```
String txNameTs;
txNameTs =
      WindowCreateSchduledTS.dtmScheduler.getValueAt(iIndexTS,0)
      .toString();
WindowEditTSDFrom.txVideoSt = txNameTs;
WindowEditTSDFrom.txInitDateSt =
      WindowCreateSchduledTS.dtmScheduler.getValueAt(iIndexTS,1)
      .toString();
WindowEditTSDFrom.txInitTimeSt =
      WindowCreateSchduledTS.dtmScheduler.getValueAt(iIndexTS,2)
      .toString();
WindowEditTSDFrom.txEndDateSt =
      WindowCreateSchduledTS.dtmScheduler.getValueAt(iIndexTS,3)
      .toString();
WindowEditTSDFrom.txEndTimeSt =
      WindowCreateSchduledTS.dtmScheduler.getValueAt(iIndexTS,4)
      .toString();

wetsdf = new WindowEditTSDFrom();
wetsdf.createWindow();
```

Esta ventana se compone de 5 campos de texto, de los cuales solo dos (fecha y hora de inicio de la reproducción) se pueden modificar.

Figura 1.6.- *Ventana Editar Vídeo*

Como se puede ver en la **Figura 1.6**, una vez que se ha introducido la nueva hora de inicio se presiona el botón Guardar para confirmar el cambio. Esto provoca que se tomen los nuevos valores para la hora de inicio y se guarden en dos variables ***String***, a continuación se hacen efectivos los cambios, para ello se llama al método ***editDFromTs()*** de la clase ***SchedulerTS***.

```
if (arg0.getSource().equals(jbSaveNewDFrom))
{
      txInitDateSt = jtfInitDate.getText();
      txInitTimeSt = jtfInitTime.getText();
```

```
      SchedulerTS.editDFromTs();

      jfEditDFromTS.dispose();
}
```

Método: ***void editDFromTs()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

Este método edita el vídeo al que se le ha cambiado la hora de inicio, también se reconfigura la escaleta a partir del vídeo que se ha editado.

En primer lugar, se recupera la nueva hora de inicio en dos variables ***String*** y se convierte a un objeto ***Date*** para que se pueda trabajar con ella, para ello se utiliza el método ***stringToDate()*** de la clase ***DateFuntions***.

```
int iIndexEdit;
iIndexEdit =
      WindowCreateSchduledTS.jtScheduler.getSelectedRow();

String txDateInit = WindowEditTSDFrom.txInitDateSt;
String txTimeInit = WindowEditTSDFrom.txInitTimeSt;
String[] arrTxInitTime = {
            txDateInit,
            txTimeInit
};
Date dInitAux;
dInitAux = DateFunctions.stringToDate(arrTxInitTime);
```

A continuación, se obtiene la hora de inicio de la cortinilla que se añade delante de vídeo que se está editando. Se distinguen dos casos excluyentes: por un lado, si se trata del primer vídeo de la tabla, la hora que se toma es el ***DTo*** del vídeo que se representa en el campo de la siguiente reproducción; en otro caso, se lee el nombre del vídeo anterior y se consulta su ***DTo***.

```
Date dBeforeAux;
//Si es el primer video de la escaleta
if(iIndexEdit == 0)
{
      String txNextTs;
      txNextTs = WindowCreateSchduledTS.jtfTsNext.getText();
      TransportStream2 ts2NextTs;
      ts2NextTs = CreateScheduledTS.getByName(UploadTS.hsTS,
            txNextTs);
      dBeforeAux = ts2NextTs.getDTo();
```

```
}
else
{
      String txBeforeTS;
      txBeforeTS =
            WindowCreateSchduledTS.dtmScheduler.getValueAt
            (iIndexEdit - 1, 0).toString();
      TransportStream2 ts2Before;
      ts2Before = CreateScheduledTS.getByName(UploadTS.hsTS,
            txBeforeTS);
      dBeforeAux = ts2Before.getDTo();
}
String[] arrTxBeforeAux;
arrTxBeforeAux = DateFunctions.dateToString(dBeforeAux);
String txDateFromDefVideo = arrTxBeforeAux[0];
String txTimeFromDefVideo = arrTxBeforeAux[1];
```

Esta hora se guarda en dos variables ***String*** que se usarán más adelante para completar la fila de la cortinilla que se va a añadir.

Se ha finalizado la parte de recolección de datos y se pasa a la edición del vídeo propiamente dicha. Se comprueba que la nueva hora de inicio es mayor que la antigua, ya que la hora de inicio solo se puede atrasar, no está permitido que se adelante el inicio de un vídeo.

```
if(dBeforeAux.before(dInitAux))
```

Una vez que se comprueba que la hora de inicio es correcta, se inserta la fila correspondiente a la cortinilla.

```
String[] arrTxDefVideo = {
            txNameDefVideo,
            txDateFromDefVideo,
            txTimeFromDefVideo,
            txDateInit,
            txTimeInit
};
WindowCreateSchduledTS.dtmScheduler.insertRow(iIndexEdit,
      arrTxDefVideo);
```

A continuación, se crea un *array* de ***String*** con los nombres de todos los vídeos que se tienen en este momento en la tabla de reproducción. Este *array* se utiliza para refrescar la cola de reproducción una vez se ha editado el vídeo.

```
arrTxTsSelec = new
      String[WindowCreateSchduledTS.jtScheduler.getRowCount()];
int j=0;
for (int i=0; i<arrTxTsSelec.length; i++)
{
      String txNameTsAux;
      txNameTsAux =
            WindowCreateSchduledTS.dtmScheduler.getValueAt
            (i, 0).toString();
      arrTxTsSelec[j] = txNameTsAux;
      j++;
}
```

Ahora se refresca la escaleta a partir del vídeo que se está editando, las posiciones anteriores a él se dejan como estaban. El proceso que se sigue para el refresco es muy similar al descrito anteriormente en el método ***updateScheduler()***.

```
int iLengthSelect = j;
if(iLengthSelect != 0)
{
      String txTsName;
      Date dFromAux;
      Date dToAux;
      dFromAux = dInitAux;
      for(int i=iIndexEdit + 1; i<iLengthSelect; i++)
      {
            txTsName = arrTxTsSelec[i];
            if (txTsName != txNameDefVideo)
            {
                  TransportStream2 ts2;
                  ts2 = CreateScheduledTS.getByName
                        (UploadTS.hsTS,txTsName);
                  if(ts2 != null)
                  {
                        dToAux = new Date (dFromAux.getTime() +
                        ts2.getLDuration());
                        ts2.setDFrom(dFromAux);
                        ts2.setDTo(dToAux);
                        dFromAux = dToAux;
                  }

                  CreateScheduledTS.editTS(UploadTS.hsTS, ts2);
                  String[] arrTxRowSchedulerData =
                        SchedulerTS.rowScheduler(ts2);
                  WindowCreateSchduledTS.dtmScheduler.setValueAt
                        (arrTxRowSchedulerData[1], i, 1);
                  WindowCreateSchduledTS.dtmScheduler.setValueAt
                        (arrTxRowSchedulerData[2], i, 2);
                  WindowCreateSchduledTS.dtmScheduler.setValueAt
                        (arrTxRowSchedulerData[3], i, 3);
                  WindowCreateSchduledTS.dtmScheduler.setValueAt
                        (arrTxRowSchedulerData[4], i, 4);
            }
            else
```

```
            {
                arrIntIndexDefVideo[k] = i;
                k++;
            }
        }
}
```

Como se puede ver, se le suma 1 al índice en la declaración del bucle *for*, esto se debe a que ya se ha añadido la fila que representa la cortinilla a la tabla de reproducción, y como se quiere comenzar a refrescar desde la fila correspondiente al vídeo es necesario actualizar el índice, ya que la posición que indica el valor actual del índice corresponde a la fila de la cortinilla y no a la del vídeo.

A continuación, se actualiza también la hora de fin de la reproducción ya que cuando se retrasa la hora de inicio de un vídeo, el fin de la reproducción se ve afectado por un retraso igual al que se le ha aplicado al vídeo.

```
String txLastTS;
txLastTS = arrTxTsSelec[iLengthSelect - 1];
TransportStream2 ts2Last;
ts2Last = CreateScheduledTS.getByName(UploadTS.hsTS, txLastTS);
Date dEnd;
dEnd = ts2Last.getDTo();
CreateScheduledTS.editTS(UploadTS.hsTS, ts2Last);
WindowCreateSchduledTS.setEndTime(dEnd);
```

Igual que se ha hecho en otras ocasiones, para actualizar el campo de fin de la reproducción se utiliza el método ***setEndTime()*** de la clase ***WindowCreateSchedulerTS***.

Por último, se refresca la cola de reproducción y se borran las filas de cortinilla que se encuentran por debajo del vídeo que se ha editado.

```
CreateScheduledTS.qTsScheduled.clear();
for (int i=0; i<iLengthSelect; i++)
{
      String txNameAux;
      txNameAux = arrTxTsSelec[i];
      if (txNameAux != txNameDefVideo)
      {
            CreateScheduledTS.addElementQueue(txNameAux);
      }
}
int c = 0;
for(int i=0; i<k; i++)
{
      WindowCreateSchduledTS.dtmScheduler.removeRow
            (arrIntIndexDefVideo[i] - c);
      c++;
}
```

1.1.8.- Borrar un vídeo de la escaleta

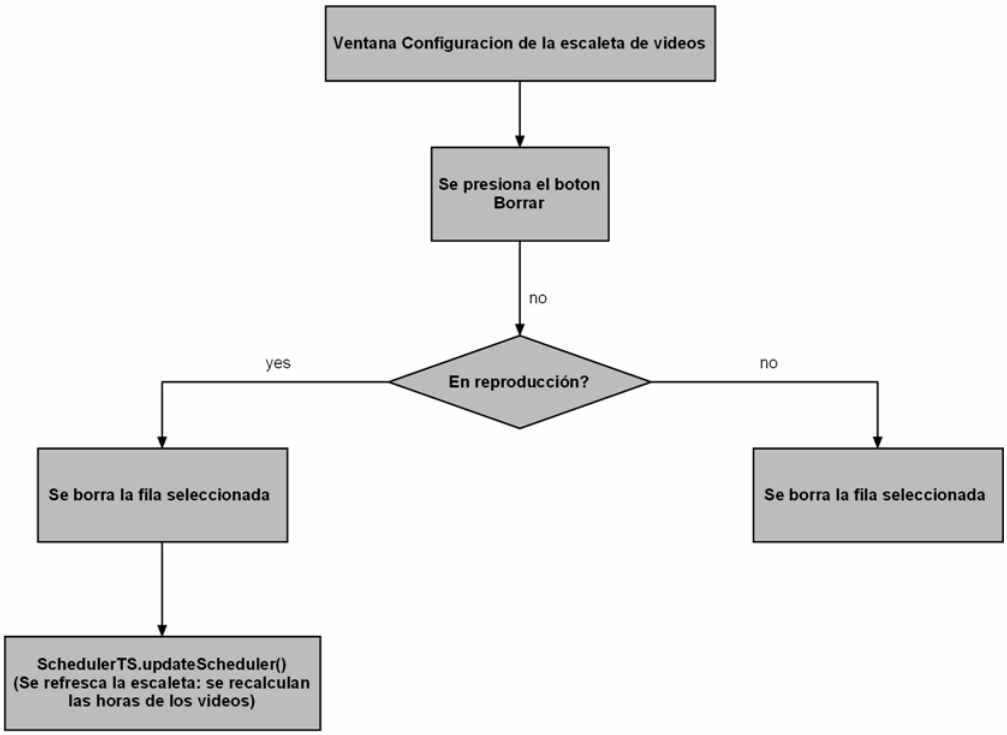


Diagrama 1.18.- *Diagrama de flujo de la acción Borrar un vídeo de la escaleta*

Otra de las funcionalidades que se implementan en la aplicación, es la de borrar vídeos que se han añadido a la escaleta. Para ello se ejecuta el método ***removeTs()*** de la clase ***SchedulerTS***. Como se puede ver en el **Diagrama 1.18** se tienen dos casos excluyentes, que la escaleta esté en reproducción o que esté parada.

```
if (arg0.getSource().equals(jbRemove))
{
	int iIndexRem;

	iIndexRem = jtScheduler.getSelectedRow();
	SchedulerTS.removeTs(iIndexRem);
}
```

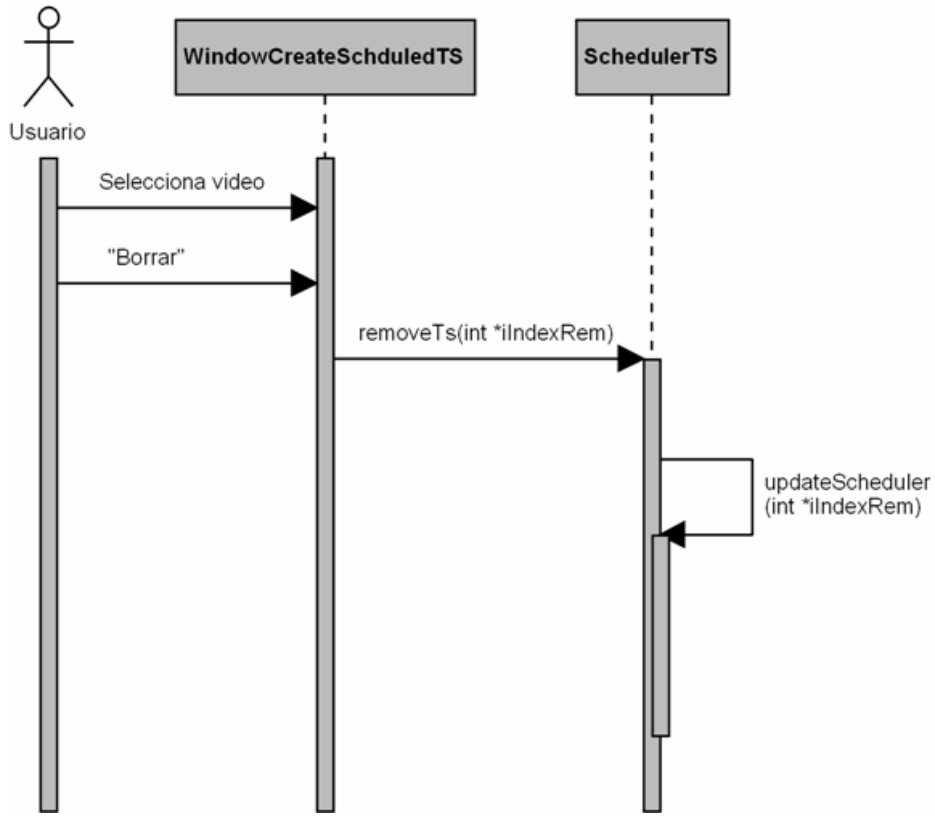


Diagrama 1.19.- *Diagrama de secuencia de la acción Borrar un vídeo de la escaleta*

En el **Diagrama 1.19** se representa el caso en que la escaleta está en reproducción, el caso en que la escaleta está parada sería igual pero sin el último paso en el que se llama al método ***updateScheduler()***.

Método: ***void removeTs(int *iIndexTS)***

- *Parámetros de entrada:*
  int *iIndexTS → Índice del vídeo que se quiere borrar
- *Salida:* ***void***

Este método borra el vídeo seleccionado y actualiza la escaleta tras el cambio.

La estructura de este método es muy similar a la utilizada en los métodos ***upTs()*** y ***downTs()*** (que se utilizan para subir y bajar vídeos por la escaleta respectivamente). Igual que en los casos comentados antes, se tienen dos casos claramente diferenciados, en un caso la escaleta no estaría en reproducción y en el otro si.

El primero es el caso más sencillo, ya que sólo es necesario borrar la línea de la tabla correspondiente al vídeo seleccionado, dado que todavía no se han configurado las horas de la escaleta. Por el contrario, si ésta se encuentra en reproducción, además de borrar la fila como en el caso anterior, se debe actualizar la configuración de la escaleta.

```
//Si la escaleta no esta en reproduccion
if(WindowCreateSchduledTS.jbSave.isEnabled())
{
      WindowCreateSchduledTS.dtmScheduler.removeRow(iIndexTS);
}
//Si la escaleta esta en reproduccion
else
{
      int iIndexRem;
      iIndexRem = iIndexTS;

      WindowCreateSchduledTS.dtmScheduler.removeRow(iIndexRem);
      WindowCreateSchduledTS.jtScheduler = new
            JTable(WindowCreateSchduledTS.dtmScheduler);
      updateScheduler(iIndexRem);
}
```

1.1.9.- Iniciar la reproducción

Antes de accionar el Play se añaden los vídeos que se quieren reproducir y se ordenan con ayuda de las funciones subir y bajar. Después se introduce la hora de inicio en los campos dedicados a ello. Una vez que se tiene listo el orden de reproducción de los vídeos y la hora de inicio, se presiona el botón Play para que comience la reproducción.

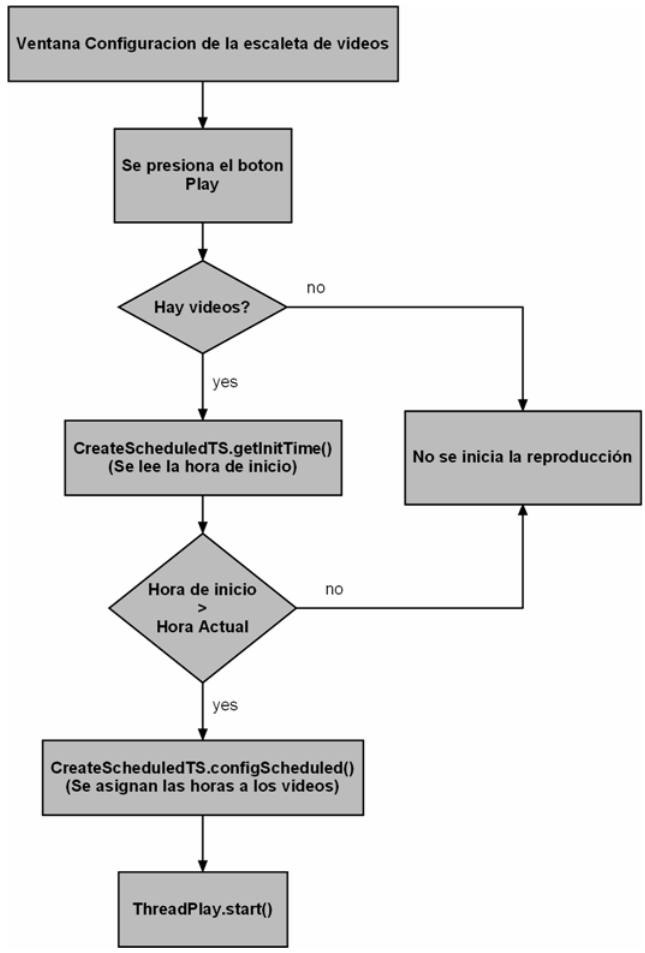


Diagrama 1.20.- *Diagrama de flujo de la acción Iniciar la reproducción*

Como se puede ver en el **Diagrama 1.20**, lo primero que se hace es comprobar que hay vídeos en la tabla, ya que es necesario tener al menos un vídeo en la tabla para comenzar la reproducción.

```
if (WindowCreateSchduledTS.dtmScheduler.getRowCount()!= 0)
{
     […]
}
else
{
      System.out.println("No hay videos para reproducir.");
      WindowCreateSchduledTS.jbSave.setEnabled(true);
      WindowCreateSchduledTS.jbEdit.setEnabled(false);
      WindowCreateSchduledTS.jbStop.setEnabled(false);
}
```

Si no hay vídeos en la tabla, se saca un mensaje de *log* y se dejan los botones como estaban, es decir, el botón del Play habilitado y los botones de Stop y Edición deshabilitados.

Tras esta comprobación, se lee la hora de inicio de los campos y se compara con la hora actual. La hora de inicio debe ser mayor que la hora actual, de lo contrario no se iniciaría la reproducción nunca. Para tomar la hora de inicio de los campos de texto se utiliza el método ***getInitTime()*** de la clase ***CreateScheduledTS***.

```
Date dTime = new Date();
Date dInitAux;

dInitAux = CreateScheduledTS.getInitTime();

if (dTime.before(dInitAux))
{
      […]
}
else
{
      System.out.println("La hora de inicio es erronea.");
      WindowCreateSchduledTS.jbSave.setEnabled(true);
      WindowCreateSchduledTS.jbEdit.setEnabled(false);
      WindowCreateSchduledTS.jbStop.setEnabled(false);
}
```

Si se cumplen ambas condiciones se configura la escaleta, es decir, se asignan las horas de inicio y fin de reproducción de cada vídeo. Estas horas se calculan a partir de la hora de inicio. Para realizar la configuración se llama al método ***configScheduled()*** de la clase ***CreateScheduledTS***.

```
arrTxTsSelec = new String[jtScheduler.getRowCount()];
for (int i=0; i<arrTxTsSelec.length; i++)
{
      arrTxTsSelec[i] = dtmScheduler.getValueAt(i, 0).toString();
}
CreateScheduledTS csts = new CreateScheduledTS();
csts.configScheduled();
```

Antes de llamar al método se crea un *array* con la lista de vídeos que hay en la tabla de reproducción, este *array* se usará de apoyo para configurar las horas de los vídeos.

Una vez que se tienen todas las horas asignadas, se deshabilita el botón de Play, y se habilitan el de Stop y el de Edición. Estos dos botones no se han puesto en activo hasta este momento porque, en primer lugar no tiene sentido poder parar la reproducción si esta no se

está ejecutándose, y por otro lado, no se puede editar la hora de inicio de un vídeo si no se han calculado todavía.

```
jbSave.setEnabled(false);
jbEdit.setEnabled(true);
jbStop.setEnabled(true);
```

La reproducción de los vídeos se lleva a cabo en un hilo independiente de la ejecución de la aplicación principal. Esto se hace así para que se pueda seguir trabajando con el resto de funcionalidades de la aplicación mientras se están reproduciendo los vídeos. De modo que sólo nos queda lanzar el hilo de reproducción para completar las acciones de inicio de la reproducción.

```
//Se instancia un nuevo objeto Java
thPlay = new Thread( new ThreadPlay(),"thPlay" );

//Se arranca el hilo, para que comience su ejecución
thPlay.start();
```

Método: ***Date getInitTime()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***Date*** → Objeto ***Date*** que representa la hora de inicio

Este método se encarga de leer la hora de inicio de los campos de texto y la devuelve en un objeto ***Date***.

Lo primero que se hace es coger el valor de los campos de texto y guardarlos en dos variables de tipo ***String***.

```
String txInitDate;
txInitDate = WindowCreateSchduledTS.jtfInitDate.getText();

String txInitTime;
txInitTime = WindowCreateSchduledTS.jtfInitTime.getText();
```

Estas dos variables se utilizan para crear un *array* que se pasa como parámetro en el método ***stringToDate()***, el cual se encarga de trasformar las dos variables de tipo ***String*** en un objeto de tipo ***Date***.

```
String[] arrTxInitTime = {
            txInitDate,
            txInitTime
};
SchedulerTS.dInit = DateFunctions.stringToDate(arrTxInitTime);
return SchedulerTS.dInit;
```

Además de devolver un objeto ***Date*** con la hora, este valor queda almacenado en la variable estática ***dInit*** que se encuentra definida en la clase ***SchedulerTS***, de manera que también se puede consultar directamente leyendo la variable.

Método: ***void configScheduled()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

Este método se encarga de calcular las horas de inicio y fin de los vídeos seleccionados para ser reproducidos. Una vez se han calculado estas horas, se actualizan los objetos ***TransportStream2*** que se encuentran en el *Set* y se rellena la tabla de reproducción con ellas.

En la primera línea del método se actualiza el valor de la variable estática ***dInit*** definida en la clase ***SchedulerTS***, para ello se ejecuta el método ***getInitTime()*** que se acaba de describir.

```
getInitTime();
```

Ya en el cuerpo del método se comprueba que la tabla de reproducción no esté vacía, si es así se definen dos variables auxiliares de tipo ***Date*** (***dFromAux*** y ***dToAux***) que se van a usar de apoyo a la hora de asignar las nuevas horas calculadas a los vídeos. Se inicializa ***dFromAux*** a la hora de inicio, ya que este será el valor del ***dFrom*** del primer vídeo.

En la parte recursiva del método se utiliza un bucle *for* para tratar todos los vídeos de la tabla. En cada iteración se siguen los siguientes pasos: primero se recupera el vídeo del ***Set*** con el método ***getByName()*** y se le asignan las horas de inicio y fin que se tienen guardadas en las variables auxiliares (el ***DTo*** se obtiene a partir del ***DFrom*** y la duración del vídeo, y el ***DFrom*** se toma del ***DTo*** del vídeo anterior); a continuación se añade el vídeo a la cola de reproducción, para ello se utiliza el método ***addElemenetQueue()*** de la

clase ***CreateScheduledTS***, como se trata de una cola FIFO los vídeos se reproducen en el orden en que se añaden; a continuación se rellenan las celdas de la tabla con las horas calculadas.

```
arrTxScheduled = WindowCreateSchduledTS.arrTxTsSelec;
if(arrTxScheduled.length != 0)
{
      String txTsName;
      Date dFromAux;
      Date dToAux;
      dFromAux = SchedulerTS.dInit;
      for(int i=0; i<arrTxScheduled.length; i++)
      {
            txTsName = arrTxScheduled[i];
            TransportStream2 ts2;
            ts2 = getByName(UploadTS.hsTS, txTsName);
            if(ts2 != null)
            {
                  dToAux = new Date (dFromAux.getTime() +
                        ts2.getLDuration());
                  ts2.setDFrom(dFromAux);
                  ts2.setDTo(dToAux);
                  dFromAux = dToAux;

                  addElementQueue(txTsName);

                  String[] arrTxRowSchedulerData =
                        ShedulerTS.rowScheduler(ts2);

                  WindowCreateSchduledTS.dtmScheduler.setValueAt
                        (arrTxRowSchedulerData[1], i, 1);
                  WindowCreateSchduledTS.dtmScheduler.setValueAt
                        (arrTxRowSchedulerData[2], i, 2);
                  WindowCreateSchduledTS.dtmScheduler.setValueAt
                        (arrTxRowSchedulerData[3], i, 3);
                  WindowCreateSchduledTS.dtmScheduler.setValueAt
                        (arrTxRowSchedulerData[4], i, 4);
            }
      }
      […]
}
```

Por último, se actualiza la hora de fin de la reproducción con el método ***setEndTime()*** al que se le pasa como parámetro de entrada el ***DTo*** del último vídeo.

```
String txLastTS;
txLastTS = arrTxScheduled[arrTxScheduled.length - 1];
TransportStream2 ts2Last;
ts2Last = getByName(UploadTS.hsTS, txLastTS);
Date dEnd;
dEnd = ts2Last.getDTo();
WindowCreateSchduledTS.setEndTime(dEnd);
```

Método: ***run() → ThreadPlay***

Como ya se ha comentado, este hilo se encarga de la reproducción de los vídeos de la escaleta. Se ha implementado este hilo para darle independencia a esta funcionalidad, ya que se quiere seguir trabajando con la aplicación durante la reproducción. Interesa, por ejemplo, poder modificar la escaleta o añadir vídeos nuevos mientras otros se están emitiendo.

Para comprender esta forma de realizar la reproducción es necesario tener en cuenta que debido a las características de laboratorio, los cambios realizados en la configuración no se hacen efectivos hasta que no se activa la nueva configuración. Por esa razón, y con la intención de reducir al máximo el *blackout* entre vídeos se realizan los cambios mientras el vídeo está en reproducción, de esta manera en el momento que termine la reproducción del vídeo actual el siguiente ya está cargado y sólo es necesario activar la nueva configuración para que comience a reproducirse.

A continuación (**Diagrama 1.21**) se incluye el diagrama de flujo correspondiente al hilo de reproducción.

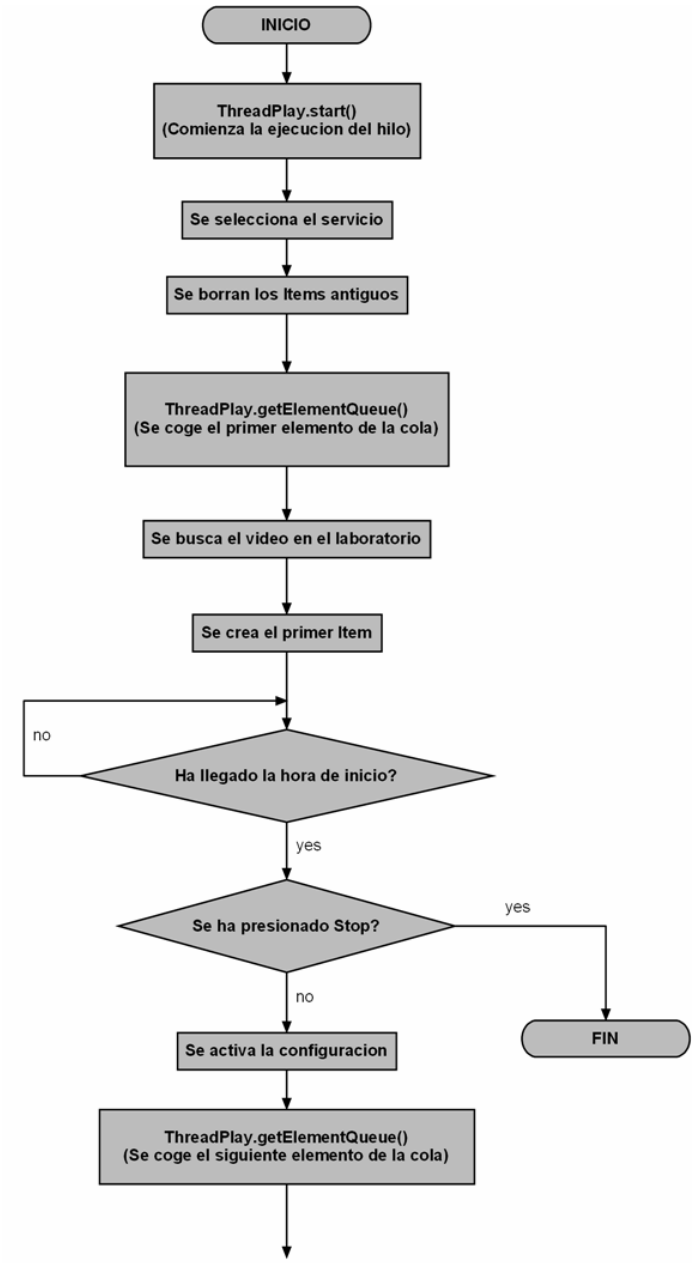


Diagrama 1.21.- *Diagrama de flujo del hilo de reproducción*

En primer lugar, se selecciona el servicio sobre el que se quiere trabajar. Para ello se utiliza el método ***getByName()*** de la clase ***Service*** que se proporciona en el paquete de la *API* del iMux.

```
Service[] arrService;
arrService = Service.getByName(UploadTS.imcConn, txNameService);

Service sService = null;
if(arrService.length == 0)
{
      System.out.println("No se ha encontrado ningun Servicio con
            ese nombre.");
}
if(arrService.length > 1)
{
      System.out.println("Advertencia: Hay mas de un Servicio con
            ese nombre.");
      sService = arrService[0];
}
if(arrService.length ==1)
{
      sService = arrService[0];
}
```

Una vez que se tiene el servicio, se borran todos los *Items* que tenga el servicio para evitar que interfieran en la reproducción. Estos *Items* pueden haberse quedado guardados de reproducciones anteriores que no se finalizaron correctamente y podrían dar problemas ya que al activar la nueva configuración se solapan con los *Items* que se acaban de crear impidiendo que se visualicen los vídeos que se están programando.

```
if (sService != null)
{
      ServiceTStreamItem[] arrSTSItem;
      arrSTSItem = sService.getServiceTStreamItems();
      if (arrSTSItem != null)
      {
            if (arrSTSItem.length != 0)
            {
                        for (int i=0; i<arrSTSItem.length; i++)
                                                                  {

      arrSTSItem[i].delete();
                                                                  }


            }
      }
      else
      {
            System.out.println("No hay ningun Service Schedule Item.");
      }
}
```

Ahora ya se puede comenzar con la reproducción, se coge el primer elemento de la cola de reproducción con el método ***getElementQueue()*** y se guarda en la variable ***ts2PlayNow*** que es la variable que se encarga de guardar el vídeo que se encuentra en reproducción. Además, se escribe el nombre del primer vídeo en el campo de texto correspondiente a la siguiente reproducción mientras se espera a que llegue la hora de inicio. También se borra la primera fila de la tabla, puesto que ya se ha tomado ese vídeo.

```
ts2PlayNow = getElementQueue();
WindowCreateSchduledTS.dtmScheduler.removeRow(0);
WindowCreateSchduledTS.jtfTsNext.setText(ts2PlayNow.getTxName());
```

A partir del nombre se busca en el laboratorio el vídeo que se va a reproducir.

```
TStream[] arrTsTstream = TStream.getByName(UploadTS.imcConn,
      ts2PlayNow.getTxName());
TStream tsTstream = null;
if(arrTsTstream.length == 0)
{
      System.out.println("No se ha encontrado ningun TS con ese
            nombre.");
}
if(arrTsTstream.length > 1)
{
      System.out.println("Advertencia: Hay mas de un TS con ese
            nombre.");
      tsTstream = arrTsTstream[0];
}
if(arrTsTstream.length == 1)
{
      tsTstream = arrTsTstream[0];
}
```

Con el servicio y el ***TStream*** se crea el primer *Item* y se habilita.

```
ServiceTStreamItem stsiTsItem =
      sService.createServiceTStreamItem(tsTstream);
stsiTsItem.setEnabled(true);
```

Se espera a que llegue la hora de inicio para activar la nueva configuración.

```
while(lTime < ts2PlayNow.getDFrom().getTime())
{
      dTime = new Date();
      lTime = dTime.getTime();
}
scActiv.activate(true);
```

Una vez que se ha activado el primer vídeo, se toma el siguiente elemento de la cola de reproducción, se borra la fila de la tabla y se actualizan los campos de reproducción en curso y siguiente reproducción. El primer vídeo pasa ahora al campo de reproducción en curso y el que se acaba de leer se coloca en el campo de siguiente reproducción.

```
ts2PlayNext = getElementQueue();

WindowCreateSchduledTS.dtmScheduler.removeRow(0);

WindowCreateSchduledTS.jtfTsPlay.setText(ts2PlayNow.getTxName());
WindowCreateSchduledTS.jtfTsNext.setText(ts2PlayNext.getTxName());
```

A partir de este punto se comienza con la parte recursiva del hilo, la condición que se impone para continuar con la reproducción es que la cola no esté vacía.

Si es así, se borra el *Item* que está en reproducción, y se comprueba si entre el vídeo en reproducción y el siguiente hay cortinilla. Para realizar esta comprobación se compara el ***DTo*** del vídeo en reproducción y el ***DFrom*** del siguiente vídeo, si son iguales no hay cortinilla. Si esto es así, se guarda en la variable ***ts2PlayNow*** el siguiente vídeo que se guarda en la variable ***ts2PlayNext***. Si por el contrario, hay cortinilla entre ellos se le asigna a la variable ***ts2PlayNow*** el vídeo por defecto correspondiente a la cortinilla y se le dan las horas de inicio y fin de reproducción que son el ***DTo*** del vídeo en reproducción y el ***DFrom*** del siguiente vídeo.

Una vez que ya se sabe cuál va a ser el siguiente vídeo en reproducirse, se lee del laboratorio y se guarda en una variable ***TStream*** (que es una variable propia de la *API* del iMux).

Igual que se ha hecho en la inicialización, a partir del servicio y de la variable ***TStream*** se crea el nuevo *Item* con el método ***createServiceTStreamItem***.

Ya se tiene la nueva configuración creada, de modo que se espera a que termine la reproducción actual para activarla. Tras la activación se borra la primera fila de la tabla y se actualizan los campos de texto que nos informan de la reproducción actual y de la siguiente. Para realizar esta actualización de los campos se necesita saber si la reproducción en curso se trata de una cortinilla o de un vídeo de la escaleta. Si se trata de una cortinilla se guarda el texto “cortinilla” en el campo que informa de la reproducción en curso, si no es así se lee el siguiente vídeo de la cola de reproducción con el método ***getElementQueue()*** y se guarda el resultado en la variable ***ts2PlayNext***. Por último se rellenan los campos con los nombre de los correspondientes vídeos.

```
while(!CreateScheduledTS.qTsScheduled.isEmpty())
{
	stsiTsItem.setEnabled(false);
	stsiTsItem.delete();

	//Si no hay cortinilla entre los videos

	if(ts2PlayNow.getDTo().equals(ts2PlayNext.getDFrom()))
	{
		ts2PlayNow = ts2PlayNext;
	}
	else
	{
		Date dFromDefault = ts2PlayNow.getDTo();
		Date dToDefault = ts2PlayNext.getDFrom();
		ts2PlayNow = CreateScheduledTS.getByName
			(UploadTS.hsTS, CreateScheduledTS.txNameDefVideo);
		ts2PlayNow.setDFrom(dFromDefault);
		ts2PlayNow.setDTo(dToDefault);
	}
	txNameTs = ts2PlayNow.getTxName();

	arrTsTstream = TStream.getByName(UploadTS.imcConn,txNameTs);
	tsTstream = null;
	if(arrTsTstream.length == 0)
	{
		System.out.println("No se ha encontrado
			ningun TS con ese nombre.");
	}
	if(arrTsTstream.length > 1)
	{
		System.out.println("Advertencia: Hay mas de
			un TS con ese nombre.");
		tsTstream = arrTsTstream[0];
	}
	if(arrTsTstream.length == 1)
	{
		tsTstream = arrTsTstream[0];
	}
	stsiTsItem =
		sService.createServiceTStreamItem(tsTstream);
	stsiTsItem.setEnabled(true);
	dTime = new Date();
	lTime = dTime.getTime();
	while(lTime < ts2PlayNow.getDFrom().getTime())
	{
		dTime = new Date();
		lTime = dTime.getTime();
	}
	scActiv.activate(true);

	WindowCreateSchduledTS.dtmScheduler.removeRow(0);

	//Si el video en reproduccion no es la cortinilla
	if(!ts2PlayNow.getTxName().equals(CreateScheduledTS.
		txNameDefVideo))
	{
```

```
                ts2PlayNext = getElementQueue();
                WindowCreateSchduledTS.jtfTsPlay.setText
                      (ts2PlayNow.getTxName());
                WindowCreateSchduledTS.jtfTsNext.setText
                      (ts2PlayNext.getTxName());
          }
          else
          {
                WindowCreateSchduledTS.jtfTsPlay.setText
                      (SchedulerTS.txNameDefVideo);
          }
          […]
}
```

A la salida del bucle se debe repetir una iteración pero sin tomar datos de la cola, ya que ésta está vacía. Si no se hace esto, el vídeo que está en la variable ***ts2PlayNext*** no se procesa.

```
stsiTsItem.setEnabled(false);
stsiTsItem.delete();

if(ts2PlayNow.getDTo().equals(ts2PlayNext.getDFrom()))
{
      ts2PlayNow = ts2PlayNext;
}
else
{
      Date dFromDefault = ts2PlayNow.getDTo();
      Date dToDefault = ts2PlayNext.getDFrom();
      ts2PlayNow = CreateScheduledTS.getByName(UploadTS.hsTS,
            CreateScheduledTS.txNameDefVideo);
      ts2PlayNow.setDFrom(dFromDefault);
      ts2PlayNow.setDTo(dToDefault);
}
txNameTs = ts2PlayNow.getTxName();

arrTsTstream = TStream.getByName(UploadTS.imcConn, txNameTs);
tsTstream = null;
if(arrTsTstream.length == 0)
{
      System.out.println("No se ha encontrado ningun TS con ese
            nombre.");
}
if(arrTsTstream.length > 1)
{
      System.out.println("Advertencia: Hay mas de un TS con ese
            nombre.");
      tsTstream = arrTsTstream[0];
}
if(arrTsTstream.length == 1)
{
      tsTstream = arrTsTstream[0];
}
```

```
stsiTsItem = sService.createServiceTStreamItem(tsTstream);
stsiTsItem.setEnabled(true);
dTime = new Date();
lTime = dTime.getTime();
while(lTime < ts2PlayNow.getDFrom().getTime())
{
      dTime = new Date();
      lTime = dTime.getTime();
}

scActiv.activate(true);
if(ts2PlayNow.getTxName().equals(CreateScheduledTS.
            txNameDefVideo))
{
      stsiTsItem.setEnabled(false);
      stsiTsItem.delete();

      ts2PlayNow = ts2PlayNext;

      txNameTs = ts2PlayNow.getTxName();

      arrTsTstream = TStream.getByName(UploadTS.imcConn,
            txNameTs);
      tsTstream = null;
      if(arrTsTstream.length == 0)
      {
            System.out.println("No se ha encontrado ningun TS con
                  ese nombre.");
      }
      if(arrTsTstream.length > 1)
      {
            System.out.println("Advertencia: Hay mas de un TS con
                  ese nombre.");
            tsTstream = arrTsTstream[0];
      }
      if(arrTsTstream.length == 1)
      {
            tsTstream = arrTsTstream[0];
      }

      stsiTsItem = sService.createServiceTStreamItem(tsTstream);
      stsiTsItem.setEnabled(true);
      dTime = new Date();
      lTime = dTime.getTime();
      while(lTime < ts2PlayNow.getDFrom().getTime())
      {
            dTime = new Date();
            lTime = dTime.getTime();
      }

      scActiv.activate(true);
}

WindowCreateSchduledTS.jtfTsPlay.setText(ts2PlayNow.getTxName());
WindowCreateSchduledTS.jtfTsNext.setText("");
```

Por último se controla cuando finaliza la reproducción del último vídeo para borrar el *Item* y dar por terminada la reproducción. También se restaura el estado de los botones a como estaban antes de comenzar la reproducción, es decir, Stop y Editar deshabilitados y Play habilitado.

```
dTime = new Date();
lTime = dTime.getTime();
while (lTime < ts2PlayNow.getDTo().getTime())
{
      dTime = new Date();
      lTime = dTime.getTime();
}
stsiTsItem.delete();
scActiv.activate(true);
WindowCreateSchduledTS.jtfTsPlay.setText("");
WindowCreateSchduledTS.jbSave.setEnabled(true);
WindowCreateSchduledTS.jbEdit.setEnabled(false);
WindowCreateSchduledTS.jbStop.setEnabled(false);
```

Método: ***TransportStream2 getElementQueue()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***TransportStream2*** → Objeto ***TransportStream2*** que representa el siguiente vídeo que se debe reproducir

Este método lee el siguiente vídeo a reproducir de la cola de reproducción.

En esta cola se guardan los nombres de los vídeos en el orden en que se van a reproducir, por lo tanto lo primero que se hace es leer el nombre del siguiente vídeo y se guarda en la variable ***txGetElementQueue*** de tipo ***String***, para leer el siguiente elemento de la cola se usa el método ***poll()*** que se proporciona en la clase ***Queue*** de Java.

Después, con el nombre del vídeo como parámetro de entrada se llama al método ***getByName()*** de la clase ***CreateScheduledTS*** que ya se ha usado anteriormente. Este método devuelve el vídeo guardado en el ***Set*** en una variable ***TransportStream2*** que se pasa como valor de retorno en el método.

```
String txGetElementQueue;
TransportStream2 ts2GetTs;
txGetElementQueue = CreateScheduledTS.qTsScheduled.poll();
ts2GetTs = CreateScheduledTS.getByName(UploadTS.hsTS,
            txGetElementQueue);
return ts2GetTs;
```

1.1.10.- Parar la reproducción en curso

En la pantalla Configuración de la escaleta de vídeos se dispone de un botón Stop que permite parar la reproducción. Una vez que se presiona el botón la reproducción se detiene inmediatamente. Como ya se ha comentado en el apartado anterior, la reproducción se ejecuta en un hilo independiente, por lo tanto, para detener la reproducción se debe matar este hilo de reproducción. Los ***Thread*** de java tienen muchos métodos para parar un hilo: ***destroy()***, ***stop()***, ***suspend()***, … pero si se mira la *API* de java se puede ver que todos estos métodos son inseguros, están obsoletos y desaconsejados. De modo que la mejor manera de terminar el hilo de forma segura es implementar directamente en la aplicación un mecanismo para parar el hilo, lo único que tiene que hacer este mecanismo es terminar el método ***run()***.

Para implementar este mecanismo se debe considerar la forma en que se ha implementado el método ***run()*** del hilo. En este caso la prioridad es parar la reproducción en el momento en que se presione el botón. Para ello lo que se hace es borrar todos los *Item* que haya configurados en el servicio, y se activa esta nueva configuración. De esta manera el vídeo en reproducción deja de reproducirse, de hecho no se reproduce nada ya que no hay ningún vídeo habilitado.

```
Service[] arrService;
try
{
      arrService = Service.getByName(UploadTS.imcConn, txNameService);

      Service sService = null;
      if(arrService.length == 0)
      {
            System.out.println("No se ha encontrado ningun Servicio con
                  ese nombre.");
      }
      if(arrService.length > 1)
      {
            System.out.println("Advertencia: Hay mas de un Servicio con
                  ese nombre.");
            sService = arrService[0];
      }
      if(arrService.length ==1)
      {
            sService = arrService[0];
      }

      //Se eliminan todos los Items del servicio
      if (sService != null)
      {
            ServiceTStreamItem[] arrSTSItem;
```

```
            arrSTSItem = sService.getServiceTStreamItems();
            if (arrSTSItem != null)
            {
                 if (arrSTSItem.length != 0)
                 {
                      for (int i=0; i<arrSTSItem.length; i++)
                      {
                           arrSTSItem[i].delete();
                      }
                 }
            }
            else
            {
                 System.out.println("No hay ningun Service Schedule
                      Item.");
            }
      }
      scActiv.activate(true);
}
catch (IOException ioe)
{
      System.out.println("Error al buscar el servicio: " +
            ioe.getMessage());
}
```

Ya se ha parado la reproducción en curso, pero todavía no se ha terminado el hilo, de modo que si se deja así, cuando llegue la hora de inicio del siguiente vídeo este comenzará a reproducirse. Para evitar esto se ha incluido una condición en el método ***run()***, de modo que cada vez que se va a activar una configuración se comprueba si se ha presionado el botón Stop, si se ha presionado se llama a ***return*** para finalizar el método, de este modo se termina el método.

Para conseguir esto se habilita el botón Play y se deshabilitan los botones Stop y Editar.

```
jbSave.setEnabled(true);
jbEdit.setEnabled(false);
jbStop.setEnabled(false);
```

Al habilitar el botón Play se consigue que el hilo de reproducción salga del bucle *while* que gestiona la reproducción. Como ya se comento en la descripción de la ejecución del hilo, una de las condiciones de la parte iterativa del hilo es que el botón Play se encuentre deshabilitado. El hecho de que este botón esté habilitado se interpreta como que se ha parado la reproducción.

Por último se asigna a la variable ***thPlay*** de tipo ***Thread*** el valor ***null*** para evitar que el hilo se quede abierto y esto de problemas en la siguiente reproducción.

```
thPlay = null;
```

1.1.11.- Modificar el vídeo de la cortinilla

Como ya se ha comentado en apartados anteriores, al modificar la hora de inicio de algún vídeo se rellena el tiempo que queda entre el final del vídeo anterior y la nueva hora de inicio con una cortinilla. Esta cortinilla consiste en la reproducción contínua de un vídeo auxiliar para evitar que este intervalo quede sin emisión. Este vídeo es identificado por la aplicación y esta lo reproduce automáticamente sin que el usuario tenga que programarlo. Este vídeo está ya cargado en el laboratorio cuando se arranca la aplicación y puede ser modificado por el usuario, para ello en la parte inferior izquierda de la pantalla de configuración de la escaleta se encuentra el campo que permite modificar este vídeo auxiliar. Para cambiar el vídeo de la cortinilla se especifica la ruta donde se encuentra el nuevo vídeo que se quiere cargar y se presiona el botón Cargar.

Diagrama 1.22.- *Diagrama de flujo de la acción Modificar el vídeo de la cortinilla*

Una vez que se ha seleccionado el nuevo vídeo que se quiere asignar a la cortinilla se carga este vídeo presionando el botón Cargar. Al pulsar se lanza el método ***uploadDefaultVideo()*** de la clase ***CreateScheduledTS***. En el **Diagrama 1.22** se puede ver el diagrama de flujo de esta operación.

```
if (arg0.getSource().equals(jbUploadDefVideo))
{
      CreateScheduledTS.uploadDefaultVideo();
}
```

En el **Diagrama 1.23** que se incluye más abajo se representa el diagrama de secuencia con las acciones necesarias para modificar el vídeo de la cortinilla.

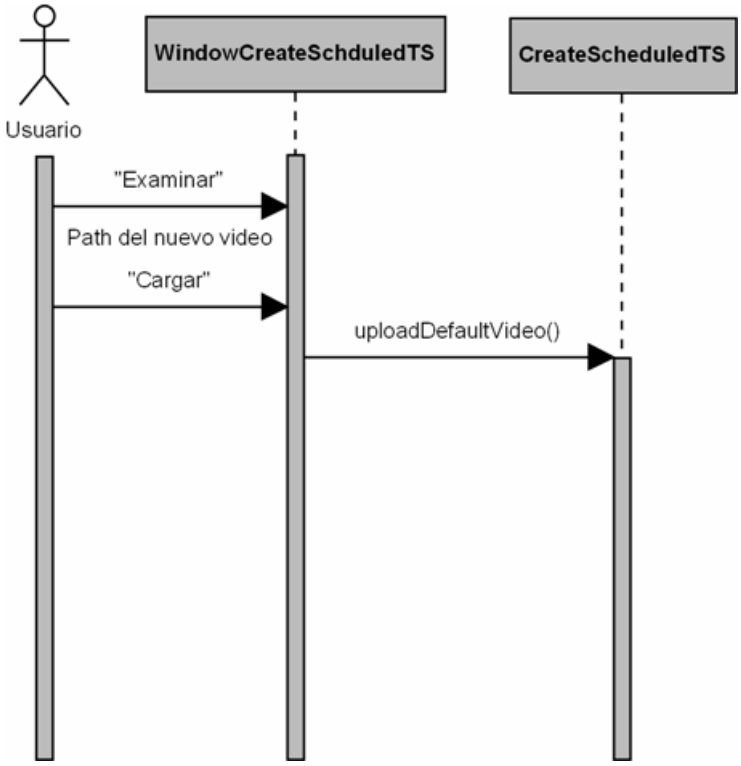


Diagrama 1.23.- *Diagrama de secuencia de la acción Modificar el vídeo de la cortinilla*

Método ***void uploadDefaultVideo()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

Este método cambia el vídeo que se reproduce a modo de cortinilla cuando se ha modificado la hora de inicio de un vídeo mientras se espera a que llegue la hora de comenzar la siguiente reproducción programada.

En primer lugar se crea un objeto ***File*** con el *path* que se ha guardado en el campo de texto que se ha configurado para ello en la ventana Configuración de la escaleta de vídeos. También se crea una variable ***TransportStream2*** en la que se guarda el tamaño del nuevo vídeo.

```
File fTs = new
      File(WindowCreateSchduledTS.jtfDefaultVideo.getText());

TransportStream2 ts2Add = new TransportStream2();

Long lFsize = new Long(fTs.length());
//Como se obtiene el tamaño en Bytes se pasa a bits
lFsize = lFsize * 8;
//Se guarda el tamaño del archivo en la variable ts2 que
//despues se guarda en el hashset
ts2Add.setDSize(lFsize);
```

A continuación se borra, tanto del laboratorio como del ***Set***, el antiguo vídeo que representaba a la cortinilla.

```
try
{
      /*
       * Se recupera en la variable ts2 el video de cortinilla
       * */
      TStream[] arrTsTstream =
      TStream.getByName(UploadTS.imcConn, txNameDefVideo);
      TStream tsTstream = null;
      if(arrTsTstream.length == 0)
      {
            System.out.println("No se ha encontrado ningun TS con
                  ese nombre.");
      }
      if(arrTsTstream.length > 1)
      {
            System.out.println("Advertencia: Hay mas de un TS con
                  ese nombre.");
            for (int i=0; i<arrTsTstream.length; i++)
            {
                  arrTsTstream[i].delete();
                  TransportStream2 ts2Def;
                  ts2Def = CreateScheduledTS.getByName
                        (UploadTS.hsTS, txNameDefVideo);
                  UploadTS.hsTS.remove(ts2Def);
            }
      }
      if(arrTsTstream.length == 1)
      {
            tsTstream = arrTsTstream[0];
            tsTstream.delete();
            TransportStream2 ts2Def;
            ts2Def = CreateScheduledTS.getByName(UploadTS.hsTS,
                  txNameDefVideo);
            UploadTS.hsTS.remove(ts2Def);
      }
[…]
}
```

Si hay más de un vídeo en el laboratorio con el mismo nombre se borran todos, para evitar problemas a la hora de reproducir la cortinilla. El laboratorio no permite filtrar dos vídeos con el mismo nombre, de manera que es posible encontrar dos vídeos con el mismo nombre aunque el contenido sea diferente. Esto puede llevar a errores a la hora de reproducir ya que se accede a los vídeos cargados en el laboratorio por el nombre, así que puede ocurrir que aunque se haya cargado un nuevo vídeo para la cortinilla se visualice otro diferente si no se borran todos los vídeos con el mismo nombre.

Ahora se crea una nueva variable ***TStreamPackets*** para cargar el vídeo en el laboratorio. A partir de esta variable se obtiene el *bitrate* del nuevo vídeo que se guarda en la variable ***ts2Add*** de tipo ***TransportStream2*** que se ha definido más arriba.

```
TStreamPackets tspPack;

[…]

//Se guarda el nombre del TS en la variable ts2
ts2Add.setTxName(txNameDefVideo);
tspPack = TStream.createTStreamPackets(UploadTS.imcConn,
      txNameDefVideo);

/*
 * Se carga el ts en el laboratorio
 * */
tspPack.update(fTs, true);
tspPack.getTStreamPacketsPIDs();
tspPack.setBitrate(tspPack.getDetectedBitrate());
```

Por último, se calcula la duración del vídeo del mismo modo que se ha descrito en funcionalidades anteriores como la de cargar un vídeo. La variable ***ts2Add*** se guarda en el ***Set*** con todos los datos que se han ido obteniendo.

```
double dBitrate = tspPack.getBitrate();
//Se guarda el bitrate en la variable ts2
ts2Add.setDBitrate(dBitrate);

double dDuration = lFsize/dBitrate;
long lDurationMili = (long) (dDuration * 1000);

ts2Add.setLDuration(lDurationMili);

/*
 * Se guarda el ts en el Set
 * */
UploadTS.hsTS.add(ts2Add);

TStreamPacketsPID[] arrTsp = tspPack.getTStreamPacketsPIDs();
if (arrTsp != null)
```

```
{
      for (int i=0; i<arrTsp.length; i++)
      {
            int iPid = arrTsp[i].getPidType();
            if (iPid != TStreamPacketsPID.PID_TYPE_AUDIO_MPEG1 &&
                iPid != TStreamPacketsPID.PID_TYPE_AUDIO_MPEG2)
            {
                 if (iPid !=
                      TStreamPacketsPID.PID_TYPE_VIDEO_MPEG1 &&
                      iPid !=
                      TStreamPacketsPID.PID_TYPE_VIDEO_MPEG2)
                 {
                      arrTsp[i].setDestinationPID(-1);
                      arrTsp[i].setPidType(TStreamPacketsPID.
                           PID_TYPE_UNREFERENCED);
                 }
            }
      }
}
```

Igual que se hacía cuando se cargaba un vídeo nuevo se eliminan las PIDs que no son necesarias, y se dejan únicamente las que corresponden al audio y al vídeo.

1.2.- Gestión de Aplicaciones

La gestión de aplicaciones es algo más sencilla que la gestión de vídeos, en primer lugar, el problema debido a la falta de precisión de segundos del laboratorio es más tolerable en este caso. Otro de los motivos es que el concepto de escaleta es muy diferente al que se considera para los vídeos. En este caso, consiste en dar accesibilidad a distintas aplicaciones en momentos diferentes y no a una reproducción continua.

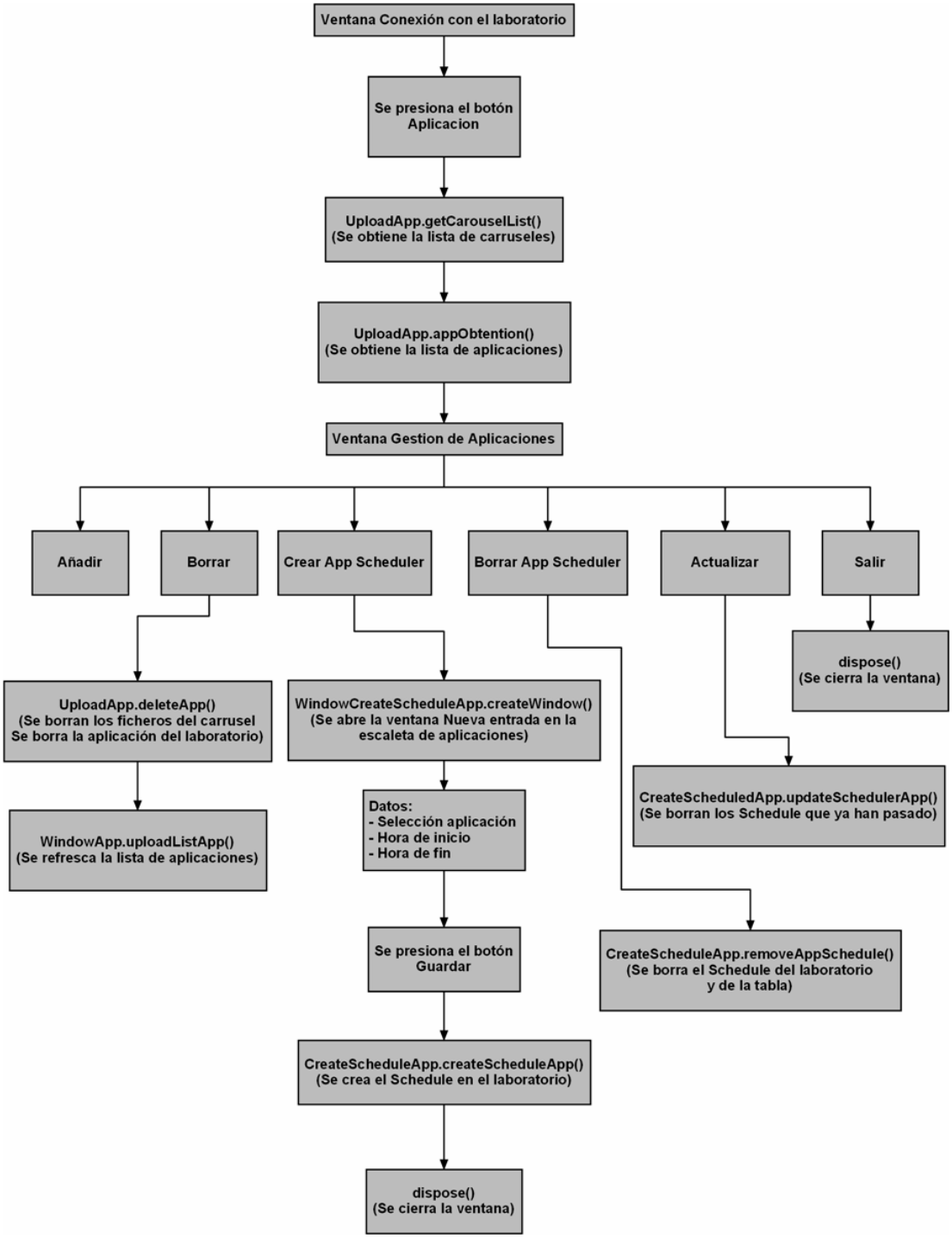


Diagrama 1.24.- *Diagrama de flujo de la gestión de aplicaciones*

En el diagrama de flujo representado arriba (**Diagrama 1.24**) se incluyen los flujos de las acciones disponibles en el bloque de gestión de aplicaciones.

Para acceder a la gestión de aplicaciones se debe presionar el botón Aplicación de la ventana de inicio. Cuando la clase ***LabConnect*** recibe el *listener* del botón se lanza la ventana Gestión de aplicaciones.

```
private WindowApp wa = new WindowApp();

[…]

else if (arg0.getSource().equals(jbApp))
{
      wa.createWindow(imcConn);
}
```

Para crear esta ventana se necesita conocer la lista de carruseles y aplicaciones disponibles en el laboratorio.

```
[…]

//SELECCION DEL CARRUSEL
/*
 * Panel con el combobox
 * */

UploadApp ulApp = new UploadApp();
String[] arrTxApp = ulApp.getCarouselList(imcConnListApp);
for (int i=0; i<arrTxApp.length; i++)
{
      jcbCarr.addItem(arrTxApp[i]);
}

[…]

//LISTA DE APLICACIONES DISPONIBLES
/*
 * Se crea la lista
 * */

arrTxApp = uaApp.appObtention(imcConnListApp);

if (dlmListApp.size() != 0)
{
      dlmListApp.clear();
}

if (arrTxApp != null)
{
      for (int i=0; i<arrTxApp.length; i++)
      {
            dlmListApp.addElement(arrTxApp[i]);
      }
}
```

En primer lugar se consulta la lista de carruseles disponibles, para ello se llama al método ***getCarouselList()*** de la clase ***UploadApp*** que devuelve un *array* de ***String*** con los nombres de todos los carruseles disponibles. Este *array* se utiliza, con ayuda de un bucle *for*, para rellenar los *Items* del ***ComboBox*** que se tiene en la ventana para seleccionar el carrusel con el que se va a trabajar.

Por otro lado, se obtienen las aplicaciones que hay guardadas en el laboratorio con ayuda del método ***appObtention()*** de la misma clase ***UploadApp***. En este caso, el método también devuelve un *array* con los nombres de las aplicaciones. Con este *array* se rellena la lista de aplicaciones disponibles de la ventana. Esta secuencia de acciones se representa en el **Diagrama 25** que se muestra más abajo.

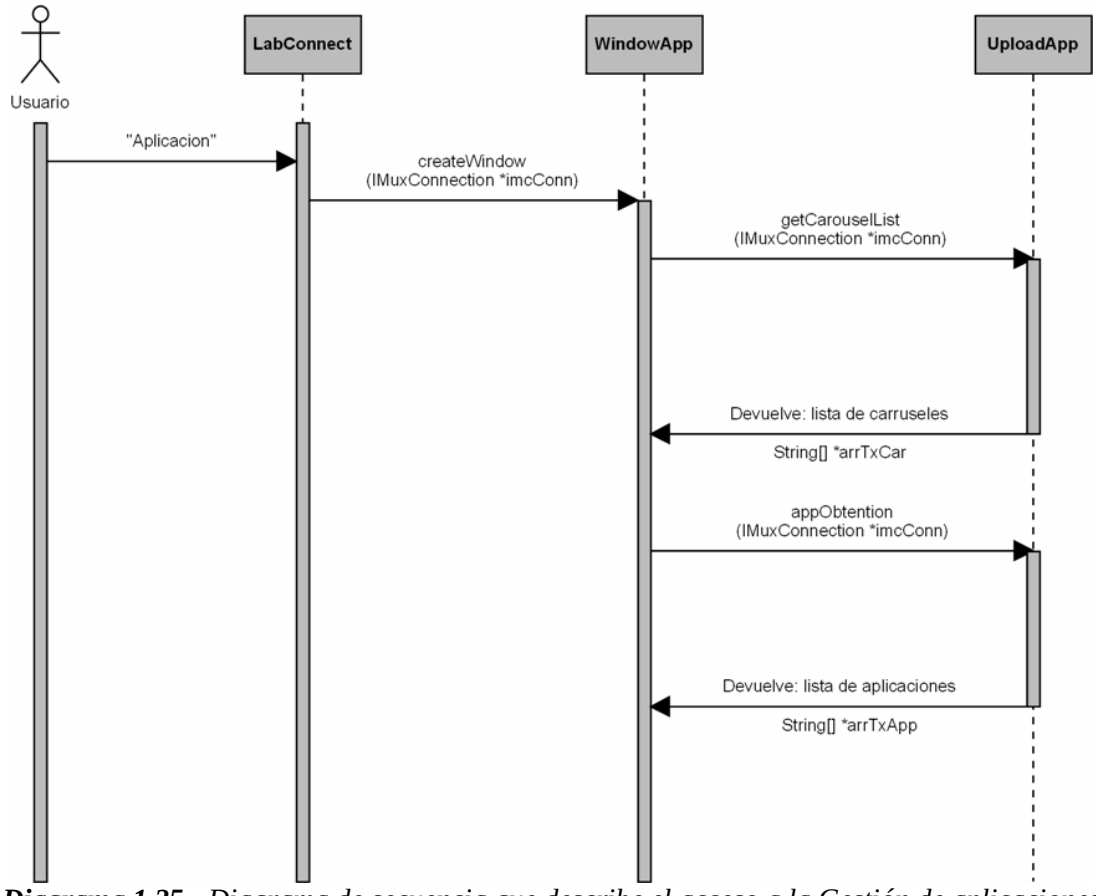


Diagrama 1.25.- *Diagrama de secuencia que describe el acceso a la Gestión de aplicaciones*

Como se puede ver en la **Figura 1.7**, esta ventana dispone de 6 botones que dan acceso a las siguientes acciones:

- Añadir aplicación MHP
- Borrar aplicación MHP
- Crear *Schedule* de aplicaciones
- Borrar *Schedule* de aplicaciones
- Actualizar escaleta de aplicaciones
- Salir de la ventana

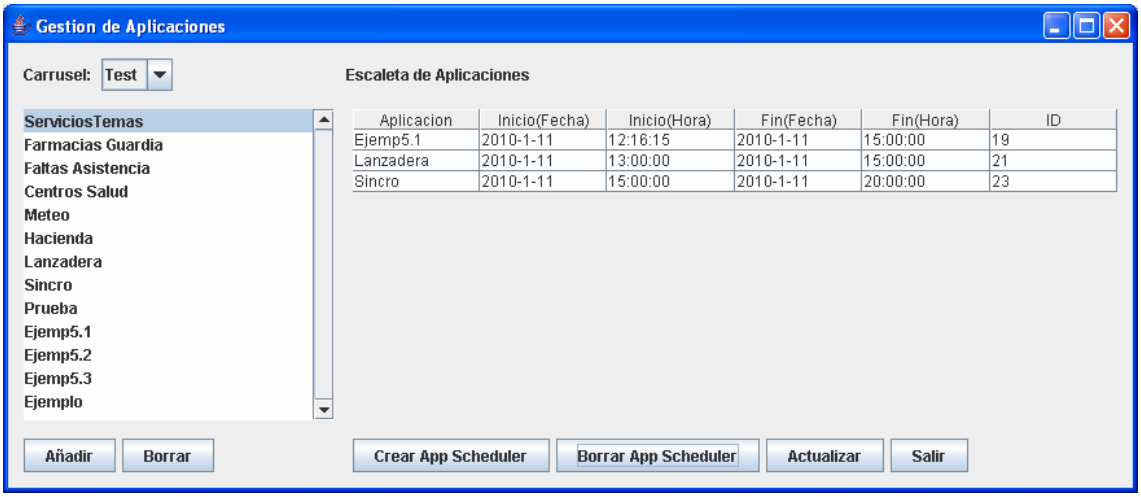


Figura 1.7.- *Ventana Gestión de aplicaciones*

Método: ***String[] getCarouselList(IMuxConnection *imcConn)***

- *Parámetros de entrada:*
  IMuxConnection *imcConn → Objeto resultante de la conexión
- *Salida:* ***String[]*** → *Array* de ***String*** con los nombres de los carruseles disponibles en el laboratorio

Este método consulta al laboratorio que carruseles hay disponibles para trabajar con ellos y devuelve el nombre de todos ellos en un *array* de ***String***.

En primer lugar se pide al laboratorio la lista de carruseles guardados en él con el método ***getAll()*** que se proporciona en la clase ***Carousel*** de la *API* del iMux. Si esta lista no está vacía se rellena el *array* de salida con ayuda de un bucle *for*.

```
private Carousel[] arrCarCarruseles;

[…]

String[] arrTxCarousel = null;

try
{
      arrCarCarruseles = Carousel.getAll(imcConn);
      if (arrCarCarruseles != null)
      {
            arrTxCarousel = new String [arrCarCarruseles.length];
            for (int i=0; i<arrCarCarruseles.length; i++)
            {
                  arrTxCarousel[i] = arrCarCarruseles[i].getName();
            }
      }
}
catch (IOException ioe)
{
      System.out.println("Excepcion de entrada-salida al intentar
            acceder a la lista de carruseles: " +
            ioe.toString());
}

return arrTxCarousel;
```

Método: ***String[] appObtention(IMuxConnection *imcConn)***

- *Parámetros de entrada:*
  IMuxConnection *imcConn → Objeto resultante de la conexión
- *Salida:* ***String[]*** → *Array* con los nombres de todas las aplicaciones disponibles

Este método obtiene una lista de todas las aplicaciones cargadas en el laboratorio.

El esquema de este método es muy similar al del método ***getCarouselList()*** que se ha descrito anteriormente. En primer lugar se obtiene la lista de aplicaciones cargadas con el método ***getAll()*** de la clase ***Application*** que se proporciona en la *API* del laboratorio. Luego se traduce esta lista en un *array* de ***String*** con los nombres de dichas aplicaciones.

```
Application[] arrApp;

try
{
      arrApp = Application.getAll(imcConn);

      if (arrApp != null)
      {
            String[] arrTxApp = new String[arrApp.length];
```

```
            for (int i=0; i<arrApp.length; i++)
            {
                  arrTxApp[i] = arrApp[i].getAITName();
            }

            return arrTxApp;
      }
      else
      {
            return null;
      }

}
catch (IOException ioe)
{
      System.out.println("Excepción de entrada-salida al
            consultar las aplicaciones: "+ioe.toString());
      return null;
}
```

1.2.1.- Añadir aplicación MHP

Esta funcionalidad permite añadir nuevas aplicaciones MHP a la lista de aplicaciones disponibles. De acuerdo con las características del laboratorio estas aplicaciones se cargan en el carrusel en forma de fichero .zip, donde se encuentran todos los archivos que componen la aplicación. La acción de crear una nueva aplicación se debe hacer en dos pasos, en primer lugar se descomprime el fichero .zip en el carrusel y una vez que se tienen todos los archivos subidos al carrusel se crea la aplicación propiamente dicha en el laboratorio.

A continuación se incluye el diagrama de flujo de la acción Añadir una aplicación MHP (**Diagrama 1.26**).

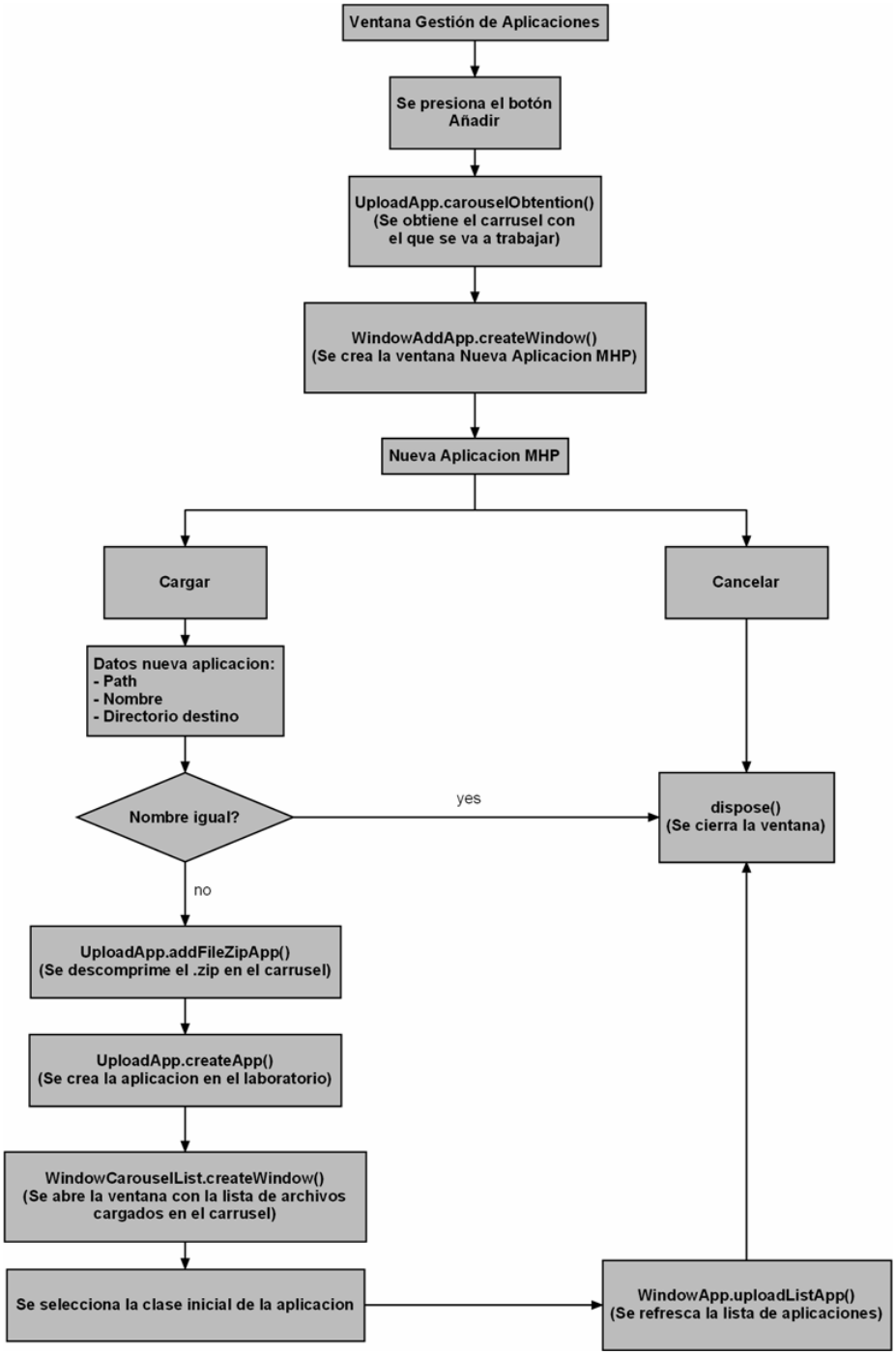


Diagrama 1.26.- *Diagrama de flujo de la acción Añadir aplicación MHP al laboratorio*

Antes de añadir la nueva aplicación se debe seleccionar el carrusel con el que se quiere trabajar. Para ello se dispone de un ***Combo Box*** en la parte superior de la pantalla, una vez seleccionado se presiona el botón Añadir que da acceso a la ventana Nueva Aplicación MHP (**Figura 1.8**) a través del método ***createWindow()*** de la clase ***WindowAddApp***. Antes de lanzar la nueva pantalla se lee el carrusel seleccionado del

Combo Box y se guarda el nombre del mismo en la variable estática ***txCarousel*** de tipo ***String***. Con este nombre como parámetro de entrada, se llama al método ***carouselObtention()*** de la clase ***UploadApp*** del que se obtiene el carrusel correspondiente del laboratorio.

```
private UploadApp ua = new UploadApp();
public static Carousel cCarousel;
public static String txCarousel;

[…]

//BOTON AÑADIR APLICACION
if (arg0.getSource().equals(jbAddApp))
{
      txCarousel = jcbCarr.getSelectedItem().toString();

      cCarousel = ua.carouselObtention(imcConnListApp,
            txCarousel);
      WindowAddApp wAddApp = new WindowAddApp();
      wAddApp.createWindow();
}
```

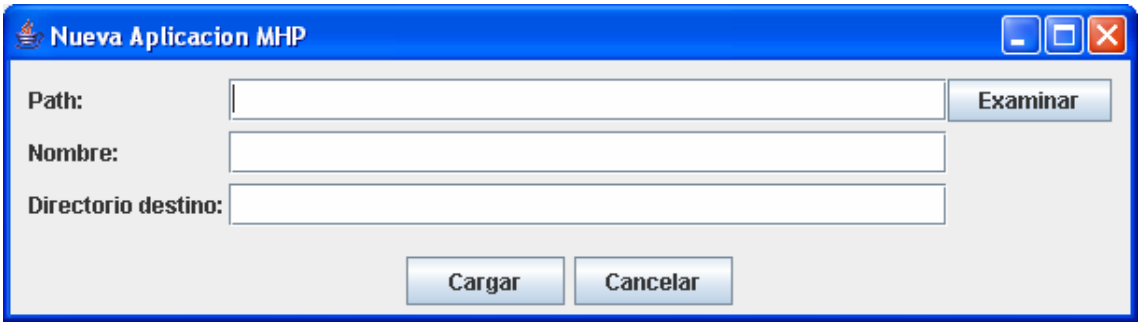


Figura 1.8.- *Ventana Nueva aplicación MHP*

Método: ***Carousel carouselObtention(IMuxConnection *imcConn, String *txCarName)***

- *Parámetros de entrada:*
  - ***IMuxConnection *imcConn*** → Objeto resultante de la conexión.
  - ***String *txCarName*** → Nombre del carrusel seleccionado.
- *Salida:* ***Carousel*** → Carrusel seleccionado

Este método busca en el laboratorio el carrusel a partir del nombre que se le da como parámetro de entrada en la variable ***txCarName***.

La adquisición del carrusel se realiza través del método ***getByName()*** de la clase ***Carousel*** que se proporciona en la *API* del laboratorio.

```
Carousel[] arrCarCarousels = null;
Carousel cCarousel = null;

try
{
      arrCarCarousels = Carousel.getByName(imcConn, txCarName);
      if(arrCarCarousels.length == 0)
      {
            System.out.println("No se ha encontrado ningun      carrusel
                  con ese nombre.");
      }
      if(arrCarCarousels.length > 1)
      {
            System.out.println("Advertencia: Hay mas de un      carrusel
                  con ese nombre.");
            cCarousel = arrCarCarousels[0];
      }
      if(arrCarCarousels.length == 1)
      {
            cCarousel = arrCarCarousels[0];
      }
      return cCarousel;
}
catch (IOException ioe)
{
      System.out.println("Excepcion de entrada-salida al intentar
            acceder a la lista de carruseles: " + ioe.toString());
      return null;
}
```

Tras pedir el carrusel al laboratorio se comprueba si realmente existe algún carrusel con ese nombre, si no es así se devuelve la variable con valor ***null***. Si este método devuelve ***null*** en lugar de un objeto ***Carousel*** esto significa que no existe ningún carrusel con ese nombre en el laboratorio, y por lo tanto se está intentando obtener un objeto que no existe.

Como se puede ver en la **Figura 1.8**, la ventana Nueva Aplicación MHP se compone de tres campos de texto. El primero de ellos que recibe el nombre de *Path* es en el que se debe introducir la ruta donde se encuentra el fichero .zip de la aplicación que se va a cargar. En este caso se tienen dos opciones, se puede escribir la ruta directamente en el campo de texto, o bien se puede utilizar el botón Examinar para explorar el sistema y seleccionar el fichero a cargar.

En el siguiente campo (Nombre) se asigna la denominación que se le quiere dar a la aplicación. Este nombre será el que aparecerá en la lista que se utiliza para gestionar las aplicaciones. Se debe recordar que no puede haber dos aplicaciones con el mismo nombre para evitar confusiones a la hora de manejarlas.

Por último se introduce el nombre del directorio destino, este campo representa la carpeta en la que se descomprimen los ficheros que contiene el archivo .zip que se está cargando. Es importante que el nombre de la carpeta también sea único, ya que esto facilita la posterior selección de la clase principal de la aplicación.

Después de introducir todos los datos en sus respectivos campos, se presiona el botón Cargar.

En primer lugar se leen los campos de texto y se guarda su valor en 3 variables de tipo ***String*** (***txPathFileApp***, ***txNameFileApp***, ***txDirDestFile***). A continuación se obtiene la lista de todas las aplicaciones disponibles con el método ***appObtention()*** de la clase ***UploadApp***, esta lista se utiliza para comprobar que no existe ninguna aplicación con el nombre que se le quiere dar a la aplicación que se está creando, porque como ya se ha comentado antes no pueden existir dos aplicaciones con el mismo nombre.

```
else if (arg0.getSource().equals(jbUploadFile))
{
	txPathFileApp = jtfPathApp.getText();
	txNameFileApp = jtfNameApp.getText();
	txDirDestFile = jtfDirDest.getText();
	String[] arrTxApp;
	arrTxApp = ua.appObtention(WindowApp.imcConnListApp);

	int i = 0;
	if (arrTxApp.length != 0)
	{
		while (i<arrTxApp.length &&
			!arrTxApp[i].equals(txNameFileApp))
		{
			i++;
		}
	}
	if (i == arrTxApp.length)
	{
		UploadApp ulapp = new UploadApp();
		ulapp.addFileZipApp(WindowApp.cCarousel);
		ulapp.createAPP(WindowApp.imcConnListApp,
			WindowApp.cCarousel);
		WindowCarouselList wcl = new WindowCarouselList();
		wcl.createWindow(WindowApp.imcConnListApp);
		WindowApp wapp = new WindowApp();
		wapp.uploadListApp();
	}
	jfNewApp.dispose();
}
```

Una vez que se ha comprobado que no existe ninguna aplicación con el mismo nombre se comienza con las acciones de crear la nueva aplicación. Como ya se ha comentado antes, este proceso se lleva a cabo en dos pasos: en primer lugar se descomprime el fichero .zip en el carrusel con el método ***addFileZipApp()*** de la clase ***UploadApp***. A continuación se crea la aplicación en el laboratorio con el método ***createAPP()*** de la misma clase. Ya se ha creado un nuevo objeto ***Application*** y ahora se le debe asignar la clase inicial, es decir la clase desde la que el laboratorio tiene que lanzar la aplicación. Para asignar esta clase inicial a la nueva aplicación se abre una nueva ventana (***WindowCarouselList***) que contiene una lista con todos los archivos que hay en este momento cargados en el carrusel, de entre estos ficheros se selecciona el que corresponde a la clase inicial de la nueva aplicación.

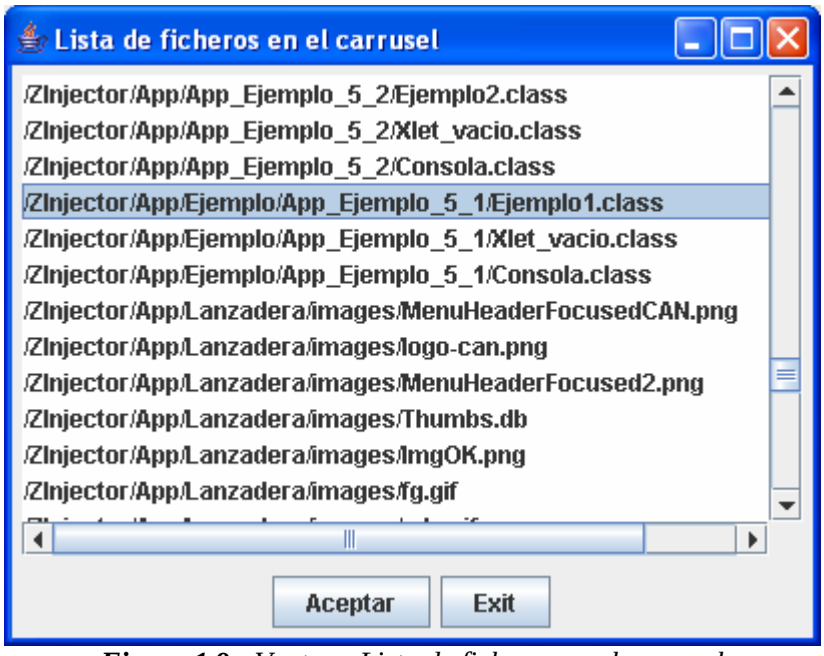


Figura 1.9.- *Ventana Lista de ficheros en el carrusel*

Para la creación de la ventana mostrada en la **Figura 1.9** primero hay que obtener del laboratorio la lista de todos los ficheros que contiene el carrusel con el que se está trabajando, para ello se utiliza el método ***getObjectList()*** que requiere 4 parámetros de entrada en los que se especifica que tipo de objetos se quieren incluir en la lista de salida:

- ****boFiles*** → Si se quieren incluir ficheros en la lista
- ****boDirectories*** → Si se quieren incluir directorios en la lista
- ****boStreamEvents*** → Si se quieren incluir *Stream Events* en la lista
- ****boCarouselLinks*** → Si se quieren incluir *carousel links* en la lista

En este caso sólo nos interesan los ficheros de modo que se pone el primer parámetro a *true* y el resto a *false*. El resto es el procedimiento habitual en estos casos, en primer lugar se limpia la lista para evitar que queden residuos de listas anteriores y se rellena con los datos obtenidos en el *array* resultante de la llamada a ***getObjectList()***.

```
//LISTA DE FICHEROS QUE CONTIENE EL CARRUSEL
/*
 * Se crea la lista
 * */

arrTxFile = WindowApp.cCarousel.getObjectList(true, false,
      false, false);

if(dlmFile.size() != 0)
{
      dlmFile.clear();
}
if(arrTxFile != null)
{
      for(int i=0; i<arrTxFile.length; i++)
      {
            dlmFile.addElement(arrTxFile[i]);
      }
}
```

Con la ventana creada y la clase principal seleccionada se presiona el botón Aceptar para asignar esta clase a la aplicación que se acaba de crear. Para ello se utiliza el método ***setInitialClass()***, pero es necesario un cambio de formato en la clase inicial, ya que tal como se obtiene al leer de la lista de la ventana no es interpretada correctamente por el laboratorio.

El método ***getObjectList()*** devuelve un *array* de ***String*** con el *path* completo de cada objeto que hay en el carrusel, y esto es lo que se representa en la lista de archivos de la ventana. Pero este formato no nos sirve como parámetro de entrada del método ***setInitialClass()***. En primer lugar no se puede utilizar así porque sobra la parte inicial de la ruta, donde se repite el *Class path* de la aplicación y no es necesario repetirlo ya que este tiene su propio campo al que ya se le ha asignado su valor en el momento de creación del objeto ***Application***. Por otro lado, y esto si que es un problema más de formato propiamente dicho, en este campo el laboratorio requiere que los paquetes que pueda tener la definición de la clase inicial estén separados por puntos en lugar de por barras (“/”) como se obtiene de la lista de archivos disponibles. También se debe eliminar el .class del final que indica la extensión del archivo. Se explica esto con un ejemplo

.

Path completo del archivo: /FA/com/cein/tdttb/main.class

Class Path: /FA

Clase inicial: com.cein.tdttb.main

Para realizar esta transformación se llama al método ***getInitialClass()*** que devuelve la clase inicial seleccionada lista para ser guardada en el laboratorio.

```
if (arg0.getSource().equals(jbSaveFile))
{
      Application[] arrAppNew;
      Application appNew = null;
      try
      {
            arrAppNew = Application.getByName(imcConnClass,
                  WindowAddApp.txNameFileApp);
            if(arrAppNew.length == 0)
            {
                  System.out.println("No se ha encontrado ninguna
                        aplicacion con ese nombre.");
            }
            if(arrAppNew.length > 1)
            {
                  System.out.println("Advertencia: Hay mas de un
                        TS con ese nombre.");
            }
            if(arrAppNew.length == 1)
            {
                  appNew = arrAppNew[0];
            }
            String txInitialClassPath;
            txInitialClassPath =
                  jlistFile.getSelectedValue().toString();
            String txClassPath;
            txClassPath = appNew.getClasspath();
            String txInitialClass;
            txInitialClass = getInitialClass(txClassPath,
                  txInitialClassPath);
            appNew.setInitialClass(txInitialClass);
            jfClass.dispose();
      }
      catch (IOException ioe)
      {
            System.out.println("Excepcion de entrada-salida al
                  consultar la lista de objetos del carrusel: " +
                  ioe.getMessage());
      }

}
```

Por último se actualiza la lista de aplicaciones de la ventana Gestión de Aplicaciones para que esté disponible la que se acaba de crear, para ello se llama al método ***uploadListApp()*** de la clase ***WindowApp***.

Para tener una visión más clara de cómo se implementa esta acción se puede ir al **Diagrama 1.27** donde se representa el diagrama de secuencia de todo lo explicado en esta operación.

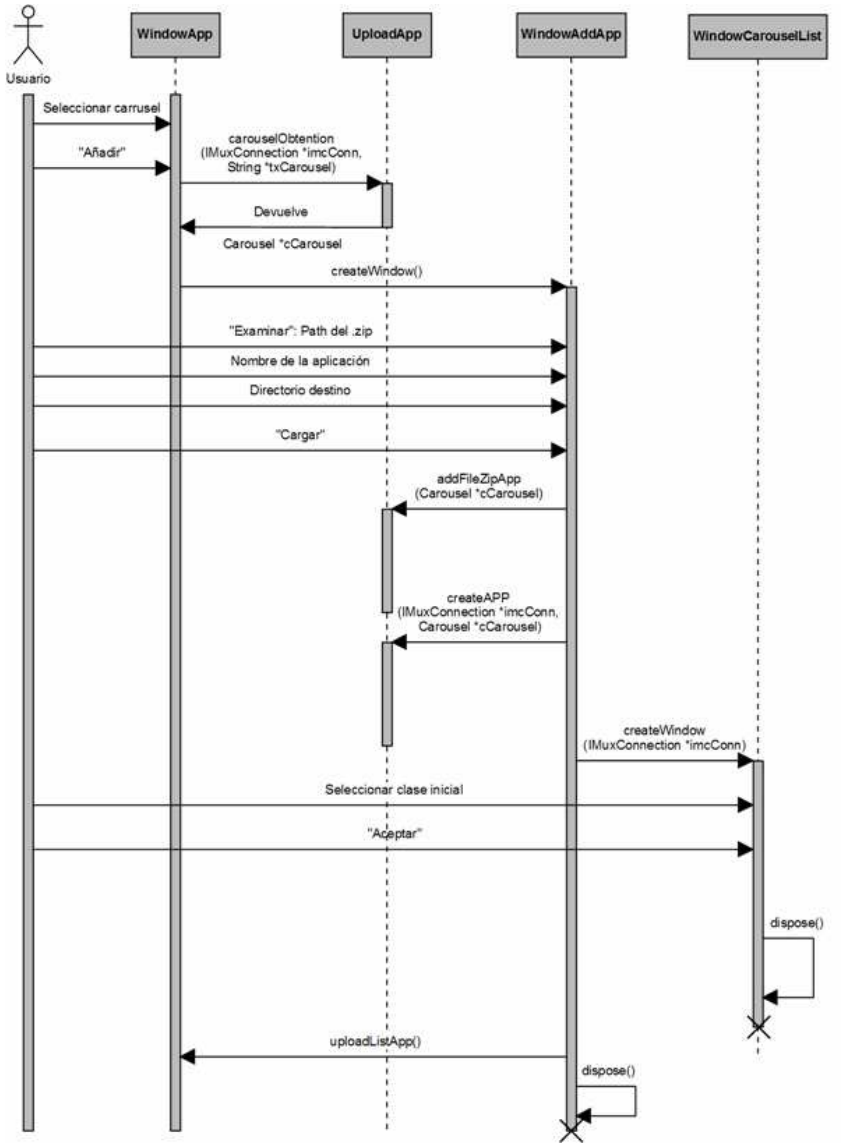


Diagrama 1.27.- *Diagrama de secuencia de la acción Añadir nueva aplicación MHP*

Método: ***void addFileZipApp(Carousel * cCarAddApp)***

- *Parámetros de entrada:*
  Carousel *cCarAddApp → Carrusel con el que se está trabajando
- *Salida:* ***void***

Este método se encarga de subir el fichero .zip correspondiente a la aplicación que se está creando.

En primer lugar se lee el campo *Path* de la ventana Nueva Aplicación MHP y se crea el objeto ***File*** que se va a subir al laboratorio. A continuación se crea la carpeta en la que se van a descomprimir todos los archivos que contiene el fichero .zip, para ello se toma el nombre de la carpeta destino también del campo correspondiente de la ventana. Se puede observar que se ha creado una variable ***txDir*** de tipo ***String*** a la que se le ha dado una ruta de acceso, esta ruta corresponde a la ubicación de la carpeta que se va a crear para la nueva aplicación. Se puede prescindir de esta variable, y la carpeta se crea en el directorio raíz del carrusel, pero en este caso se ha creado la variable para tener localizadas todas las aplicaciones de este programa en una misma carpeta, ya que en el laboratorio hay otras aplicaciones cargadas que no corresponden a las creadas desde esta aplicación.

Tras crear la carpeta destino con el método ***objectMkDir()*** se sube el objeto ***File*** con el método ***objectUploadZIP()***, este método realiza la descompresión del fichero en la dirección indicada por el parámetro ***txDirDestFile***. Ambos métodos heredan del objeto ***Carousel*** que se proporciona en la *API* del laboratorio.

```
private String txDir = new String ("/ZInjector/App/");

[…]

File fileAppZip = new File(WindowAddApp.txPathFileApp);

try
{

      txCreateDir = txDir + WindowAddApp.txDirDestFile;

      cCarAddApp.objectMkDir(txCreateDir, true);

      int iError = cCarAddApp.objectUploadZIP(txCreateDir, fileAppZip,
            true);
      if (iError == 0 )
      {
            System.out.println("El .zip de la aplicacion ha sido
                  cargado correctamente.");
```

```
        }
        else if (iError == -1)
        {
                System.out.println("El .zip de la aplicacion esta corrupto
                        y no se ha podido cargar.");
        }
        else
        {
                System.out.println("No ha sido posible cargar " + iError +
                        " elementos del .zip de la aplicacion.");
        }
}
catch (SecurityException se)
{
        System.out.println("Excepcion de seguridad al subir la
                aplicacion al carrusel: " + se.getMessage());
}
catch (IOException ioe)
{
        System.out.println("Excepcion de entrada-salida al subir la
                aplicacion al carrusel: " + ioe.getMessage());
}
```

Como se puede apreciar, el método ***objectUploadZIP()*** devuelve un valor de tipo ***int*** con el que se comprueba si el fichero se ha cargado correctamente.

Método: ***void createApp(IMuxConnection *imcConn, Carousel *cCarNewApp)***

- *Parámetros de entrada:*
  - ***IMuxConnection *imcConn*** → Objeto resultante de la conexión
  - ***Carousel *cCarNewApp*** → Carrusel con el que se está trabajando
- *Salida:* ***void***

Este método crea un nuevo objeto ***Application*** en el laboratorio. La creación del nuevo objeto se lleva a cabo a través de la ejecución del método ***createApplication()***, a este método se le pasan como parámetros:

- ***IMuxConnection *imcConn*** → Objeto resultante de la conexión
- ***String *txNameFileApp*** → Nombre que se le quiere dar a la aplicación
- ***int *iOrgID*** → *OrgID* para la transmisión de la aplicación
- ***int *iAppID*** → *AppID* para la transmisión de la aplicación
- ***boolean *boRemote*** → Verdadero si se trata de una aplicación remota

A continuación se le asignan tanto el carrusel en el que se está subiendo como el *classpath* donde se encuentran todos sus archivos, con los métodos ***setDsmccSource()*** y ***setClasspath()*** respectivamente.

Todos estos parámetros son requeridos por el laboratorio para la correcta creación de la aplicación.

```
Application appNew;

try
{
      appNew = Application.createApplication(imcConn,
            WindowAddApp.txNameFileApp, 500, 501, false);
      appNew.setDsmccSource(cCarNewApp);
      appNew.setClasspath(txCreateDir);
}
catch (SecurityException se)
{
      System.out.println("Excepcion de seguridad al crear una
            nueva aplicacion: " + se.getMessage());
}
catch (IOException ioe)
{
      System.out.println("Excepcion de entrada-salida al crear
            una nueva aplicacion: " + ioe.getMessage());
}
```

Método: ***String getInitialClass(String *txClassPath, String *txInitialClassPath)***

- *Parámetros de entrada:*
  - ***String *txClassPath*** → ***String*** con el *class path* de la aplicación
  - ***String *txInitialClassPath*** → ***String*** con el *path* completo de la clase inicial seleccionada
- Salida: ***String*** → *Path* de la clase inicial ya transformado

Como se ha comentado este método se encarga de realizar la transformación de la clase inicial para hacerla interpretable por el laboratorio.

En la primera parte del método se recorren ambos ***String*** (*class path* y *path* de la clase inicial) para quitar la parte correspondiente al *class path* del ***String*** de la clase inicial. Realmente en este bucle no se elimina nada, sino que lo que se hace es buscar el índice desde el cual se comienza a leer el *path* real de la clase inicial. Este índice se queda guardado en la variable ***k*** de tipo ***int***, este valor se utiliza como punto de inicio para la

segunda parte en la que se cambian las barras (“/”) por puntos y se elimina el .class del final.

```
int j = 0;
int k = 0;
int iIndexClassPath = 0;
int iIndexInitialClassPath = 0;

while( k<txClassPath.length() || iIndexClassPath != (-1))
{
	iIndexClassPath = txClassPath.indexOf("/", j);
	if(iIndexClassPath != -1)
	{
		j = iIndexClassPath + 1;
	}
	iIndexInitialClassPath = txInitialClassPath.indexOf("/", k);
	k = iIndexInitialClassPath + 1;
}
String txAux;
String txInitialClass = "";
while(iIndexInitialClassPath != -1)
{
	iIndexInitialClassPath = txInitialClassPath.indexOf("/", k);
	if(iIndexInitialClassPath != -1)
	{
		txAux = txInitialClassPath.substring(k,
			iIndexInitialClassPath);
		k = iIndexInitialClassPath + 1;
		txInitialClass = txInitialClass + txAux;
		txInitialClass = txInitialClass + ".";
	}
}
iIndexInitialClassPath = txInitialClassPath.indexOf(".", k);
txAux = txInitialClassPath.substring(k, iIndexInitialClassPath);
txInitialClass = txInitialClass + txAux;

return txInitialClass;
```

En el segundo bucle se utiliza una variable auxiliar ***txAux*** en la que se van guardando los ***substring*** que se obtienen para los distintos paquetes de la clase inicial. Con esta variable también se va montando la nueva variable ***txInitialClassPath*** que se ha limpiado antes de entrar en el bucle. En cada iteración se intercala un punto entre los paquetes para darle el formato requerido por el laboratorio.

Método: ***void uploadListApp()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

Cada vez que se realiza alguna modificación (como borrar o añadir una aplicación) en la lista de aplicaciones disponibles, se debe actualizar ésta en la ventana Gestión de Aplicaciones. Este método se encarga de refrescar esta lista y se debe llamar tras cada modificación.

El desarrollo de este método se puede dividir en dos pasos, en una primera parte se limpia la lista de las aplicaciones anteriores, y posteriormente se rellena con las aplicaciones disponibles en este momento. Previamente se ha obtenido un *array* con los nombres de las aplicaciones activas en este momento en el laboratorio con el método ***appObtention()***, este *array* se usa en el bucle *for* para rellenar la nueva lista.

```
arrTxApp = ua.appObtention(imcConnListApp);

if (dlmListApp.size() != 0)
{
      dlmListApp.clear();
}

if (arrTxApp != null)
{
      for (int i=0; i<arrTxApp.length; i++)
      {
            dlmListApp.addElement(arrTxApp[i]);
      }
}
```

1.2.2.- Borrar aplicación MHP

Del mismo modo que se tiene la opción de añadir aplicaciones, se deben poder borrar aquellas que ya no son necesarias para evitar que se vayan acumulando y se dificulte así su manejo. Por ese motivo se ha implementado esta funcionalidad, que permite, una vez seleccionada una aplicación, borrarla tanto del laboratorio como de la lista visible de la ventana.

Como ya se ha visto, esta acción está disponible desde la ventana Gestión de Aplicaciones, y su funcionamiento es muy sencillo. Lo primero que se debe hacer es seleccionar en la lista la aplicación que se desea borrar, y presionar el botón Borrar situado

debajo de la lista. Al pulsar este botón se llama al método ***deleteAPP()*** que pertenece a la clase ***UploadApp***. Este método se encarga de realizar las acciones necesarias para borrar completamente la aplicación, el desarrollo del mismo se explicará más adelante.

```
//BOTON BORRAR APLICACION
else if (arg0.getSource().equals(jbDelApp))
{
      String txDelApp;
      txDelApp = jlApp.getSelectedValue().toString();
      UploadApp ulapp = new UploadApp();
      ulapp.deleteAPP(imcConnListApp, WindowApp.cCarousel, txDelApp);
      uploadListApp();
}
```

Tras borrar la aplicación del laboratorio se llama al método ***uploadListApp()***, que ya se ha descrito unas líneas más arriba, para refrescar la lista de aplicaciones disponibles, ya que se ha visto modificada por la desaparición de la aplicación que se acaba de borrar. Todo esto se puede ver representado en el **Diagrama 1.28**.

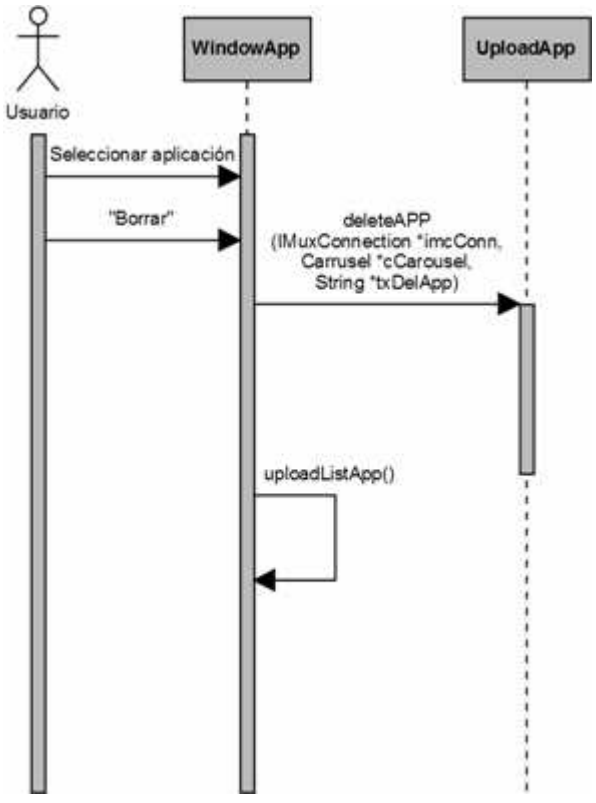


Diagrama 1.28.- *Diagrama de secuencia de la acción Borrar una aplicación MHP*

Método: ***void deleteAPP(IMuxConnection *imcConn, Carousel cCarDelApp, String *txDelApp)***

- *Parámetros de entrada:*
  IMuxConnection *imcConn → Objeto resultante de la conexión
  Carousel *cCarDelApp → Carrusel con el que se está trabajando
  String *txDelApp → Nombre de la aplicación que se va a borrar
- *Salida:* ***void***

Este método se encarga de borrar la aplicación seleccionada del laboratorio, para ello, al igual que pasaba a la hora de añadir una aplicación nueva se llevan a cabo dos acciones distintas. Por un lado se deben borrar todos los ficheros que pertenecen a la aplicación del carrusel, y por otro, se debe eliminar el objeto ***Application*** que se ha creado en el laboratorio.

En primer lugar se debe buscar la aplicación que se quiere borrar en el laboratorio, para ello se utiliza el método ***getByName()*** de la clase ***Application*** que se proporciona en la *API* del laboratorio ya que se tiene el nombre de la aplicación a borrar.

El objeto ***Application*** resultante del método ***getByName()*** se guarda en la variable ***appDel***. A partir de esta variable se obtiene el *class path* de la aplicación que se utiliza para borrar todos los archivos de la aplicación que hay cargados en el carrusel. En concreto se elimina la carpeta que se encuentra en la ruta indicada por el *class path*. Esta carpeta contiene todo el contenido que se descomprimió en el momento en que se creó la aplicación.

Por último se borra el objeto ***appDel*** del laboratorio llamando al método ***delete()***, para concluir con la eliminación definitiva de la aplicación.

```
Application[] arrAppDel;
Application appDel = null;
String txDirApp;
try
{
      arrAppDel = Application.getByName(imcConn, txDelApp);
      if(arrAppDel.length == 0)
      {
            System.out.println("No se ha encontrado ninguna
                  aplicacion con ese nombre.");
      }
      if(arrAppDel.length > 1)
      {
```

```
            System.out.println("Advertencia: Hay mas de una
                  aplicacion con ese nombre.");
            appDel = arrAppDel[0];
      }
      if(arrAppDel.length == 1)
      {
            appDel = arrAppDel[0];
      }
      if (appDel != null)
      {
            txDirApp = appDel.getClasspath();
            cCarDelApp.objectDelete(txDirApp);
            appDel.delete();
      }
}
catch (SecurityException se)
{
      System.out.println("Excepción de seguridad al borrar una
            aplicacion: "+se.toString());
}
catch (IOException ioe)
{
      System.out.println("Excepción de entrada-salida al borrar
            una aplicacion: "+ioe.toString());
}
```

1.2.3.- Crear Schedule de aplicaciones

Ya se tiene la lista de aplicaciones disponibles y se sabe cómo gestionarla, es decir, cómo añadir nuevas aplicaciones o como borrar las que ya no son necesarias. Ahora se va a describir cómo utilizar estas aplicaciones, es decir, cómo se pueden programar para que estén disponibles en los receptores en los momentos precisos.

El concepto de escaleta de aplicaciones difiere un poco del que se trató para el caso de los vídeos. Cuando se habla de vídeos, una escaleta consiste en una consecución de vídeos previamente seleccionados y ordenados, y en la que sólo puede haber un vídeo en reproducción al mismo tiempo. En cambio, para las aplicaciones la escaleta es una tabla en la que quedan reflejadas todas las entradas que se han configurado, y si que puede haber más de una aplicación disponible al mismo tiempo. Se debe tener en cuenta que las aplicaciones con las que estamos trabajando no son, en general, autoejecutables y se accede a ellas cuando el usuario expresamente lo desea. Así cuando un usuario esté interesado en consultar alguna aplicación, accederá a los servicios interactivos a través de algún botón de su mando a distancia. Lo primero que ve un usuario al acceder a las aplicaciones interactivas en TDT es lo que se denomina lanzadera. La lanzadera es una herramienta que permite la navegación a través de los servicios interactivos, sería lo que

conocemos como menú principal en una interfaz. A través de esta aplicación lanzadera, el espectador selecciona y lanza la aplicación deseada. Para acceder a ella se presiona el botón del mando que da acceso a los servicios interactivos, y se desplegará un menú con las aplicaciones disponibles en ese momento. Cuando el usuario acceda a la lanzadera, las aplicaciones que se quiere que vea el usuario deben estar activadas. Ese es el motivo por el cual es necesario crear distintos *Schedule* para las aplicaciones, por ejemplo puede interesar que una determinada aplicación esté activa todo el día, sin embargo otra sólo interesa que esté activada durante la tarde o durante la emisión de un programa o película concreta. Así, cada aplicación tiene un rango de tiempo durante el cual es accesible a los usuarios y aparece en la aplicación lanzadera. Y esto se consigue con la creación de los *Schedule*.

El término anglosajon *Schedule* se puede traducir como horario, calendario, programación,… En este caso se habla de *Schedule* para referirse a la planificación de la disponibilidad de las distintas aplicaciones gestionadas en los receptores. En el *Schedule* de aplicaciones se definen las horas de inicio y fin entre las que se quiere que una determinada aplicación este disponible, es decir, se programa la disponibilidad de las aplicaciones a través de los *Schedule*.

Para crear un nuevo *Schedule* primero se presiona el botón Crear App *Schedule* y en ese momento se lanza una nueva ventana (Nueva entrada en la escaleta de aplicaciones) desde la que se configura el nuevo *Schedule*.

```
//BOTON AÑADIR SCHEDULE DE APLICACIONES
else if (arg0.getSource().equals(jbAddScheItemApp))
{
      WindowCreateScheduledApp wcsa = new
            WindowCreateScheduledApp();
      wcsa.createWindow();
}
```

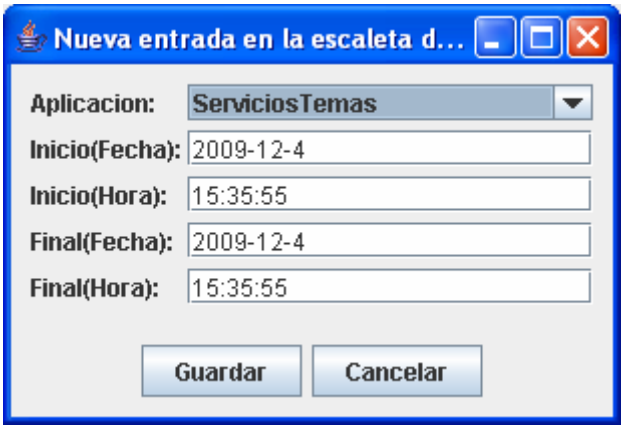


Figura 1.10.- *Ventana Nueva entrada en la escaleta de aplicaciones*

Como se puede ver en la **Figura 1.10**, en la parte superior hay un ***Combo Box*** el cual se ha rellenado con la lista de aplicaciones disponibles para seleccionar en él la aplicación de la que se quiere crear el *Schedule*. A continuación hay cuatro campos de texto donde se deben introducir las fechas y horas de inicio y de fin del *Schedule*. En el momento de creación de la ventana se les da a estos campos el valor de la fecha y hora actual, esta información puede servir de referencia de los formatos utilizados a la hora de introducir los datos definitivos. Debido a la forma en que se ha programado esta funcionalidad, para que el *Schedule* se cree correctamente se deben introducir los datos con el formato adecuado, si no se hace así, la aplicación no será capaz de leer correctamente los datos introducidos por el ususario.

Después de seleccionar la aplicación y de introducir las horas de inicio y fin se presiona el botón Guardar para hacer efectiva la creación del *Schedule*. Al pulsar este botón se llama al método ***createScheduledApp()*** de la clase ***CreateScheduledApp*** donde se ejecutan las acciones necesarias para crear en el laboratorio el *Schedule***.**

```
public static String txAppSt = "";
public static String txInitDateAppSt = "";
public static String txInitTimeAppSt = "";
public static String txEndDateAppSt = "";
public static String txEndTimeAppSt = "";

[…]

if (arg0.getSource().equals(jbSaveSchApp))
{
      txAppSt = jcbApp.getSelectedItem().toString();
      txInitDateAppSt = jtfInitDateApp.getText();
      txInitTimeAppSt = jtfInitTimeApp.getText();
      txEndDateAppSt = jtfEndDateApp.getText();
      txEndTimeAppSt = jtfEndTimeApp.getText();
```

```
        CreateScheduledApp csapp = new CreateScheduledApp();
        csapp.createScheduledApp(WindowApp.imcConnListApp);

        jfNewSchApp.removeAll();
        jfNewSchApp.dispose();
}
```

Antes de llamar al método se leen los datos introducidos en la ventana y se guardan en 5 variables estáticas de tipo ***String*** que se utilizan posteriormente en el método para crear correctamente el *Schedule*.

Una vez que se ha retornado del método ***createScheduledApp()*** con el *Schedule* creado se cierra la ventana con el método ***dispose()***.

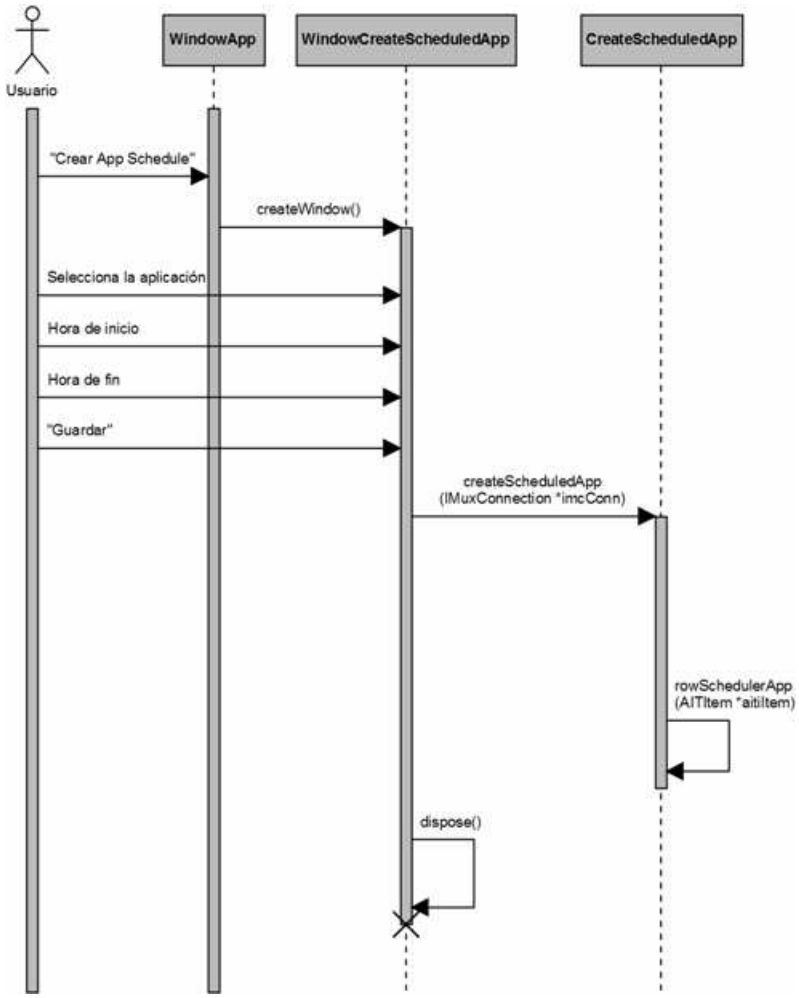


Diagrama 1.29.- *Diagrama de secuencia de la acción Crear un nuevo* Schedule *de aplicaciones*

Método: ***void createScheduleApp(IMuxConnection *imcConn)***

- *Parámetros de entrada:*
  IMuxConnection *imcConn → Objeto resultante de la conexión
- *Salida:* ***void***

Como ya se ha comentado este método crea un *Schedule* en el laboratorio. El laboratorio tiene su propio gestor de *Schedule* de modo que se va a aprovechar para facilitar así las cosas. En este caso no se tiene el inconveniente de la precisión, ya que a la hora de tratar las aplicaciones el hecho de tener una precisión de minutos es menos importante que en el caso de los vídeo. Para los vídeos no se pudo hacer uso de este gestor, pero en este caso se va a sacrificar la precisión porque compensa más la simplificación que esto supone en la programación de la escaleta de aplicaciones.

En cuanto al desarrollo del método, éste comienza recuperando la aplicación de la cual se quiere crear el *Schedule* en una variable de tipo ***Application*** y de nombre ***aApplication***. Para ello se utiliza el método ***getByName()*** al que se le pasa como parámetro el nombre de la aplicación seleccionada que se encuentra guardado en la variable estática ***txAppSt*** de la clase ***WindowCreateScheduledApp***.

```
String txAppSelec = WindowCreateScheduledApp.txAppSt;

Application[] arrApp = Application.getByName(imcConn,
      txAppSelec);
if (arrApp.length == 0)
{
      System.out.println("No se ha encontrado ninguna aplicación
            con ese nombre.");
}

if (arrApp.length>1)
{
      aApplication = arrApp[0];
      System.out.println("Advertencia: Se han encontrado varias
            aplicaciones con ese nombre.");
}
if(arrApp.length == 1)
{
      aApplication = arrApp[0];
}
```

A continuación se recupera también el servicio con el que se está trabajando en la variable ***sService*** de tipo ***Service***, y también se usa el método ***getByName()***.

```
Service[] arrService = Service.getByName(imcConn, "Prueba");
if (arrService.length == 0)
{
      System.out.println("No se ha encontrado ningun servicio con
            ese nombre.");
}
if (arrService.length>1)
{
      sService = arrService[0];
      System.out.println("Advertencia: Se han encontrado varios
            servicios con ese nombre.");
}
if(arrService.length == 1)
{
      sService = arrService[0];
}
```

El siguiente paso es recuperar la *AIT* del servicio que se acaba de obtener. Si ésta no existe todavía, se crea con el método ***createAIT()*** de la clase ***AIT***.

```
ait = sService.getAIT();

if (ait == null)
{
      ait = AIT.createAIT(imcConn, sService.getName());
}
```

Por último se crea el ***ScheduleItem*** con las horas de inicio y fin que se le quieren dar a la aplicación que se está programando.

```
AITItem aitiItem = ait.createAITItem(aApplication);
String[] arrTxFrom = {
            WindowCreateScheduledApp.txInitDateAppSt,
            WindowCreateScheduledApp.txInitTimeAppSt
};
String[] arrTxTo = {
            WindowCreateScheduledApp.txEndDateAppSt,
            WindowCreateScheduledApp.txEndTimeAppSt
};
Date dFrom = DateFunctions.stringToDate(arrTxFrom);
Date dTo = DateFunctions.stringToDate(arrTxTo);
aitiItem.createAITScheduleItem
      (AITScheduleItem.SCHEDULE_TYPE_ONCE, dFrom, dTo);
rowSchedulerApp(aitiItem);
```

Antes de salir del método, se tiene que crear la fila que representa el *Schedule* en la tabla que registra todas las aplicaciones que se han programado. Para ello se tiene el método ***rowScheduleApp()***.

Método: ***void rowScheduleApp(AITItem *aitItemRow)***

- *Parámetros de entrada:*
  AITItem *aitItemRow → Objeto ***AITItem*** del cual se quiere crear la fila
- *Salida:* ***void***

En la ventana Gestión de Aplicaciones hay una tabla en la que se visualizan los *Schedule* que se han configurado. De modo que cada vez que se crea un nuevo *Schedule* se debe añadir una nueva fila a la tabla, esta fila se compone de 5 celdas, en primer lugar se coloca el nombre de la aplicación programada, las dos celdas siguientes corresponden a la fecha y hora de inicio de la activación de la aplicación y las dos últimas celdas contienen la fecha y hora en que la aplicación deja de estar habilitada.

Antes de describir el desarrollo del método, se van a explicar un par de conceptos que facilitan la comprensión de la manera en que se ha desarrollado el método. En primer lugar comentar que se tienen dos objetos que pueden parecer iguales pero que no lo son, y esto puede llevar a confusiones. Por un lado se tienen los ***AITItems***, que es el objeto que se pasa como parámetro de entrada a este método. Este tipo de objetos son, como su propio nombre indica, *Items* de la *AIT*, es decir, una vez seleccionada la *AIT* de un servicio se le van añadiendo aplicaciones a la tabla por medio de estos objetos ***AITItem***. Por otro lado se tienen los ***AITScheduleItem***, estos objetos son los que realmente representan los *Schedule* que se pretenden crear en este caso. Estos ***AITScheduleItem*** se crean dentro de un ***AITItem*** ya existente, Por lo tanto primero se debe crear un ***AITItem*** y después dentro de él se pueden añadir tantos ***AITScheduleItem*** como se desee. En este caso, debido a la forma en que se ha programado la funcionalidad Añadir *Schedule*, se tiene un único ***AITScheduleItem*** por cada ***AITItem***, es decir, que cada vez que se añade un *Schedule* nuevo se crean un ***AITItem*** y un ***AITScheduleItem*** nuevos, pero esto no tiene porque ser así, porque como se ha comentado antes, un ***AITItem*** puede contener varios ***AITScheduleItem***.

A continuación se describe cómo se ha programado este método. En primer lugar se obtiene el nombre de la aplicación y la ID del *Item* a partir del objeto ***AITItem*** que se ha pasado como parámetro.

```
txRowNameApp = aitItemRow.getApplication().getName();
iIDSchApp = aitItemRow.getId();
txIDSchApp = iIDSchApp.toString();
```

A continuación, se obtienen los ***AITScheduleItem*** que contiene el parámetro ****aitItemRow***, y para cada uno de los ***AITScheduleItem*** se crea una fila para la tabla. Como se ha comentado antes, en general, sólo va a tener un ***AITScheduleItem*** porque se han creado así, pero se utiliza un bucle *for* para evitar errores.

```
AITScheduleItem[] arrAitSchItem;
arrAitSchItem = aitItemRow.getAITScheduleItems();
if (arrAitSchItem != null)
{
      for (int j=0; j<arrAitSchItem.length; j++)
      {
            Date dFromRow;
            dFromRow = arrAitSchItem[j].getFrom();
            String[] arrTxDFromRow;
            arrTxDFromRow = DateFunctions.dateToString(dFromRow);
            String txRowInitDate;
            String txRowInitHour;
            txRowInitDate = arrTxDFromRow[0];
            txRowInitHour = arrTxDFromRow[1];

            Date dToRow;
            dToRow = arrAitSchItem[j].getTo();
            String[] arrTxDToRow;
            arrTxDToRow = DateFunctions.dateToString(dToRow);
            String txRowEndDate;
            String txRowEndHour;
            txRowEndDate = arrTxDToRow[0];
            txRowEndHour = arrTxDToRow[1];

            String[] arrTxRowScheduler = {
                              txRowNameApp,
                              txRowInitDate,
                              txRowInitHour,
                              txRowEndDate,
                              txRowEndHour,
                              txIDSchApp
            };
            WindowApp.dtmSchedulerApp.addRow(arrTxRowScheduler);
      }
}
```

Para crear cada fila de la tabla, primero se lee el *From* del ***AITScheduleItem***, como éste se obtiene en un objeto ***Date***, se transforma en dos variables de tipo ***String*** con el método ***dateToString()***. Se repiten los mismos pasos para el *To*, y una vez que se tienen las 4 variables de inicio y fin se crea un *array* de ***String*** donde se guardan estas 4 variables junto con el nombre de la aplicación y la ID del *Item* que se han obtenido anteriormente.

Este *array* representa los datos que van a componer la nueva fila de la tabla, de modo que sólo queda añadir la fila pasando como parámetro de entrada el *array* que se acaba de formar.

1.2.4.- Borrar Schedule de aplicaciones

Del mismo modo que se pueden añadir nuevos *Schedule*, es necesario poder borrarlos. Para borrar un *Schedule* primero se selecciona en la tabla la fila que representa el *Schedule* que se quiere eliminar y a continuación, se presiona el botón Borrar *Schedule* situado debajo de la tabla, como se puede ver en la **Figura 1.7** de la ventana Gestión de Aplicaciones que se ha incluido más arriba.

```
//BOTON BORRAR SCHEDULE DE APLICACIONES
else if (arg0.getSource().equals(jbDelScheItemApp))
{
      int iIndexDel;
      iIndexDel = jtSchedulerApp.getSelectedRow();
      CreateScheduledApp.removeAppSchedule(iIndexDel);
}
```

Al presionar el botón, se coge el índice de la fila seleccionada y se guarda su valor en la variable ***iIndexDel***, esta variable se pasa como parámetro de entrada en el método ***removeAppSchedule()***, el cual se encarga de borrar el *Schedule* del laboratorio y eliminar la fila de la tabla.

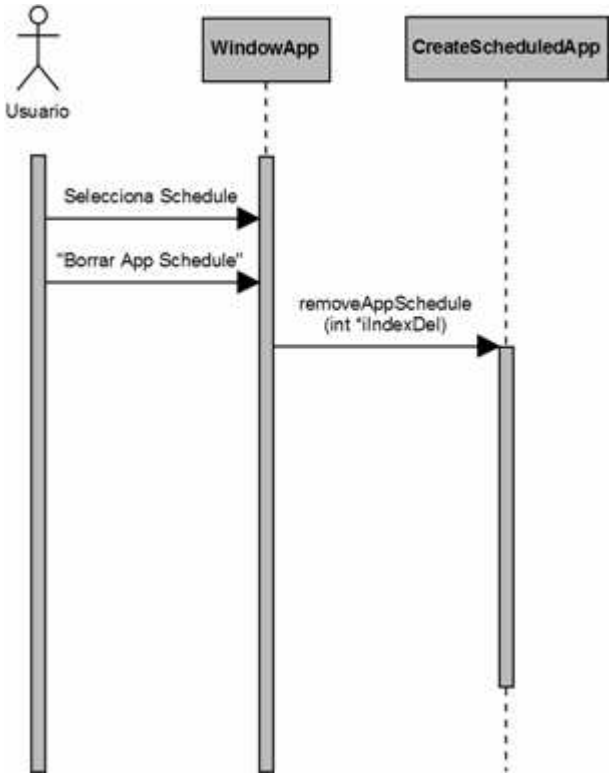


Diagrama 1.30.- *Diagrama de secuencia de la acción Borrar un* Schedule *de aplicaciones*

Método: ***void removeAppSchedule(int *iIndexDel)***

- *Parámetros de entrada:*
  int *iIndexDel → Índice de la fila seleccionada
- *Salida:* ***void***

Este método se encarga de eliminar el *Schedule* del laboratorio y de borrar la fila seleccionada de la tabla.

En primer lugar se obtiene el servicio con el que se está trabajando y se guarda en la variable ***sService***, como ya se ha hecho en ocasiones anteriores.

```
Service[] arrService;
arrService = Service.getByName(WindowApp.imcConnListApp, "Prueba");
if (arrService.length == 0)
{
      System.out.println("No se ha encontrado ningun servicio con
            ese nombre.");
}
if (arrService.length>1)
{
      sService = arrService[0];
      System.out.println("Advertencia: Se han encontrado varios
```

```
            servicios con ese nombre.");
}
if(arrService.length == 1)
{
      sService = arrService[0];
}
```

A continuación se busca la *AIT* correspondiente a este servicio, para obtener todos sus ***AITItems***.

```
ait = sService.getAIT();

if (ait != null)
{
      AITItem[] arrAitItem;
      arrAitItem = ait.getAITItems();
      if (arrAitItem != null)
      {
            for (int i=0; i<arrAitItem.length; i++)
            {
                  String txIDSchApp;
                  txIDSchApp =
                        WindowApp.jtSchedulerApp.getValueAt
                        (iIndexDel, 5).toString();
                  int iIDSchApp;
                  iIDSchApp = Integer.parseInt(txIDSchApp);
                  if (arrAitItem[i].getId() == iIDSchApp)
                  {
                        arrAitItem[i].delete();
                  }
            }
      }
}
```

Una vez que se tienen todos los *Items* guardados en un *array*, se lee el ID del *Item* de la fila seleccionada en la tabla, y se busca el *Item* que corresponda con esa ID para borrarlo.

Por último se borra la fila seleccionada de la tabla.

```
WindowApp.dtmSchedulerApp.removeRow(iIndexDel);
```

1.2.5.- Actualizar la escaleta de aplicaciones

Esta funcionalidad surge fruto de otro problema que se detecta en el laboratorio. En este caso, el gestor de *Schedule* del ***iMux*** no elimina los *Schedule* de aplicaciones cuando ya han finalizado, es decir, que se van acumulando en el laboratorio y la única manera de eliminarlos es borrarlos uno a uno manualmente. Esto no es muy práctico, ya que tiene que ser el usuario el que compruebe si el *Schedule* ya ha expirado y borrarlo él manualmente. Para solucionar esta limitación del ***iMux***, se ha decidido implementar un nuevo botón Actualizar cuya función es detectar los *Schedule* que ya han sido ejecutados y eliminarlos. Para ello se ha implementado el método ***updateScheduleApp()***. Esto se representa en el **Diagrama 1.31** que representa el diagrama de secuencia de la acción Actualizar la escaleta de aplicaciones.

```
//BOTON ACTUALIZAR ESCALETA DE APLICACIONES
else if (arg0.getSource().equals(jbUpdateSchedule))
{
      CreateScheduledApp.updateSchedulerApp();
}
```

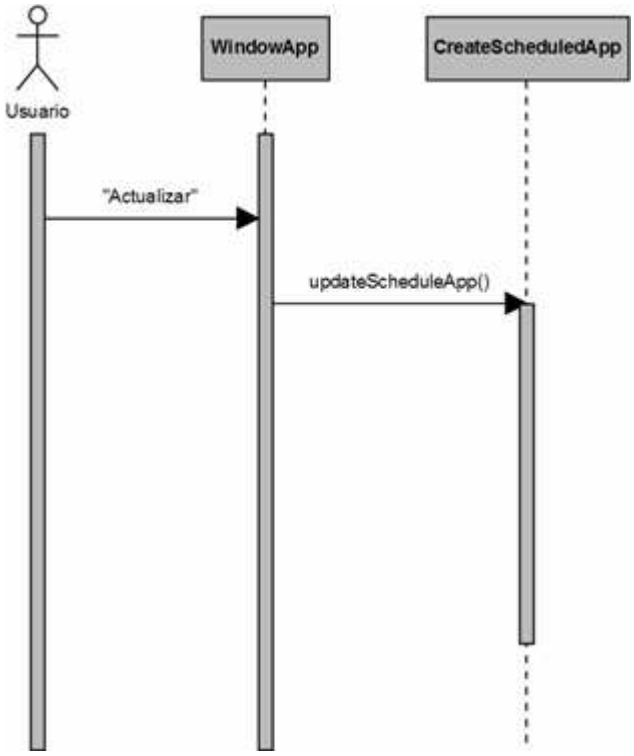


Diagrama 1.31.- *Diagrama de secuencia de la acción Actualizar la escaleta de aplicaciones*

Método: ***void updateScheduleApp()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

Básicamente lo que se hace en este método es recorrer la tabla de la escaleta, y para cada fila se comprueba si la hora de fin de cada *Schedule* es menor que la hora actual. Si es así significa que ya ha terminado ese *Schedule* y por tanto se borra tanto de la tabla como del laboratorio, para ello se utiliza el método ***removeAppSchedule()*** que se acaba de describir más arriba.

```
for (int i=0; i<WindowApp.jtSchedulerApp.getRowCount(); i++)
{
	String[] arrTxEndTime = new String[2];
	String txEndDate;
	txEndDate = WindowApp.dtmSchedulerApp.getValueAt(i,3)
		.toString();
	String txEndTime;
	txEndTime = WindowApp.dtmSchedulerApp.getValueAt(i,4)
		.toString();
	arrTxEndTime[0] = txEndDate;
	arrTxEndTime[1] = txEndTime;

	Date dEndTime;
	dEndTime = DateFunctions.stringToDate(arrTxEndTime);
	Date dNow = new Date();

	if (dNow.getTime() > dEndTime.getTime())
	{
		removeAppSchedule(i);
		i--;
	}
}
```

1.3.- Generación de Stream Events

1.3.1.- Introducción a Stream Events

Cuando se trabaja con aplicaciones MHP, a menudo es necesario que estas estén sincronizadas con los contenidos audiovisuales para los que han sido creadas. Por ejemplo, si se hace una aplicación para un concurso de preguntas y respuestas, interesa que el usuario pueda contestar a dichas preguntas a la vez que éstas son formuladas al concursante en plató. Si se quiere presentar la pregunta 3 en el minuto 7 del programa, el problema es: ¿cómo sabe el receptor cuando se llega al minuto 7 de programa?. En muchos sistemas multimedia esto se hace considerando cuando empieza el programa, y a partir de entonces se cuentan los 7 minutos. Cuando se trata de servicios de televisión digital, enviar una indicación de cuando empieza un programa no es una solución válida. En televisión no existe un concepto real de tiempo, ya que el usuario puede incorporarse al flujo en cualquier punto. Si se selecciona el programa una vez empezado no hay forma de saber cuanto tiempo ha transcurrido desde que empezó el programa, o si se cambia de canal en mitad de un programa, también se pierde la referencia de tiempos. Pero si no se puede usar el tiempo como mecanismo para la sincronización de las aplicaciones, ¿cómo se puede obtener un tiempo de referencia que permita determinar el momento en que se debe activar un evento? Para solucionar esto DSM-CC (*Digital Storage Media Command and Control*) define el NPT (*Normal Play Time*).

NPT (*Normal Play Time*)

Se trata de un código de tiempos asociados a un flujo. Se envía en el flujo de transporte mediante descriptores especiales que viajan en secciones privadas de MPEG. Aunque este código de tiempos va completamente separado de los DSM-CC *Stream Events* (ambos viajan en flujos distintos), los DSM-CC pueden usar estos códigos para ayudarles a decidir cuando activar los eventos. Como se verá mas adelante los *Stream Events* transportan en uno de sus campos el valor NPT al que debe ejecutarse el evento cuando el NPT llegue a este valor indicado por el *Stream Event*. Es decir, se utiliza el NPT para indicarle al receptor cuando se debe producir un evento.

Un código de tiempo NPT puede comenzar en cualquier valor y normalmente se va incrementando a lo largo de un programa (como si fuera un reloj que cuenta su duración), aunque puede tener discontinuidades que reflejen ediciones de contenido. Es decir, que incluso si un flujo que contiene referencias NPT es editado (ya sea porque el distribuidor

ha decidido acortar el programa, o que se le hayan introducido anuncios y por lo tanto se haya alargado en el tiempo), el valor del NPT para un determinado punto del flujo no necesita actualización y puede seguir siendo el mismo para ese segmento de contenidos. Esto se ilustra en los siguientes diagramas (**Figura 1.11** y **Figura 1.12**):

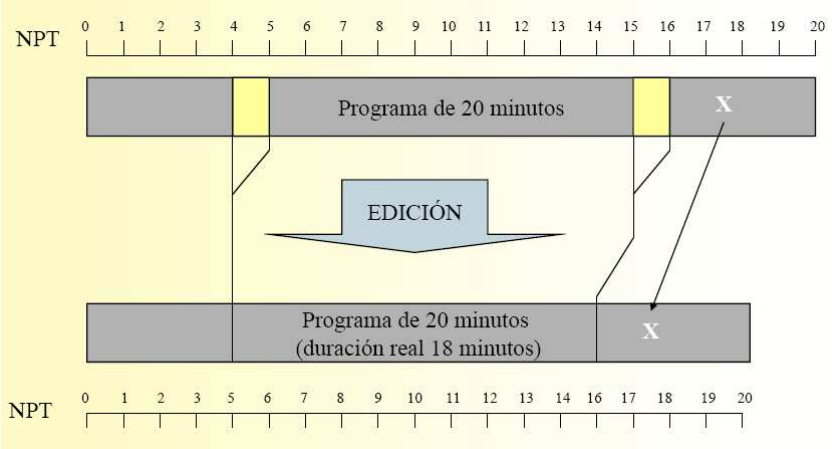


Figura 1.11.- *El difusor ha reducido la duración del programa, pero la referencia NPT (marcada con una X) se mantiene.*

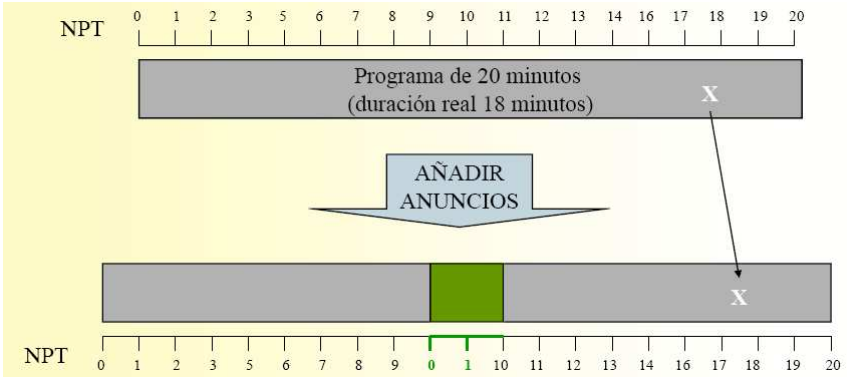


Figura 1.12.- *También se permite el anidamiento de NPTs. Se ha insertado un bloque de anuncios y la referencia NPT se mantiene en el mismo minuto de la aplicación.*

A pesar de la edición, los NPTs de la versión editada son los mismos que se encontraban en la versión original. Esta es una gran ventaja que tiene el uso de NTPs.

A diferencia de la sincronización con impulsos de reloj, en este caso no se envían descriptores NPT de forma periódica a modo de reloj ya que esto consumiría mucho ancho de banda. Sólo se envían descriptores para indicar cambios que afecten al NPT. El receptor se encarga de calcular el NPT.

Los descriptores "*NPT reference descriptor*" informan al receptor acerca de los valores del NPT en un punto específico del flujo, mientras que los "*NPT endpoint descriptor*" le informan sobre el inicio y el final de los tiempos para un evento en particular. Cada descriptor "*NPT reference descriptor*" consta de tres partes principales:

- Valor NPT: es el valor exacto del NPT en ese punto del flujo.
- Tasa de NPT: las tasas son valores fraccionarios y se almacenan como un numerador y un denominador. Esto permite al receptor del NPT interpolar entre los valores de referencia de un NPT de un descriptor y el próximo.
- ID del contenido: permite identificar las partes anidadas de los contenidos.

Con la información recibida del valor y del ID se pueden manejar casos específicos del medio televisivo como puede ser la inserción de anuncios. Si se analiza esto desde el punto de vista del receptor, un cambio en la identificación del contenido permite distinguir entre varios casos:

- Si llega un descriptor NPT con un nuevo valor NPT y el mismo ID de contenido que el anterior: discontinuidad de edición.
- Si llega un descriptor NPT con un nuevo ID de contenido :
  - Si también llega un descriptor de tipo "*NPT endpoint descriptor*": cambio de programa.
  - En caso contrario: anidamiento (por ejemplo, anuncios)

En la **Figura 1.13** se ilustran los tres casos, con los descriptores asociados a cada discontinuidad:

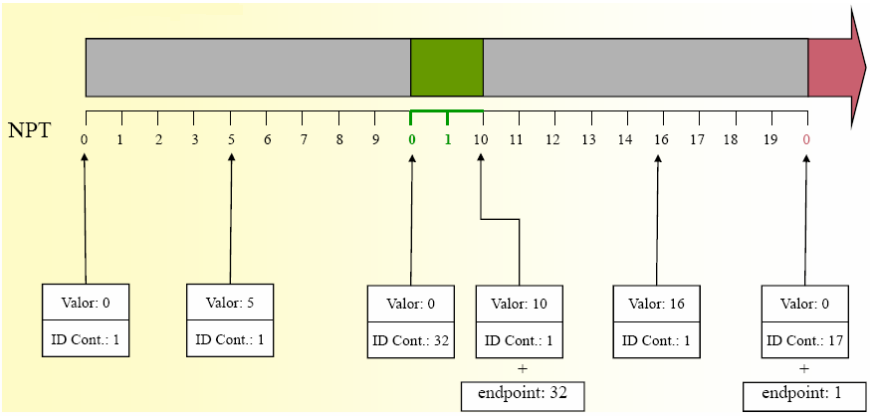


Figura 1.13.- *Esquema donde se representan los distintos tipos de descriptores*

Una vez que se conoce el funcionamiento del NPT surge otra duda: ¿cómo se accede desde las aplicaciones al NPT de un determinado flujo?

Para ello se usan los *Stream Events*, que son marcas que se difunden y permiten al receptor (y, por tanto a las aplicaciones) sincronizarse con puntos específicos de los flujos de contenidos. Estos puntos específicos son los valores NPT.

Desde la perspectiva de DSM-CC, los *Stream Events* se dividen en dos partes: los objetos DSM-CC *Stream Events* y los descriptores de *Stream Event*. Los objetos *DSM-CC Stream Events* se almacenan en un *object carousel* y son como cualquier otro objeto DSM-CC. Un objeto *Stream Event* describe los eventos permitidos en el servicio correspondiente a ese carrusel, para ello incluye un identificador de evento (que debe ser único dentro de ese carrusel) y el nombre del *Stream Event*.

Por otro lado se tienen los descriptores, estos se insertan en el flujo MPEG como un marcador y le informan al receptor de que un evento ha sido creado. Cada descriptor de *Stream Event* tiene tres partes:

- ID del *Stream Event*
- Valor NPT
- Datos específicos para la aplicación

La ID informa al receptor de cual de los objetos *Stream Event* está asociado a ese descriptor. Cada aplicación se suscribe al tipo de evento que le interesa con la ID, el *midleware*[3] sólo avisa a las aplicaciones que están suscritas.

Existen dos tipos de descriptores de Stream Event:

- Do-it-now Stream Events
- Scheduled Stream Events

[3]*Midleware*: Es la interfaz MHP que hace de intermediario entre las aplicaciones y la red de difusión

Do-it-now Stream Event

Si un terminal MHP recibe un *Do-it-now Stream Event*, entregará el *Stream Event* y su carga útil (los datos específicos para la aplicación que viajan en el descriptor) de inmediato a la aplicación MHP. Este tipo de descriptores sólo se envían una vez, por lo que si el receptor lo pierde no hay forma de recuperarlo, por este motivo el receptor debe adoptar las medidas necesarias para asegurarse de que se recibe correctamente. Uno de los beneficios de estos descriptores es que pueden utilizarse para sincronizar programas en vivo, donde no se sabe de antemano en qué instante se va a necesitar enviar un evento. Por ejemplo, durante un partido de fútbol se puede mostrar la ficha del jugador que acaba de marcar el gol. En este caso a priori no se sabe en que minuto del partido se va a marcar un gol ni quién va a ser el autor, lo que se hace es enviar un *Stream Event* cuando se marca el gol con el nombre del jugador que ha marcado en el campo de datos del *Stream Event*.

Los *Do-it-now Stream Events* deben tener un identificador que esté comprendido en el rango que va entre 0x0001 y 0x3fff.

Scheduled Stream Events

En este caso cada descriptor tiene asociado un valor de NPT que informa de cuando debe ser activado el evento (para los *Do-it-now* este campo es siempre -1). Esto permite al receptor conocer con cierta antelación que tiene que activar un evento cuando el NPT alcance un valor determinado. Además este tipo de descriptores también añaden un poco más de fiabilidad ya que se pueden enviar varios descriptores con los mismos valores con el fin de asegurarse de que al menos uno de ellos es decodificado correctamente. Las especificaciones de MHP nos dicen que es obligatorio reenviarlo al menos 5 veces en los 5 segundos previos a su ejecución.

El problema de estos descriptores es que sólo se pueden usar en programas pre-grabados, ya que se debe conocer con anterioridad el momento en que debe ser ejecutado el evento para poder incluirlo en el descriptor. Para la programación en directo, el momento exacto de ejecución no se conoce y, por lo tanto, no es posible especificar una fecha y hora en el descriptor. Por otro lado la transmisión de la información por adelantado puede resultar problemática. Si, por ejemplo, se están generando eventos para un concurso de preguntas y respuestas, la información acerca de la respuesta correcta es enviada en el descriptor, y está disponible en el receptor antes incluso de que se formule la pregunta. De este modo, un jugador podría manipular el receptor y extraer la solución a las preguntas

antes de que estas se formulen y así ganar el concurso. Para evitar estos problemas la mejor opción es utilizar los descriptores *Do-it-now*.

El identificador de los *Scheduled Stream Event* debe pertenecer al rango comprendido entre 0x8000 y 0xbfff.

Procesamiento de los *Stream Events*

Cuando llega un descriptor de evento de flujo:

1. Se comprueba si existe el ID del evento en los objetos *Stream Event* del carrusel de objetos por defecto del servicio. Si no existe se ignora el descriptor de evento.
2. Si es un descriptor *Do-it-now* se dispara el evento de forma inmediata.
3. Si no es *Do-it-now*, se comprueba su NPT asociado: si ya existe un evento con el mismo ID y NPT (es un reenvío) o si el NPT ya ha pasado se ignora el descriptor de evento.
4. Si no estamos en el caso anterior, cuando el valor de NPT correspondiente alcanza el valor especificado en el descriptor de evento, se dispara el evento.

Una de las ventajas que ofrece la separación de los *Stream Events* en objetos y descriptores es que permite reutilizar los eventos. Varios descriptores pueden contener el mismo identificador de evento, incluso si se activan en momentos diferentes pueden contener distintos datos. Esto permite a una aplicación usar el identificador de evento para definir distintos tipos de eventos. Por ejemplo, siguiendo con el ejemplo del fútbol, los descriptores que tengan el identificador 1 pueden indicar que se ha marcado un gol, los que lleven el identificador 2 que se ha cometido una falta o el identificador 3 puede indicar el final del partido. Por lo tanto, una aplicación puede iniciar la tramitación de un evento conociendo únicamente el identificador de evento.

Uso de los *Stream Events* en aplicaciones

Para una aplicación MHP, los objetos *DSM-CC Stream Events* están representados por la clase ***org.dvb.dsmcc.DSMCCStreamEvent*** (para representar objetos *Stream Event*), que es una subclase de ***org.dvb.dsmcc.StreamEvent*** (para representar *Stream Event* concretos, es decir, los descriptores que debe escuchar el *listener*).

A través del ***DSMCCStreamEvent*** una aplicación se suscribe a determinados tipos de eventos y se define un *listener* para recibirlos. A partir de este momento el *middleware* notificará a través del *listener* los eventos concretos en el instante adecuado mediante objetos ***Stream Event***.

La clase ***DSMCCStreamEvent***:

```
public class DSMCCStreamEvent extends DSMCCStream
{
     public DSMCCStreamEvent(DSMCCObjectaDSMCCObject);
     public DSMCCStreamEvent(java.lang.String path);
     public DSMCCStreamEvent(java.lang.String path,
           java.lang.Stringname);

     String[] getEventList();

     int subscribe(java.lang.StringeventName, StreamEventListener l);
     void unsubscribe(inteventId, StreamEventListener l);
     void unsubscribe(java.lang.String eventName,
            StreamEventListener l);
}
```

El método ***getEventList()*** permite a la aplicación obtener una lista de los nombres de los eventos que son válidos en ese momento.

Una vez conocida la lista de eventos, se puede elegir el evento al que se quiere suscribir, para ello se usa el método ***subscribe()***. La solicitud deberá especificar el nombre del evento al que se desea suscribir, y el *listener* que será notificado cuando se reciban los eventos. El oyente debe aplicar la interfaz ***org.dvb.dsmcc.StreamEventListener***.

Una aplicación puede darse de baja usando el método ***unsubscribe()***, se usará un constructor u otro dependiendo de si el parámetro de entrada es el nombre o el identificador del evento.

La clase ***Stream Event***:

```
public class StreamEvent extends java.util.EventObject
{
     public StreamEvent(DSMCCStreamEvent source, long npt, String name,
            int eventId, byte[] eventData);

      int getEventId();
      String getEventName();
      byte[] getEventData();
      long getEventNPT();
      Object getSource();
}
```

El método ***getEventData()*** devuelve los datos específicos que se incluyeron en el descriptor. Por otro lado, el método ***getSource()*** devuelve el objeto ***org.dvb.dsmcc.DSMCCStreamEvent*** al que está asociado. Esto da facilidades a la hora de anular una suscripción a un evento o suscribirse a otros eventos que puedan estar relacionados. [MOR 05]

1.3.2.- Programación de una aplicación MHP

Tras esta introducción teórica a la sincronización de aplicaciones haciendo uso de los *Stream Events*, se va a describir la aplicación sincronizada que se ha creado para mostrar el funcionamiento de estos elementos de sincronismo. Se trata de una aplicación muy sencilla con la que se muestra el nombre del vídeo que se está reproduciendo en un *banner* en la parte inferior de la pantalla. El funcionamiento es muy sencillo, cada vez que comienza la reproducción de un vídeo se envía un *Stream Event* con el nombre del vídeo en el campo de datos, al recibir este evento la aplicación muestra un *banner* con la información del campo de datos durante 10 segundos, después de este tiempo el *banner* desaparece.

A continuación se va a describir cómo se ha implementado la aplicación MHP que se ejecuta en el receptor. Las aplicaciones interactivas reciben el nombre de ***Xlets*** y se transmiten de forma cíclica. Aunque estos ***Xlets*** están implementados en Java no siguen el modelo convencional de las aplicaciones Java en el que cada aplicación tiene el control completo de su ciclo de vida y de los recursos de la maquina virtual sobre la que se ejecutan. En cambio, se podria decir que se parecen a los ***applets***[4], puesto que varias aplicaciones pueden estar ejecutándose a la vez y no tienen por qué iniciarse a través de línea de comandos (en el caso de los ***applets***, pueden ser iniciados por el navegador web). Esta diferencia se puede ver resumida en la **Tabla1** que se incluye más abajo.

[4] Un ***applet*** es un componente de una aplicación que se ejecuta en el contexto de otro programa, por ejemplo un navegador web. A diferencia de un programa, un ***applet*** no puede ejecutarse de manera independiente, ofrece información gráfica y a veces interactúa con el usuario, típicamente carece de sesión y tiene privilegios de seguridad restringidos. Un Java ***applet*** es un código Java que carece de un método main, por eso se utiliza principalmente para el trabajo de paginas web, ya que es un pequeño programa que es utilizado en una página HTML y representado por una pequeñe pantalla gráfica dentro de ésta.

	Xlet	main() application
Ciclo de Vida	Loaded, Paused, Active y Destroyed	main y exit
Num. de aplicaciones en VM	Multiples aplicaciones Xlet	Básicamente una aplicación por VM

Tabla 1.- *Comparación entre una aplicación Java convencional y un Xlet*

Por supuesto, hay diferencias entre el mundo de la televisión digital y el de la web lo que hace que no se puedan ejecutar los ***applets*** en un receptor de TDT. La evolución de estos ***applets*** para que sean compatibles con la tecnología utilizada en TDT es a lo que hoy se llama ***Xlet***. Este concepto fué introducido por Sun en la especificación de JavaTV y adoptado como el formato de aplicaciones Java para MHP. Como en el caso de los ***applets***, el interfaz ***Xlet*** permite a una fuente externa (***ApplicationManager*** o Gestor de aplicaciones en el caso de un receptor de televisión digital) iniciar y detener una aplicación. El interfaz ***Xlet*** se encuentra en el `paquete javax.tv.xlet`.

```
public interface Xlet {

      public void initXlet(XletContext ctx)
         throws XletStateChangeException;

      public void startXlet()
         throws XletStateChangeException;

      public void pauseXlet();
      public void destroyXlet(boolean unconditional)
         throws XletStateChangeException;
}
```

Si se compara esta definición con la clase `java.applet.Applet` se aprecian algunas similitudes. Al igual que en la clase ***applet***, tiene métodos que permiten al gestor de aplicaciones iniciarlos, arrancarlos y pararlos. Pero también hay diferencias entre estas dos clases.

En primer lugar, y quizás la más significativa, es que a diferencia de los ***applets*** la ejecución de un ***Xlet*** tambien puede ser pausada y reanudada. La razón es muy sencilla, en un entorno como un receptor de TDT varias aplicaciones pueden estar ejecutandose a la vez, pero debido a resticciones impuestas por el hardware sólo una de ellas puede estar visible, es decir, que las aplicaciones que no estén visibles deben ser pausadas para que liberen los recursos y estos puedan ser utilizados por la aplicación visible.

Otra de las diferencias es que un ***Xlet*** es mucho más simple que un ***applet***, y por tanto, los ***Xlet*** de alguna forma están mas limitados en lo que pueden hacer cuando interatúan con su entorno.

En un ***Xlet*** queda claramente definido un ciclo de vida (**Diagrama 1.32**) que es controlado por el ***XletManager*** (Controlador del Ciclo de Vida). Los diferentes estados por los que puede pasar un ***Xlet*** son: ***Loaded***, ***Paused***, ***Started*** y ***Destroyed***.

- Estado Inicializado (***Loaded***): se genera una instancia de la aplicación ***Xlet***, pero no se inicializa.
- Estado Pausado (***Paused***): el ***Xlet*** es inicializado y pausado con el mínimo uso de recursos.
- Estado Arrancado (***Started***): el ***Xlet*** es ejecutado normalmente proporcionando su servicio.
- Estado Destruido (***Destroyed***): el ***Xlet*** se destruye así mismo y libera sus recursos. Este estado sólo se alcanza una vez.

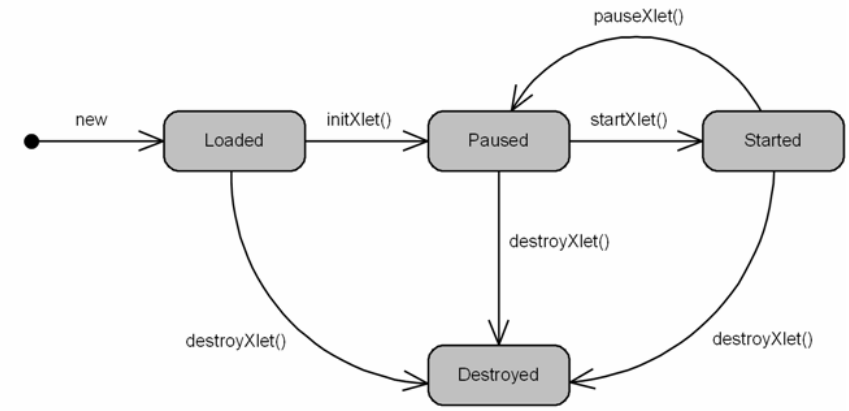


Diagrama 1.32.- *Ciclo de vida de un Xlet*

El gestor de aplicaciones carga el archivo con la clase principal del ***Xlet*** (como se indicó desde el emisor) y crea una instancia del ***Xlet*** llamando al constructor por defecto. En este momento, el ***Xlet*** alcanza el estado Cargado (***Loaded***). [MOR 05]

El ***Xlet*** puede ser iniciado de 3 formas:

- Por indicación directa del usuario
- Invocado por otra ***Xlet***
- Automático por configuración en la AIT (Tabla de información de las aplicaciones)

Cuando se ordena el inicio de la aplicación el gestor de aplicaciones (***ApplicationManager***) en el receptor llama al método ***initXlet()***, pasando un nuevo objeto ***XletContext*** como parámetro de entrada para el ***Xlet***.

El ***Xlet*** puede usar este ***XletContext*** para inicializarse, y para precargar datos de gran tamaño como imágenes que pueden requerir de tiempo para cargarse desde el *object carousel*. Cuando la inicialización se ha completado, el ***Xlet*** se queda en estado Pausado (***Paused***) y puede comenzar inmediatamente.

Una vez que el método ***initXlet()*** ha terminado, el gestor de aplicaciones llama al método ***startXlet()***. Esto provoca que el ***Xlet*** pase del estado Pausado al estado Arrancado (***Started***), y el ***Xlet*** podrá interactuar con el usuario.

Durante la aplicación el gestor de aplicaciones puede llamar al método ***pauseXlet()***. Esto hará que la aplicación pase del estado Arrancado al estado Pausado. La aplicación puede volver a ejecutarse de nuevo llamando al método ***startXlet()***. Durante la vida del ***Xlet*** se pueden producir varios cambios de estado entre Pausado y Arrancado.

Al final de su vida, el gestor de aplicaciones llamará al método ***destroyXlet()***, esto hará que el ***Xlet*** pase al estado Destruido (***Destroyed***) y libere todos sus recursos. Como ya se ha comentado antes, este estado sólo se puede alcanzar una vez, de modo que una vez que el ***Xlet*** pasa al estado Destruido no se puede volver a ejecutar esa instancia del ***Xlet***.

A continuación, en el **Diagrama 1.33**, se incluye un diagrama de secuencia del ciclo de vida del ***Xlet*** y los estados por los que transita.

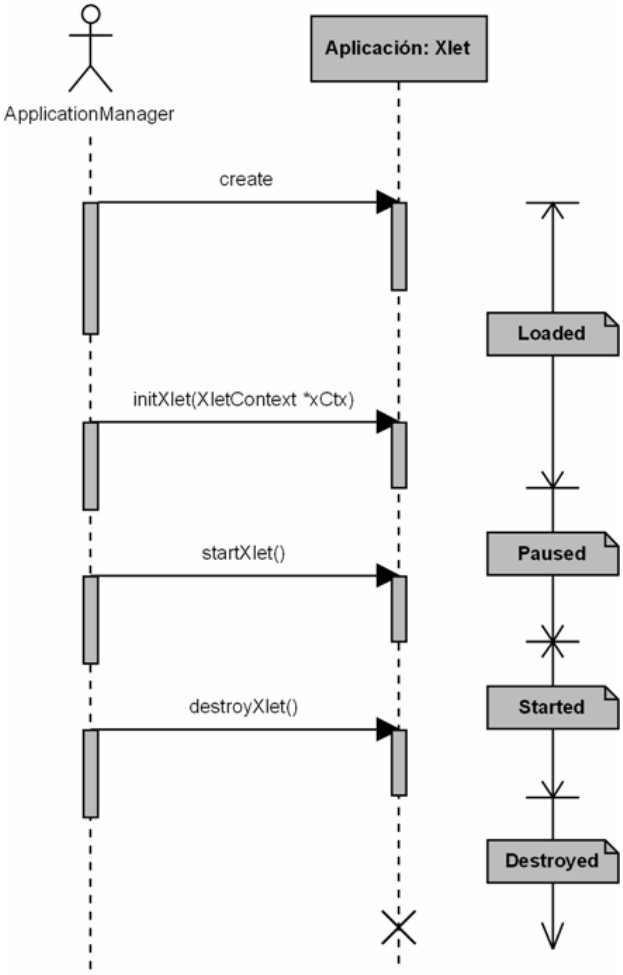


Diagrama 1.33.- *Diagrama de secuencia de un Xlet*

Como ya se ha visto, la interfaz ***Xlet*** define 4 métodos abstractos pero no los implementa (***initXlet()***, ***startXlet()***, ***pauseXlet()*** y ***destroyXlet()***), será la aplicación la que sobrescriba estos métodos para implementar la interfaz ***Xlet***. Estos métodos son los encargados de ordenar las transiciones entre estados, y se implementan teniendo en cuenta las caracteristicas de la aplicación. A continuación se va a detallar en que consiste realmente la aplicación ejemplo que se ha desarrollado y también cómo ha sido implementada. En primer plano, se puede ver el **Diagrama 1.34** que representa el diagrama de flujo de esta aplicación, y que se puede utilizar como un pequeño resumen de la implementación que se va a detallar a continuación.

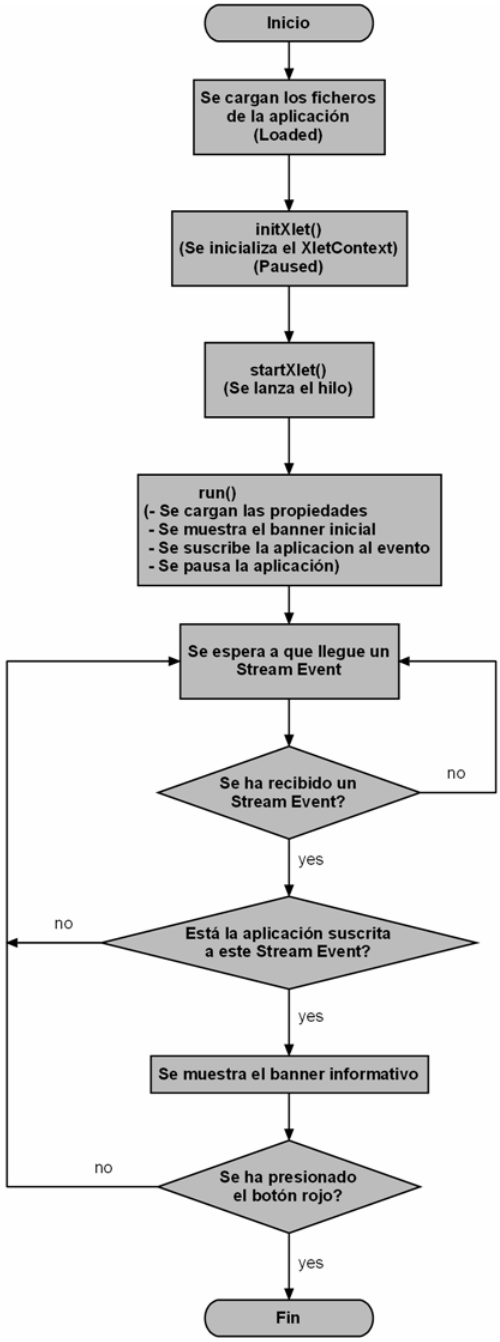


Diagrama 1.34.- *Diagrama de Flujo de la aplicación MHP de sicronización*

Como se puede ver en el **Diagrama 1.32** el ciclo de vida de un ***Xlet*** comienza en el estado ***Loaded*** para ello lo primero que se debe hacer es cargar los ficheros de la aplicación. Una vez que la aplicación se encuentra en este estado, ya puede ser lanzada por el usuario, en ese momento se ejecuta el método ***initXlet()***. Este método se encarga de inicializar el ***XletContext*** y pasa la aplicación del estado en que estaba (***Loaded***) al estado ***Paused***. A continuación se lanza el método ***startXlet()***, con este método se realiza la

transición al estado ***Started***, en este caso lo que hace este método es lanzar el hilo de la aplicación. Para conseguir la sincronización de la aplicación con los contenidos es necesario que ésta escuche los eventos en el momento que llegan, por esa razón se ha implementado un hilo que estará corriendo durante todo el ciclo de vida de la aplicación a la espera de los ***Stream Events***. De este modo al pausar la aplicación sólo se liberan los recursos de la parte gráfica pero la aplicación sigue corriendo en un segundo plano no visible para el usuario.

Cuando se recibe un ***Stream Event*** se ejecuta el método ***receiveStreamEvent()***, este método comprueba si la aplicación está suscrita a ese evento, si no está suscrita lo deja pasar sin hacer nada, pero si coincide con el evento al que se ha suscrito se mostrará por pantalla el banner informativo durante 10 segundos, y pasado este tiempo la aplicación volverá al estado ***Paused***.

Por último, si se quiere matar la aplicación se debe presionar el botón rojo del mando a distancia. El encargado de escuchar los eventos recibidos en el receptor provenientes del mando a distancia es el método ***userEventReceiver()***. Es esta aplicación sólo se ha configurado el caso en el que se presiona el boton rojo del mando a distancia ya que la aplicación no requiere el uso de ningún otro botón.

A continuación, se detalla la implementación de los métodos que se han mencionado en la explicación anterior. Se va a comenzar por los 4 métodos abstractos que define la interfaz ***Xlet***.

Método: ***void initXlet(XletContext *xcContext)***

- *Parámetros de entrada:*
  XletContext *xcContext → Puente entre el ***Xlet*** y el *Application Manager*
- *Salida:* ***void***

Como ya se ha comentado, con este método se realiza la transición del estado ***Loaded*** al estado ***Paused*** y por tanto es el primer método que se ejecuta cuando se lanza la aplicación.

```
public void initXlet(XletContext xcContext) throws
      XletStateChangeException
{
      this.xcContext = xcContext;
```

```
        boFirstStart = true;
        System.out.println("The initXlet() method has been called.
              Our Xlet context is " + xcContext + " Date " +
              (new Date()).getTime());
}
```

En este método se inicializa el ***XletContext***, éste hace de puente entre el ***Xlet*** y el *Application Manager*.Es decir, permite que el *Application Manager* sea correctamente notificado de cualquier cambio en los estados de las ***Xlets***.

También se inicializa la variable ***booleana boFirstStart***, esta variable se utiliza para controlar que la aplicación no vuelva a ser lanzada mientras su ciclo de vida no se complete.

Método: ***void startXlet()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

Con este método se realiza la transición entre el estado ***Paused*** y el estado ***Started***. Dadas las caracteristicas de la aplicación, este método solo se ejecuta una vez, para ello se utiliza la variable de control ***boFirstStart***, si esta variable tiene como valor ***false*** el método no hace nada. Si por el contrario su valor es ***true***, se considera que es la primera vez que se llama a este método y se lanza el hilo que se va a encargar de mantener corriendo la aplicación durante todo su ciclo de vida.

```
public void startXlet() throws XletStateChangeException
{
      if (boFirstStart)
      {
            System.out.println("The startXlet() method has been
                  called to start the Xlet for the first time.  Hello,
                  world!" + " Date " + (new Date()).getTime());
            thStartThread =  new Thread(this);
            thStartThread.start();
      }
}
```

Método: ***void pauseXlet()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

Con este método se pasa la aplicación al estado ***Paused***, en este método la aplicación libera los recursos gráficos y continúa ejecutandose en un segundo plano a la espera de la llegada de un ***Stream Event***. Para liberar los recursos, cuando se llama al método ***pauseXlet()***, se libera la pantalla haciendo invisible la escena de la aplicación.

```
public void pauseXlet()
{
      hsScene.setVisible(false);
}
```

Método: ***void destroyXlet(boolean *boDestroy)***

- *Parámetros de entrada:*

  XletContext *xcContext → Puente entre el ***Xlet*** y el *Application Manager*
- *Salida:* ***void***

La aplicación puede ser terminada en cualquier momento evocando al método ***destroyXlet()***. En este caso, el método es llamado por la aplicación pero también podría ser llamado por el *Application Manager* (éste se encarga de manejar los estados de cada aplicación y hacer de puente a los recursos del *Set-Top Box*).

```
public void destroyXlet(boolean boDestroy) throws
      XletStateChangeException
{
      if (boDestroy)
      {
            System.out.println("destroyXlet() called");
            hsScene.setVisible(false);
            thStartThread = null;
            prEventProp = null;
            xcContext.notifyDestroyed();
      }
      else
      {
            System.out.println("destroyXlet() called, but I don't
                  want to die, I'm waiting for streamevents.");
      }
}
```

Como se puede ver en las líneas de código del método, para terminar la aplicación en primer lugar se hace invisible la escena, para liberar los recursos de los que hace uso la aplicación, y se mata el hilo de la aplicación, por último se notifica al ***XletContext*** que se va a destruir la aplicación.

Método: ***run() → thStartThread***

Como se acaba de comentar la aplicación corre sobre un hilo que se lanza desde el método ***startXlet()***, de modo que se debe implementar un método ***run()*** que defina lo que va a hacer la aplicación una vez que se haya arrancado el hilo.

```
public void run()
{
      if (boFirstStart)
      {
            prEventProp = PropertyCharger("data/event.properties");
            uerRepos = new UserEventRepository("Repos");
            uerRepos.addAllColourKeys();
            uerRepos.addAllArrowKeys();
            getResources();
            boFirstStart = false;
            if (prEventProp != null)
            {
                  dsoObject = new DSMCCObject
                        (prEventProp.getProperty("EventPath"));
                  try
                  {
                        dsoObject.synchronousLoad();
                  }

            […]

                  try
                  {
                        dseEvent = new DSMCCStreamEvent(dsoObject);
                  }

            […]

                  try
                  {
                        iSubscribedEventID=dseEvent.subscribe
                              (prEventProp.getProperty("EventName"), this);
                  }

            […]

            }
            else
            {
                  try
                  {
```

```
                    destroyXlet(true);
                }
                catch (XletStateChangeException xsce)
                {
                    xsce.printStackTrace();
                }
                return;
            }
            Date dtNow = new Date();
            System.out.println("Starting countdown of 10 seconds "+
                (new Date()).getTime());
            while (new Date().before(new Date(dtNow.getTime()+10000)))
            {}
            pauseXlet();
        }
        else
        {
            return;
        }
}
```

Se puede ver que al igual que en el método ***startXlet()*** se comprueba que es la primera vez que se accede a este método con la variable ***boFirstStart***.

En primer lugar se cargan las propiedades del *Stream Event* en el fichero de propiedades con el método ***PropertyCharger***, y se añaden las teclas del mando al repositorio para que puedan ser escuchadas por la aplicación.

A continuación, se llama al método ***getResource()*** que se encarga de crear la escena donde se representa el interfaz gráfico de la aplicación. En este momento se muestra por pantalla el *banner* inicial de la aplicación, este *banner* es meramente informativo, indica que la aplicación se ha iniciado con éxito y que está listo para recibir los eventos.

Se cambia el valor de la variable ***boFirstStart*** a ***False*** para evitar que se vuelva a lanzar la aplicación.

Si el objeto ***Property*** que ha devuelto el método ***PropertyCharger*** no es nulo se hace la carga del objeto *DSM-CC* que previamente se ha guardado en la variable ***dsoObject*** con la función ***dsoObject.synchronousLoad()***. A continuación se guarda el objeto ***DSM-CC Stream Event*** que se obtiene de la variable ***dsoObject*** en la variable ***dseEvent***.

Por último, la aplicación se suscribe al *Stream Event* con el método ***subscribe()*** que ya se comento cuando se describió la clase ***DSMCCStreamEvent***, y se pausa la aplicación llamando al método ***pauseXlet()***.

En esta aplicación se hace uso de un fichero de propiedades, pero se podría eliminar éste y definir las propiedades directamente como constantes.

Método: ***Properties PropertyCharger(String *txProperties)***

- *Parámetros de entrada:*
  String *txProperties → *Path* donde se encuentra el fichero de propiedades
- *Salida:* ***Properties*** → Objeto ***Properties*** donde se cargan las propiedades

En el método ***PropertyCharger()*** se carga el fichero de propiedades donde se guarda información relativa al *Stream Event*, como puede ser el *path*, el nombre,… De esta forma cada vez que se necesite alguna de estas propiedades se tomará directamente del objeto ***Properties*** que se crea en este método.

```
public static Properties PropertyCharger(String txProperties)
{
	Properties prProperties = new Properties();
	InputStream isGeneral = null;
	try
	{
		isGeneral = new FileInputStream((new File(new
			File(txProperties).getCanonicalPath())));
		prProperties.load(isGeneral);
	}
	catch (IOException e)
	{
		System.out.println("Input/Output error while charging
			properties: " + e.toString() + (new Date()).getTime());
	}
	finally
	{
		try
		{
			if(isGeneral!= null)
			{
				isGeneral.close();
			}
		}
		catch(IOException e)
		{
			System.out.println("Input/Output error in
				finally when we wanted to close the
				InputStream: " + e.toString() +
				(new Date()).getTime());
		}
	}
	return prProperties;
}
```

Método: ***void getResources()***

- *Parámetros de entrada:* Ninguno
- *Salida:* ***void***

En el método ***getResources()*** se crea la escena donde se representa el interfaz gráfico de la aplicación. En este caso, se trata de un *banner* en la parte inferior de la pantalla, como ya se ha comentado anteriormente.

```
public void getResources()
{
      HSceneFactory hsfFactory = HSceneFactory.getInstance();
      HSceneTemplate hstTemplate = new HSceneTemplate();
      hstTemplate.setPreference(
            HSceneTemplate.SCENE_SCREEN_DIMENSION,
            new org.havi.ui.HScreenDimension(1,1),
            HSceneTemplate.REQUIRED);
      hstTemplate.setPreference(
            HSceneTemplate.SCENE_SCREEN_LOCATION,
            new org.havi.ui.HScreenPoint(0,0),
            HSceneTemplate.REQUIRED);

      hsScene = hsfFactory.getBestScene(hstTemplate);

      rRect = hsScene.getBounds();
      setBounds(rRect);
      setVisible(true);

      hsScene.add(this);

     hstText = new
            HStaticText(prEventProp.getProperty("InitialText"),100,350,
            520,150,new Font("Tiresias",Font.PLAIN,26),Color.black,new
            DVBColor(255,172,255,128),new HDefaultTextLayoutManager());
      hstText.setVisible(true);
      hsScene.add(hstText);
      hstText.setBackgroundMode(HStaticText.BACKGROUND_FILL);
      hsScene.setVisible(true);
      emManage.addUserEventListener(this, uerRepos);

      this.requestFocus();
}
```

Para crear la escena se usa la clase ***HSceneFactory***, esta clase es como una fábrica de ***HScene***. Estos objetos son como los componentes raíz de cada aplicación y sustituyen al ***Frame*** o el ***Window*** para PC. Una vez creada la escena vacía, se le asignan las preferencias utilizando las plantillas ***HSceneTemplate***.

Método: ***void receiveStreamEvent(StreamEvent *seEvent)***

- *Parámetros de entrada:*

  StreamEvent *seEvent → Evento que se ha recibido

- *Salida:* ***void***

Por último se tiene el método ***receiveStreamEvent()*** que implementa la interfaz del ***Stream Event Listener***. Es decir, gestiona la recepción de los descriptores de *Stream Event*.

```
public void receiveStreamEvent(StreamEvent seEvent)
{
      int iEventID = seEvent.getEventId();
      if (iEventID == iSubscribedEventID)
      {
            byte[] arrbyDatos = seEvent.getEventData();
            if (arrbyDatos != null)
            {
                  for (int i = 0; i < arrbyDatos.length;  i++)
                  {
                        System.out.println(arrbyDatos[i]);
                  }
                  if (arrbyDatos.length > 0)
                  {
                        hstText.setTextContent(new String(arrbyDatos),
                              HStaticText.NORMAL_STATE);
                  }
                  else
                  {
                        hstText.setTextContent(new String("He recibido
                              un Stream Event pero no datos"),
                              HStaticText.NORMAL_STATE);
                  }
            }
            else
            {
                  hstText.setTextContent(new String("He recibido un
                        Stream Event pero no datos"),
                  HStaticText.NORMAL_STATE);
            }
            hstText.repaint();
            hsScene.setVisible(true);
            Date dtNow = new Date();
            System.out.println("Starting countdown of 10 seconds "+
                  (new Date()).getTime());
            while (new Date().before(new Date(dtNow.getTime()+10000)))
            {}
            pauseXlet();
      }
}
```

En este método se gestiona la recepción de los descriptores de *Stream Events*. En primer lugar, se comprueba si la aplicación está suscrita a ese *Stream Event*, si no es así se ignora el descriptor y la aplicación no hace nada. Si por el contrario, la aplicación si que está suscrita, se reacciona a la llegada del descriptor. Primero se comprueba si contiene datos, y si es así se guardan en la variable ***arrbyDatos***. A partir de aquí se pueden dar dos casos:

- El descriptor contiene datos: se muestran los datos por pantalla, en este caso debería ser el nombre del vídeo.
- El descriptor no contiene datos: se saca por pantalla el siguiente texto: "He recibido un *Stream Event* pero no datos".

Después de esto se llama al método ***pauseXlet()*** que pasa el *Xlet* al estado de pausa, donde se queda hasta que reciba otro *Stream Event* o hasta que se finalice la aplicación.

Método: ***void userEventReceived(UserEvent *ueEvent)***

- *Parámetros de entrada:*
  UserEvent *ueEvent → Evento recibido al pulsar un botón del mando
- *Salida:* ***void***

En este método se configuran las acciones que debe realizar la aplicación al recibir los eventos que provoca el usuario al presionar las teclas del mando a distancia. Para esta aplicación sólo se va a necesitar configurar el botón rojo que el usuario presionará si quiere terminar la aplicación.

```
public void userEventReceived(UserEvent ueEvent)
{
      switch(ueEvent.getCode())
      {
            case HKeyEvent.VK_COLORED_KEY_0:
            {
                  // Red button is pressed
                  try
                  {
                        destroyXlet(true);
                  }
                  catch (XletStateChangeException xsce)
                  {
                        xsce.printStackTrace();
                        System.out.println("Error while destroying " +
                              (new Date()).getTime());
                  }
                  break;
```

```
            }
            default:
            {
                  // do nothing
                  System.out.println("The event code is: " +
                        ueEvent.getCode());
                  break;
            }
      }
}
```

Como se puede ver, si se recibe un evento que indica que se ha presionado el botón rojo, se llama al método ***destroyXlet()*** para finalizar la aplicación.

1.3.3.- Programación de Stream Events

Ya se ha descrito la aplicación que se ejecuta en el receptor, pero para que esta aplicación reciba los *Stream Events* que la hacen funcionar se deben enviar los descriptores desde el servidor en los momentos adecuados.

En este caso se van a utilizar *Stream Events* de tipo *Do it now*, ya que el laboratorio con el que se está trabajando no soporta los *Schedule Stream Event*. Como se quiere que el *banner* aparezca al inicio de cada vídeo se debe enviar el *Stream Event* cuando se active la configuración en el *thread* de reproducción.

```
dTime = new Date();
lTime = dTime.getTime();
while(lTime < ts2PlayNow.getDFrom().getTime())
{
      dTime = new Date();
      lTime = dTime.getTime();
}
scActiv.activate(true);

WindowCreateSchduledTS.dtmScheduler.removeRow(0);

//Se envia el Stream Event
ulse.sendSEItem(tsTstream.getName(),"anuncio",
      "/Sincro/data/StreamEvent");
```

Para enviar los *Stream Events* se utiliza el método ***sendSEItem()*** que pertenece a la clase ***UploadSE***.

Método: ***void sendSEItem(String *txData, String *txName, String *txSE)***

- Parámetros de entrada:
  - ***String *txData*** → Datos que se van a incluir en el campo de datos del *Stream Event*.
  - ***String *txName*** → Nombre del evento
  - ***String *txSE*** → *Stream Event* guardado en el carrusel
- Salida: ***void***

En primer lugar se separa el ***String*** de datos en Bytes y se guardan en un *array* de Bytes, esto se hace porque el campo de datos de los *Stream Events* sólo admite la información en Bytes y no en ***String***.

A continuación se obtiene el evento del carrusel, y se busca el *Item* que lleve el nombre que se ha pasado en el parámetro de entrada, para enviarlo.

```
try
{
	arrByData = txData.getBytes();
	arrCsei = cCarrusel.getCarouselStreamEvent(txSE)
		.getCarouselStreamEventItems();
	for (int i=0; i<arrCsei.length; i++)
	{
		if (arrCsei[i].getEventName().equals(txName))
		{
			arrCsei[i].sendEvent(arrByData, lTime);
		}
	}
}
catch (SecurityException pste)
{
	pste.printStackTrace();
}
catch (IOException ioe)
{
	ioe.printStackTrace();
}
```

1.4.- Programación gráfica

Para la programación de la parte gráfica de la aplicación se ha utilizado el paquete Swing, que es una biblioteca gráfica para Java. Esta biblioteca es parte de la JFC (Java Foundation Classes) en la plataforma Java. [ROB 03] [LOY 02]

Este paquete sustituye a las antiguas librerías AWT que eran las utilizadas por Java hasta la versión 1.2. Las nuevas librerías aparte de tener una mayor cantidad de opciones sobre los componentes (como distintas apariencias, control sobre el *focus*, mayor número de campos que modifican su aspecto,…) se diferencian de las anteriores sobre todo en su implementación.

En AWT cuando se añade un botón, por ejemplo, el compilador genera código que le pide al sistema operativo la creación de un botón en un determinado sitio y con unas determinadas propiedades. En Swing ya no se pide nada al sistema operativo, directamente se dibuja el botón en la ventana. Con esto se eliminaron muchos problemas que existían antes con los códigos de las interfaces gráficas, que debido a que dependían del sistema operativo para obtener sus componentes gráficos, era necesario probar los programas en distintos sistemas operativos. Con Swing se mejoró mucho este aspecto, ya que lo único que se le pide al sistema operativo es una ventana y una vez que se tiene esta ventana se dibuja sobre ella todo lo que se necesite. Además el hecho de que el botón no sea un botón del sistema operativo, sino un botón pintado por Java da mayor control sobre su apariencia. Para diferenciar los componentes Swing de los AWT, sus nombres están precedidos por una “J”.

Los programas Swing necesitan importar el paquete principal de Swing, todas las clases Swing se encuentran en el paquete “**javax.swing.**”. Además de este paquete, también se necesitan clases de los paquetes principales de AWT. Por lo tanto en las primeras líneas se importan los paquetes necesarios:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

Para describir la implementación de la parte gráfica se va a cambiar el esquema seguido en este capítulo. Hasta ahora se ha descrito con detalle la implementación de cada funcionalidad, pero en este caso ese nivel de detalle puede resultar pesado ya que en esta parte (al tratarse de programación gráfica) va a haber muchos elementos que se repitan a lo largo de las distintas ventanas. En este caso se va a describir la estructura de la librería

Swing y a partir de esta estructura general se irán describiendo los elementos más relevantes utilizados en la implementación de esta aplicación.

Los elementos del paquete Swing se van a agrupar en 4 conceptos:

- **Eventos:** Son las clases o interfaces que permiten crear objetos que capturen y manejen los eventos. Un evento es una acción sobre algún componente, por ejemplo, hacer clic sobre un botón. Sin los eventos las GUI (*Graphics User Interface*) serían interfaces gráficas sin vida, es decir, el usuario no podría interactuar con ellas. En esta aplicación los eventos más utilizados son los asociados a las acciones sobre los botones, ya que en todas las ventanas de la aplicación hay más de un botón que el usuario puede utilizar para interactuar con la aplicación.
- **Layouts (Administradores de diseño):** Son clases o interfaces que permiten crear objetos que administren el diseño, la distribución y colocación de los objetos de tipo componente dentro de los objetos de tipo contenedor.
- **Componentes:** Son clases o interfaces que permiten crear los objetos gráficos que componen una GUI tales como botones, listas desplegables, campos de texto, etiquetas,… En esta aplicación se hace uso de componentes como los campos de texto que se pueden encontrar en la ventana de inicio, en los cuales el usuario debe introducir los datos de identificación para acceder a la aplicación, en otras ventanas se han utilizado campos de texto no editables, que realizan una función meramente informativa, como puede ser el caso del campo de texto que informa del vídeo que se encuentra en reprodución en la ventana de configuración de la escaleta de vídeos. Otro componente muy usado en esta aplicación son las listas, estas se utilizan para mostrar los vídeos y aplicaciones disponibles en cada momento, estas lista se encuentran en las ventanas de gestion de vídeos, configuración de la escaleta de vídeos y gestión de aplicaciones. Estos y otros componentes utilizados en esta aplicación se explicarán con más detalle en el apartado dedicado a los componentes.
- **Contenedores:** Son clases o interfaces que permiten crear objetos gráficos para contener a los componentes, tales como paneles, cuadros de dialogo, ventanas,… Para crear cada ventana de esta aplicación se ha utilizado un contenedor general de tipo ***JFrame***, este contenedor crea la ventana en sí misma. Después con ayuda de otro tipo de contenedores, como por ejemplo los ***JPanel*** se distribuyen los distintos componenetes en el espacio de la ventana. Una ventana puede contener varios paneles.

1.4.1.- EVENTOS

Además de tener una interfaz visible se debe posibilitar al usuario la interacción con la aplicación. Para ello se implementa la gestión de los eventos provocados por el usuario que tienen un efecto en las acciones de la aplicación.

El sistema de gestión de eventos de Java 1.2 es el mismo que el de Java 1.1 y por lo tanto el mismo que para las librerías AWT. Aunque los desarrolladores de Java considerasen que para mejorar el lenguaje se necesitaba dejar a un lado las librerías AWT e introducir las librerías Swing no pensaron lo mismo del sistema de gestión de eventos, consideraron que era lo suficientemente bueno.

Como ya se ha comentado, un evento es una acción sobre algún componente. Todos los sistemas operativos están constantemente atendiendo a los eventos generados por los usuarios. Estos eventos pueden ser pulsar una tecla, mover el ratón, hacer clic con el ratón, pulsar el ratón sobre un botón o menú (Java distingue entre simplemente pulsar el ratón en un sitio cualquiera o hacerlo, por ejemplo, en un botón). El sistema operativo notifica a las aplicaciones que están ocurriendo estos eventos, y ellas deciden si han de responder o no de algún modo a este evento.

El modelo de Java se basa en la delegación de eventos, es decir, el evento se produce en un determinado componente, por ejemplo, un botón. A continuación el evento se transmite al *event listener* (manejador de eventos) que esté asignado al componente en el que se produjo el evento. El objeto que escucha los eventos es el que se encarga de responder a ellos adecuadamente. Esta separación de código entre generación del evento y actuación respecto a él facilita la labor del programador y da una mayor claridad a los códigos.

A continuación se detalla lo que ocurre cuando un usuario presiona un botón de la aplicación. A la fuente del evento, en este caso un botón, se le indica quién será su manejador de eventos (el método que atenderá el evento), el manejador ha de extender la clase ***Adapter*** correspondiente o implementar la interfaz *listener*, en este caso se ha optado por implementar la interfaz ***ActionListener*** que es la que necesitan los objetos ***JButton***. Cuando el usuario pulse el botón, el objeto fuente empaqueta la información acerca del evento generando un objeto de tipo ***Event*** (***ActionEvent*** en este caso) e invoca al método correspondiente del manejador (en este caso ***actionPerformed(actionEvent)***) pasándole como información el objeto de tipo ***Event*** generado. Es responsabilidad del manejador, y

no de la fuente, responder al evento, por ello se dice que la fuente delega la gestión del evento en el manejador.

Lo que la fuente de eventos le pasa al objeto encargado de escuchar los eventos es un objeto de tipo ***Event***. En este objeto va toda la información necesaria para la correcta gestión del evento por parte del *listener*. El *listener* ha de implementar para ello una interfaz, el nombre de esta interfaz es siempre el nombre del evento más la palabra *listener*.

A continuación se añaden unas líneas de código a modo de ejemplo de cómo se han implementado los botones en esta aplicación.

```
/*
 * Panel con los botones
 * */

JPanel jpButAddTs = new JPanel();

jpButAddTs.add(jbAddTs);
jpButAddTs.add(jbDeleteTS);
jpButAddTs.add(jbExitAddTs);
jpButAddTs.add(jbAddScheItem);

jbAddTs.addActionListener(this);
jbDeleteTS.addActionListener(this);
jbExitAddTs.addActionListener(this);
jbAddScheItem.addActionListener(this);
```

En primer lugar se añaden los botones al panel, en este caso se ha creado un panel especifico que sólo va a contener los botones, de esta manera este panel se tratará como un único componente independientemente del número de botones que contenga. Esto facilita las cosas a la hora de ubicar todos los componentes en la ventana. A continuación se asocian los *listeners* a los componentes. Además de esto, se debe implementar la interfaz ***ActionListener*** en la declaración de la clase:

```
public class WindowTS extends JFrame implements ActionListener
```

Esta interfaz obliga a implementar el método ***actionPerformed()***, en este método se definen las acciones que la aplicación debe realizar cuando recibe los eventos que se han programado.

```
public void actionPerformed(ActionEvent arg0)
{
	//Se ha presionado el botón Añadir
	if (arg0.getSource().equals(jbAddTs))
	{
		WindowAddTS wats = new WindowAddTS();
		wats.createWindow();

	}
	//Se ha presionado el botón Borrar
	else if (arg0.getSource().equals(jbDeleteTS))
	{
		String txDelTS;
		txDelTS = jlTs.getSelectedValue().toString();
		boolean boDeleteTS;
		boDeleteTS = uts.deleteTS(txDelTS);
		if (boDeleteTS)
		{
			uploadListTs();
		}
	}
	//Se ha presionado el botón Salir
	else if (arg0.getSource().equals(jbExitAddTs))
	{
		jfTs.removeAll();
		jfTs.dispose();
	}
	//Se ha presionado el botón Crear TS Scheduler
	else if (arg0.getSource().equals(jbAddScheItem))
	{
		WindowCreateSchduledTS wcsts = new
		WindowCreateSchduledTS();
		wcsts.createWindow();
	}

}
```

En este caso, como se están implementando unos botones, en este método se especifica a la aplicación lo que debe hacer cuando el usuario pulse cada uno de los botones.

1.4.2.- LAYOUTS

La distribución de los componentes en un contenedor usualmente está controlada por un administrador de diseño (*layout manager*). Cada contenedor (como un panel o un *Frame*) tiene por omisión un administrador de diseño asociado con este, el cual se puede cambiar invocando al método ***setLayout()***.

Cada administrador de diseño es responsable de definir el tamaño y la situación de cada componente en sus contenedores.

El lenguaje de programación Java incluye los siguientes administradores de diseño:

- ***FlowLayout***: Es el administrador de diseño por defecto en el ***JPanel*** y en el ***JApplet***. Coloca los componentes de izquierda a derecha y de arriba hacia abajo. Les modifica la posición pero no les modifica el tamaño.
- ***BorderLayout***: Es el administrador de diseño por defecto de ***JWindow***, ***JDialog*** y ***JFrame***. Coloca los componentes en regiones, Norte, Sur, Este, Oeste y Centro. Modifica tanto la posición como el tamaño de los componentes.
- ***CardLayout***: Es raramente usado y coloca los componentes en capas, donde cada capa es como una carta de naipes, de ahí su nombre. Modifica tanto la posición como el tamaño de los componentes.
- ***GridLayout***: Es un administrador de diseño que provee flexibilidad para colocar componentes a manera de celdillas de una hoja de cálculo (Matriz). El tamaño de todas las celdas es igual.
- ***GridBagLayout***: Es parecido al GridLayout, pero no fuerza a que todos los componentes tengan el mismo tamaño.

En la descripción de los distintos componentes se muestra el uso de distintos *Layouts* dependiendo de cuál sea la disposición de los componentes que se desee.

1.4.3.- COMPONENTES

Como ya se ha comentado al principio de este apartado, los componentes son clases o interfaces que permiten crear objetos gráficos que componen la interfaz gráfica de usuario.

A continuación se van a comentar algunos de los componentes que se han utilizado en la implementación de la interfaz gráfica de esta aplicación:

JLabel

No es más que una etiqueta de texto que se puede poner al lado de cualquier componente para darle una indicación al usuario de cuál es la función de dicho componente.

En la aplicación se utilizan las etiquetas sobre todo asociadas a campos de texto para indicar al usuario la función o el contenido de estos.

```
/*
 * Declaracion de variables
* */
private String txVideo = "Video: ";
private String txInitDate = "Inicio(Fecha): ";
private String txInitTime = "Inicio(Hora): ";
private String txEndDate = "Final(Fecha): ";
private String txEndTime = "Final(Hora): ";

private JLabel jlVideo = new JLabel(txVideo);
private JLabel jlInitDate = new JLabel(txInitDate);
private JLabel jlInitTime = new JLabel(txInitTime);
private JLabel jlEndDate = new JLabel(txEndDate);
private JLabel jlEndTime = new JLabel(txEndTime);

private JTextField jtfVideo = new JTextField(10);
private JTextField jtfInitDate = new JTextField(10);
private JTextField jtfInitTime = new JTextField(10);
private JTextField jtfEndDate = new JTextField(10);
private JTextField jtfEndTime = new JTextField(10);
```

En primer lugar se declaran las variables, para crear una nueva variable de tipo ***JLabel*** se le pasa como parámetro un ***String*** con el texto que va a contener la etiqueta. En este caso se han definido variables ***String*** independientes pero también se podría haber incluido el texto directamente en la definición de las etiquetas.

```
jlVideo.setLabelFor(jtfVideo);
jlInitDate.setLabelFor(jtfInitDate);
jlInitTime.setLabelFor(jtfInitTime);
jlEndDate.setLabelFor(jtfEndDate);
jlEndTime.setLabelFor(jtfEndTime);
```

Se utiliza el método ***setLabelFor()*** para asociar cada etiqueta con el componente al que describe. En este caso, las etiquetas definen el contenido de los campos de texto, de modo que se asocian a los objetos ***JTextField*** que definen dichos campos de texto.

Por último se añaden las etiquetas y los campos de texto al panel

```
/*
 * Se colocan los campos de texto y las etiquetas.
 * */

JPanel jpData = new JPanel();
jpData.setPreferredSize(new Dimension(300, 130));
jpData.setBorder(new EmptyBorder(10, 10, 10, 10));

JPanel jpData11 = new JPanel();
jpData11.setLayout(new GridLayout(5, 1, 5, 5));

JPanel jpData12 = new JPanel();
jpData12.setLayout(new GridLayout(5, 1, 5, 5));

jpData11.add(jlVideo);
jpData12.add(jtfVideo);

jpData11.add(jlInitDate);
jpData12.add(jtfInitDate);

jpData11.add(jlInitTime);
jpData12.add(jtfInitTime);

jpData11.add(jlEndDate);
jpData12.add(jtfEndDate);

jpData11.add(jlEndTime);
jpData12.add(jtfEndTime);

jpData.setLayout(new BorderLayout());
jpData.add(jpData11, BorderLayout.WEST);
jpData.add(jpData12, BorderLayout.CENTER);
```

En la **Figura 1.14** se puede ver cómo quedan estos componentes en la ventana.

Figura 1.14.- *Ventana Editar Video*

JTextField y ***JPasswordField***

JTextField: Está pensado para obtener texto del usuario, pero únicamente se puede recoger una línea de texto por campo. Este componente tiene métodos para recoger el texto del usuario, poner un texto en él, recoger sólo el texto seleccionado, seleccionar una parte del texto, insertar texto, cortar texto, pegar texto, etc

JPasswordField: Es un ***JTextField*** en el cual al escribir no se ve lo que se está escribendo, sino un carácter (* en este caso). Se emplea para pedir *passwords* al usuario y evitar que estas puedan ser leídas por otra persona.

En la ventana de inicio de la aplicación (**Figura 1.15**) se utilizan ambos componentes, ya que en esta ventana se le pide al usuario el nombre de usuario y la contraseña que da acceso al resto de la aplicación.

Figura 1.15.- *Ventana Conexión con el laboratorio*

La implementación de la parte indicada como "Datos del laboratorio" es muy similar a la que se ha descrito en el apartado dedicado a las etiquetas. La única diferencia es que en este caso uno de los campos de texto es un objeto ***JPasswordField***, pero el tratamiento de este tipo de objetos es idéntico al que reciben los objetos ***JTextField***.

```
/*
 * Declaracion de variables
 * */

private String txIp = "Direcion IP: ";
private String txLab = "Tipo de laboratorio: ";
private String txUser = "Usuario: ";
private String txPassw = "Contraseña: ";

private JLabel jlIp = new JLabel (txIp);
private JLabel jlLab = new JLabel (txLab);
private JLabel jlUser = new JLabel (txUser);
private JLabel jlPassw = new JLabel (txPassw);

private JTextField jtfIp = new JTextField(20);
private JTextField jtfLab = new JTextField(20);
private JTextField jtfUser = new JTextField(20);
private JPasswordField jpfPassw = new JPasswordField();

[…]

jlIp.setLabelFor(jtfIp);
jlLab.setLabelFor(jtfLab);
jlUser.setLabelFor(jtfUser);
jlPassw.setLabelFor(jpfPassw);
```

```
[…]

JPanel jpDataLab11 = new JPanel();
jpDataLab11.setLayout(new GridLayout(4, 1, 5, 5));

JPanel jpDataLab12 = new JPanel();
jpDataLab12.setLayout(new GridLayout(4, 1, 5, 5));

jpDataLab11.add(jlIp);
jpDataLab12.add(jtfIp);

jpDataLab11.add(jlLab);
jpDataLab12.add(jtfLab);

jpDataLab11.add(jlUser);
jpDataLab12.add(jtfUser);

jpDataLab11.add(jlPassw);
jpDataLab12.add(jpfPassw);

jpDataLab.setLayout(new BorderLayout());
jpDataLab.add(jpDataLab11, BorderLayout.WEST);
jpDataLab.add(jpDataLab12, BorderLayout.CENTER);
jpPanelLab.add(jpDataLab);
```

JList

Se trata de un componente cuya función es mostrar al usuario una lista de opciones, en este componente las posibles selecciones se encuentran en una lista, que normalmente lleva un *scroll* incorporado, y se selecciona haciendo clic directamente sobre ellas. Se emplea cuando el número de opciones entre las que ha de escoger el usuario es demasiado grande para representarlo en forma de *radio buttons* o *checkboxes*. Este componente posee métodos para permitir simple o múltiple selección, seleccionar o deseleccionar un componente, averiguar qué componente está seleccionado, etc

En la aplicación se puede encontrar este componente, por ejemplo, en la ventana Lista de ficheros en el carrusel (**Figura 1.16**). En esta ventana el componente ***JList*** contiene la lista de ficheros que se encuentran cargados en el carrusel con el que se está trabajando.

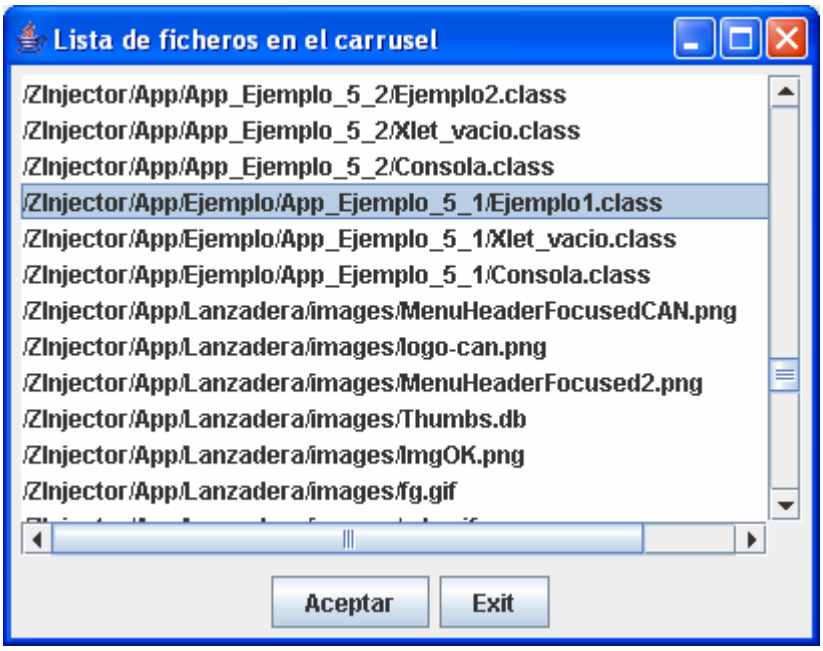


Figura 1.16.- *Ventana Lista de ficheros en el carrusel*

La implementación de este componente comienza como en otros casos por la declaración de la variable que va a representar la lista.

```
private static DefaultListModel dlmFile = new
     DefaultListModel();
private static JList jlistFile = new JList(dlmFile);
```

Como se puede ver, se crean dos variables una de tipo ***JList*** y la otra de tipo ***DefaultListModel***, esta última se pasa como parámetro para inicializar la lista. Una lista también se puede inicializar pasando como parámetro un vector con los valores iniciales que se quieran incluir a la lista, pero si no se conocen de antemano estos valores, los objetos ***ListModel*** facilitan las acciones de añadir valores a la lista.

```
/*
 * Se crea la lista
 * */

arrTxFile = WindowApp.cCarousel.getObjectList(true, false,
     false, false);

if(dlmFile.size() != 0)
{
     dlmFile.clear();
}
if(arrTxFile != null)
{
```

```
      for(int i=0; i<arrTxFile.length; i++)
      {
            dlmFile.addElement(arrTxFile[i]);
      }
}
```

Como se puede ver, se utiliza la variable ***DefaultListModel*** para rellenar la lista.

Por último se configuran algunas opciones de la lista.

```
jlistFile.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
jlistFile.setSelectedIndex(0);
jlistFile.setLayoutOrientation(jlistFile.VERTICAL);
jlistFile.setVisibleRowCount(-1);
```

El método ***setSelectionMode()*** se utiliza para indicar que la lista es de selección única, es decir, que sólo puede haber una opción seleccionada al mismo tiempo. El método ***setSelectedIndex()*** indica a la aplicación que cuando cree la lista muestre el primer elemento de la lista seleccionada por defecto. El siguiente método (***setLayoutOrientation()***) especifica la orientación que tienen los valores dentro de la lista. Por último se puede seleccionar el número de elementos que muestra la lista por defecto con el método ***setVisibleRowCount()***, en esta caso el parámetro (-1) indica que se mostrarán todos los elementos que sean visibles sin tener que cambiar el tamaño original de la lista.

JComboBox

Su filosofía es idéntica a la de ***JList***, pero en esta ocasión las opciones no se ven en un principio. El usuario ha de hacer un clic sobre una pestaña que desplegará una lista de opciones sobre las cuales el usuario escoge una mediante un clic sobre la que le interesa.

En esta ocasión se puede encontrar un ejemplo del uso de este componente en la ventana Nueva entrada en la escaleta de aplicaciones (**Figura 1.17**).

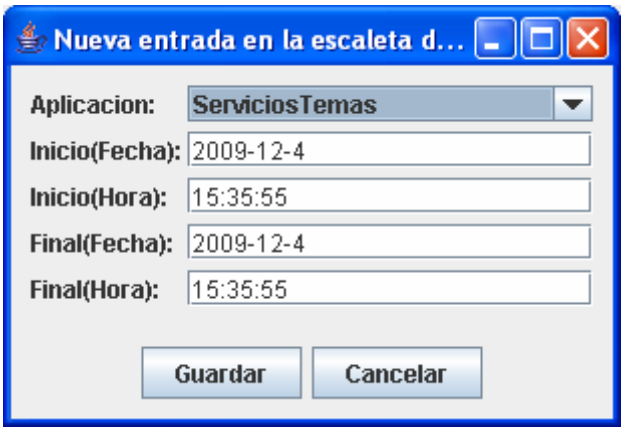


Figura 1.17.- *Ventana Nueva entrada en la escaleta de aplicaciones*

En esta ventana, el *combo box* se utiliza para que el usuario elija la aplicación que quiere programar. La implementación de este componente es idéntica a la vista para los campos de texto.

```
/*
 * Declaracion de variables
 * */
private String txApp = "Aplicacion: ";
private JLabel jlApp = new JLabel(txApp);
private JComboBox jcbApp = new JComboBox();

[…]

jlApp.setLabelFor(jcbApp);

[…]

/*
 * Se colocan los campos de texto y las etiquetas.
 * */
JPanel jpSchApp = new JPanel();
jpSchApp.setLayout(new BoxLayout(jpSchApp, BoxLayout.Y_AXIS));

JPanel jpDataApp = new JPanel();
jpDataApp.setPreferredSize(new Dimension(300, 130));
jpDataApp.setBorder(new EmptyBorder(10, 10, 10, 10));

JPanel jpDataApp11 = new JPanel();
jpDataApp11.setLayout(new GridLayout(5, 1, 5, 5));

JPanel jpDataApp12 = new JPanel();
jpDataApp12.setLayout(new GridLayout(5, 1, 5, 5));

jpDataApp11.add(jlApp);
jpDataApp12.add(jcbApp);
```

```
[…]

/*
 * Se rellena el combobox
 * */
jcbApp.removeAllItems();

UploadApp ulApp = new UploadApp();
String[] arrTxApp =
ulApp.appObtention(WindowApp.imcConnListApp);

for (int i=0; i<arrTxApp.length; i++)
{
     jcbApp.addItem(arrTxApp[i]);
}
```

La principal diferencia con la implementación de un campo de texto es que, como ocurría con las listas, hay que rellenar el *combo box* con las opciones que se le van a ofrecer al usuario.

JFileChooser

La ventana que permite la selección de archivos, suele ser una ventana modal, ya que los cambios que se produzcan en ella, o la selección que se haga, repercutirán en el funcionamiento de la aplicación general. Normalmente, la ventana de selección de archivos se utiliza para presentar una lista de archivos y permitir al usuario seleccionar cuál de ellos debe abrir la aplicación; o, por el contrario, permitir al usuario introducir un nombre o seleccionar un archivo con el que se quieren guardar datos. El objeto ***JFileChooser*** no realiza ninguna de estas acciones, es decir, no abre ni salva nada, sino que se limita a seleccionar el nombre del archivo con el que se desea realizar la acción y es responsabilidad del programa el llevar a cabo la apertura de archivo o la grabación de datos.

En la aplicación se hace uso de este componente, por ejemplo, en la ventana Nuevo Video (**Figura 1.18**).

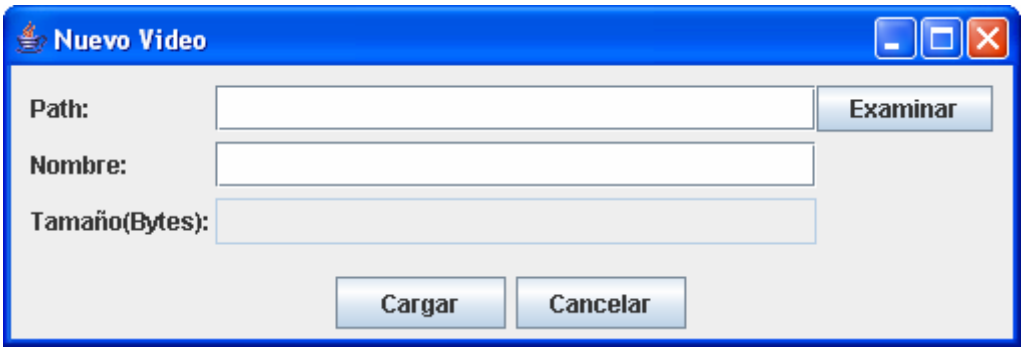


Figura 1.18.- *Ventana Nuevo Video*

A través del botón Examinar se accede a la ventana que permite la selección de archivos. Esta ventana se abre al presionar el botón, por lo tanto como se ha comentado antes, las acciones que son consecuencia de un evento generado por el usuario al presionar un botón se implementa en el método ***actionPerformed()***.

```
public void actionPerformed(ActionEvent arg0)
{
      if (arg0.getSource().equals(jbSearch))
      {
            try
            {
                  TsChooser();
            }
            catch (IOException ioe)
            {
                  System.out.println("Excepcion de entrada-salida
                        al buscar un fichero: " + ioe.getMessage());
                  System.out.println("Error al buscar el fichero.");
            }
      }
[…]
}
```

```
public void TsChooser () throws IOException
{

      JFileChooser jfchLogo = new JFileChooser();
      jfchLogo.setCurrentDirectory(new File("C://Documents and
            Settings//elegadara//Mis documentos//
            ArangurenProyect//Videos//" + "pruebas articulo"));
      if (jfchLogo.showOpenDialog(null) !=
            JFileChooser.APPROVE_OPTION)
      {
            return;
      }
      fileTs = jfchLogo.getSelectedFile();
```

```
      jtfPath.setText(fileTs.getCanonicalPath());
      jtfSize.setText(String.valueOf(fileTs.length()));
}
```

En el método ***TSChooser*** se especifica que debe hacer la aplicación desde la ventana de selección de ficheros.

1.4.4.- CONTENEDORES

Los componentes no se encuentran aislados, sino agrupados dentro de Contenedores. Los contenedores contienen y organizan la situación de los componentes, además, los contenedores son también componentes, por lo que pueden ser situados dentro de otros contenedores. También contienen el código necesario para el control de eventos, cambiar la forma del cursor o modificar el icono de la aplicación. Hay dos tipos principales de contenedores, la clase ***JWindow*** (contenedores de alto nivel) y los contenedores de nivel intermedio.

Una ventana (***JWindow***) es un rectángulo que flota libremente en la pantalla. Hay dos tipos de ventanas, el ***JFrame*** y el ***JDialog***. En esta aplicación, todas las ventanas que la componen son de tipo ***JFrame***. Un ***JFrame*** es una ventana con una barra de titulo y esquinas que permiten que se cambie el tamaño original de la ventana.

```
JFrame jfInicio = new JFrame ("Conexion con el laboratorio");

jfInicio.getContentPane().add(jpContenedor,BorderLayout.CENTER);
jfInicio.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jfInicio.pack();
jfInicio.setVisible(true);
```

Estas líneas de código pertenecen a la ventana de inicio. La ventana de inicio se diferencia de las demás en que se le ha definido una operación de salida, es decir, que si se cierra la ventana de inicio finaliza la aplicación, con el resto de ventanas no ocurre esto. La diferencia se encuentra en que en este caso se añade una línea más donde se llama al método ***setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)***. El resto de ventanas no implementan esta línea de código.

En cuanto a los contenedores de nivel intermedio, en esta aplicación se han utilizado sobre todo dos de ellos, que se describen a continuación. Este tipo de contenedores no pueden ser usados como un componente libre, siempre deben estar asociados al menos a un contenedor de alto nivel (por ejemplo, un ***JFrame***).

JScrollPane

Un ***JScrollPane*** provee un área de visualización dentro de una región grande y con barras de deslizamiento para manipularla. El ***JScrollPane*** crea y maneja las barras de desplazamiento y las contiene en un solo componente. No es posible controlar el administrador de diseño que usa, en lugar de esto se puede agregar a un ***JPanel*** para que el ***JScrollPane*** sea un componente más.

Generalmente no es posible manejar eventos en un ***JScrollPane***, los eventos son manejados a través de los componentes que estén contenidos en el ***JScrollPane***.

En la aplicación se ha utilizado este tipo de contenedor para colocar en él los elementos de tipo JList.

```
/*
 * Se coloca la lista en un panel
 * */

JScrollPane jspTs = new JScrollPane(jlTs);
jspTs.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_
    ALWAYS);
jspTs.setPreferredSize(new Dimension(250, 250));
jspTs.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
```

JPanel

Los paneles son los contenedores de propósito general más frecuentemente utilizados. Están implementados con la clase ***JPanel***, no añaden casi ninguna funcionalidad más allá de las que tienen los objetos ***JComponent***. Normalmente se usan para agrupar componentes, ya sea porque los componentes están relacionados o sólo porque agruparlos hace que la distribución sea más sencilla. Un panel puede usar cualquier controlador de distribución, y se les puede dotar de bordes fácilmente.

Un panel es como un *frame* y provee del espacio para que el programador coloque cualquier componente GUI, incluyendo también otros paneles. Cada panel hereda de la clase ***Container*** todos sus métodos y atributos, y puede tener su propio administrador de diseño.

En la aplicación se ha utilizado bastante este tipo de contenedores para agrupar los componentes y así facilitar la implementación de la distribución deseada para la ventana.

Por ejemplo, se pueden agrupar los botones de una ventana en un panel y así sólo es necesario colocar el panel en la ventana y no cada uno de los botones.

```
/*
 * Panel con los botones
 * */

JPanel jpButAddTs = new JPanel();

jpButAddTs.add(jbAddTs);
jpButAddTs.add(jbDeleteTS);
jpButAddTs.add(jbExitAddTs);
jpButAddTs.add(jbAddScheItem);

jbAddTs.addActionListener(this);
jbDeleteTS.addActionListener(this);
jbExitAddTs.addActionListener(this);
jbAddScheItem.addActionListener(this);

/*
 * Se agrupan los dos paneles en un panel contenedor
 */

JPanel jpContenedor = new JPanel();
BoxLayout blAddTs = new BoxLayout (jpContenedor,
    BoxLayout.Y_AXIS);
jpContenedor.setLayout(blAddTs);
jpContenedor.add(jspTs);
jpContenedor.add(jpButAddTs);
```

2.- MANUAL DE USUARIO

En este capítulo se va a detallar el funcionamiento de la aplicación desde el punto de vista del usuario. Para ello, se describirá el funcionamiento y significado de los campos y botones de las ventanas de la aplicación.

2.1.- Instalación de la aplicación

Como se trata de una aplicación implementada en lenguaje Java, hay un requisito mínimo que debe cumplir la máquina en la que se quiera ejecutar esta aplicación, ésta debe tener instalada la Máquina Virtual Java (JVM).

Para saber si el equipo tiene instalada alguna versión de la JVM se puede hacer lo siguiente:

- Se abre una ventana MS-DOS (en Windows INICIO→EJECUTAR y ahí se escribe "cmd" sin las comillas)
- Una vez que se ha abierto la ventana se escribe la siguiente línea de comando java –version (dejando un espacio entre la palabra java y el guión (-)).
- A continuación se presiona ENTER. Si todo es correcto y la maquina si tiene instalada una JVM debería salir algo así:

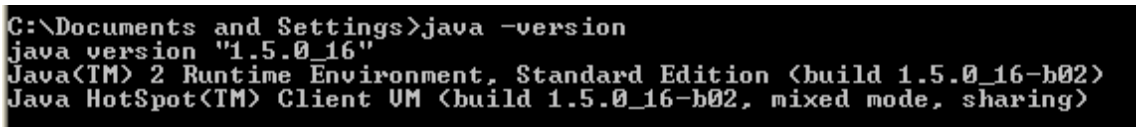


Figura2.1.- *Comprobación de la versión instalada de la JVM*

Si no se obtiene esta información significa que no hay ninguna versión de la JVM instalada y por tanto se deberá instalar.

Lo primero que se debe hacer es descargarse un paquete JRE (Java Runtime Environment), que es la máquina virtual Java más básica. Este paquete se puede descargar directamente de la página oficial de Java (www.java.com) o de numerosas páginas de distribución de Software gratuito (por ejemplo, Softonic, http://java.softonic.com/). Este paquete se distribuye como un fichero ejecutable (con extensión .exe si se está trabajando en Windows), por lo tanto para su instalación basta con ejecutar el fichero .exe que se acaba de descargar, y seguir las indicaciones del asistente de instalación hasta que finalice su instalación.

Una vez instalado el paquete JRE, se puedo comprobar si la instalación se ha completado correctamente ejecutando el comando **java –version**, comentado al inicio de esta sección.

Una vez que se tiene instalada la Máquina Virtual Java, ya se puede instalar la aplicación. Esta instalación consiste en copiar la carpeta en la que se distribuye la aplicación en un directorio del sistema. Esta carpeta contiene los siguientes ficheros y directorios:

- Scheduler.jar
- run.bat
- directorio → lib
    - imux-api.jar
    - imux-manager.jar
    - jsch-0.1.42.jar
    - pgsql-jdbc.jar

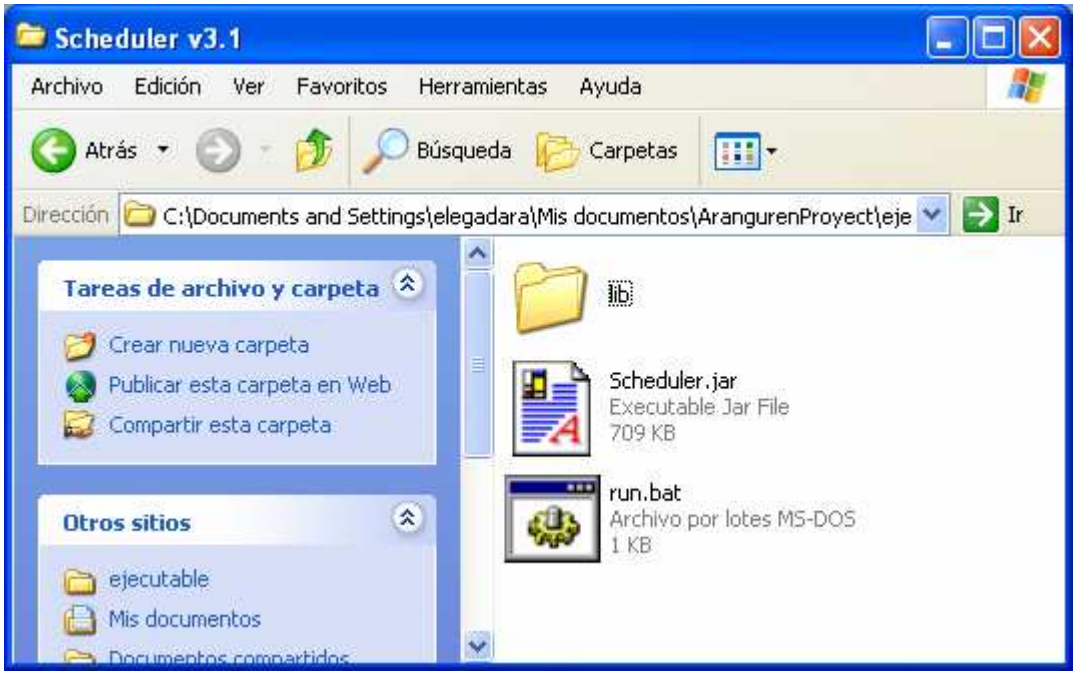


Figura2.2.- *Contenido del paquete en el que se distribuye la aplicación*

⚠ Es muy importante que estos 3 elementos (el archivo Scheduler.jar, el archivo run.bat y la carpeta lib) estén en el mismo directorio. Además la carpeta lib debe contener los 4 ficheros .jar que se han mencionado. Estos archivos son librerías necesarias para la

ejecución de la aplicación, si falta alguna de ellas o la carpeta lib no está en la ruta adecuada no será posible la ejecución de la aplicación.

Por último sólo queda lanzar la aplicación, para ello, si se ha instalado correctamente el JRE, bastaría con hacer doble clic sobre el archivo Scheduler.jar. Si esto no funciona puede ser debido a que, a veces, Windows no está configurado para asociar los .jar con la plataforma Java. Si este es el caso, se tienen dos opciones:

1.- Se clica con el botón derecho del ratón sobre el archivo Scheduler.jar y se selecciona la opción "Abrir con" y desde ahí se selecciona "Java Platform Standard". Como se muestra en la **Figura 2.3**.

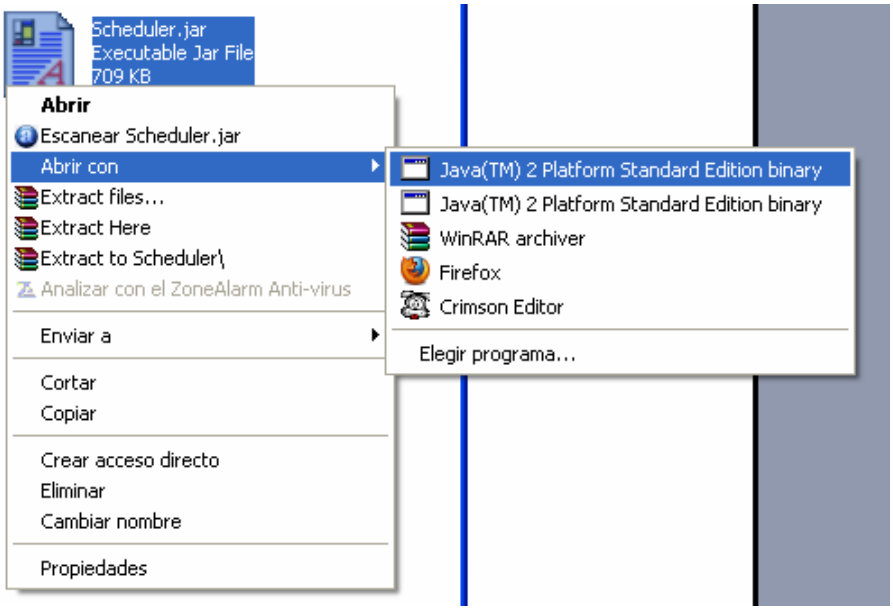


Figura 2.3.- *Cómo lanzar la aplicación desde "Abrir con"*

2.- La otra opción es lanzar la aplicación desde el archivo run.bat que se encuentra en la misma carpeta. En este caso, además de la aplicación, se abrirá también una ventana MS-DOS. En esta ventana se irán viendo algunos detalles de la ejecución del sistema. No se le hará mucho caso a esta ventana durante el uso de la aplicación, pero no se puede cerrar ya que esto provocaría el cierre de la aplicación también.

2.2.- Conexión con el laboratorio

Una vez que se ha completado con éxito la instalación de la aplicación, se puede comenzar a utilizar la aplicación. Cuando se lanza la aplicación, lo primero que aparece es la Ventana de conexión.

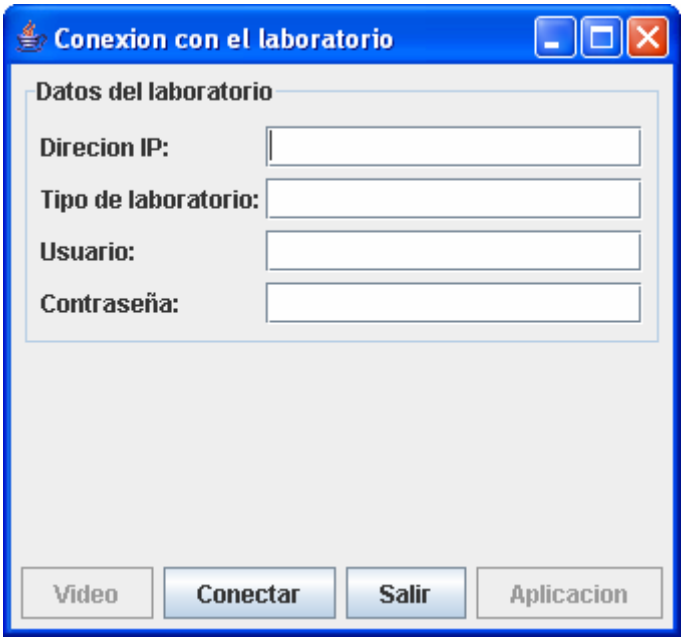


Figura 2.4.- *Ventana de inicio antes de la conexión*

Como se puede ver en la **Figura 2.4** se trata de una ventana de acceso a la aplicación. Esta aplicación utiliza un laboratorio multiplexor para completar la transmisión de los contenidos hacia los receptores. Por ello es necesario realizar la conexión con el laboratorio para posibilitar la comunicación entre éste y la aplicación.

La ventana consta de 4 campos de texto que se deben completar para realizar la conexión. Estos campos son:

- Dirección IP: Este campo contiene la dirección IP del servidor donde se encuentra alojado el laboratorio con el que se quiere establecer la conexión.
- Tipo de laboratorio: En este caso se trata del laboratorio iMux de la empresa alemana Mit-Xpert.
- Usuario: Nombre del usuario que quiere acceder a la aplicación
- Contraseña: Contraseña asignada a este usuario

Tras introducir los datos correspondientes a cada campo se presiona el botón 

de la ventana. Si la conexión se ha completado correctamente aparecerá en la ventana el texto informativo: “La conexión se ha realizado con éxito”.

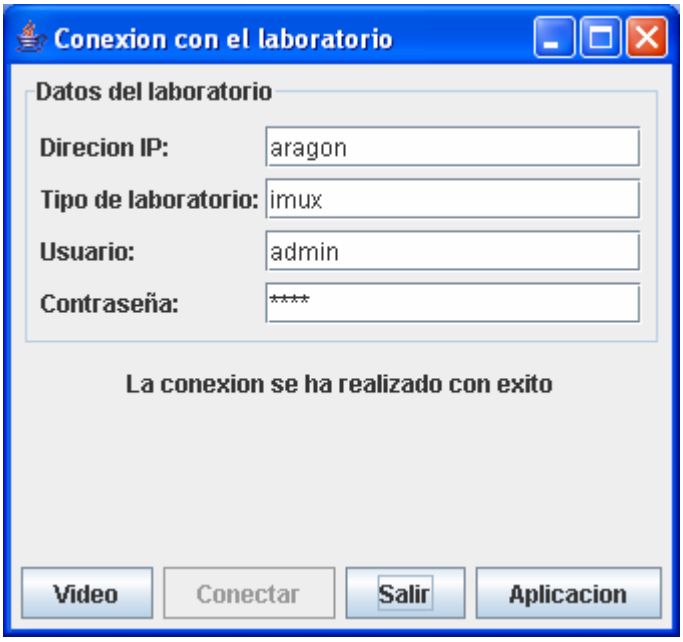


Figura 2.5- *Ventana de inicio tras la conexión*

2.3.- Vídeos

Como se puede ver en la **Figura 2.6** tras la conexión se deshabilita el botón Conectar que ya no es necesario, por el contrario los otros dos botones situados en los extremos se habilitan tras la conexión. El botón situado más a la izquierda 

es el que da acceso a este bloque de la aplicación.

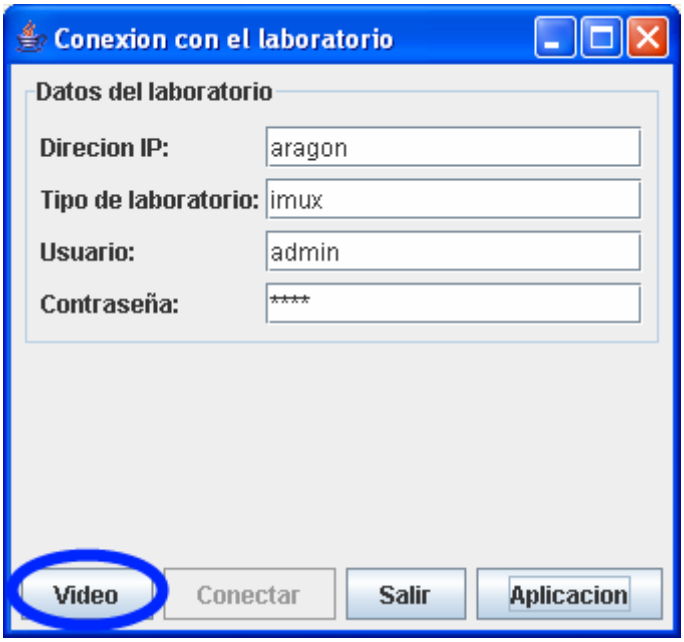


Figura 2.6.- *Acceso a la gestión de vídeos*

Esta parte dedicada al manejo de los vídeos se puede dividir en dos bloques claramente diferenciados por las características de sus funciones.

- Por un lado se tiene la **Gestión de la lista de vídeos disponibles**: En este bloque se encuentran las funciones destinadas a mantener actualizada la lista de vídeos, como son añadir un nuevo vídeo o borrar los vídeos que ya no son necesarios.
- Por otro lado está la **Configuración de la escaleta de reproducción**: En este bloque se gestiona la lista de videos que se quiere reproducir y el orden y los momentos en que se va a realizar su reproducción.

El acceso a esta parte requiere de una operación previa de carga de datos, de modo que tras presionar el botón Vídeo la aplicación carga todos los vídeos contenidos en el

laboratorio en ese momento. Esta operación puede durar varios segundos, durante este tiempo se muestra en la pantalla de inicio un mensaje que informa de que la aplicación se encuentra cargando datos.

Figura 2.7.- *Ventana de inicio durante el proceso de carga*

Una vez finalizado el proceso de carga, se despliega la ventana que da acceso a la gestión de los vídeos (ver **Figura 2.7**). En esta ventana se encuentran las funciones propias del bloque Gestión de la lista de vídeos disponibles. Por otra parte, a través de esta ventana también se tiene acceso al bloque Configuración de la escaleta de reproducción.

2.3.1.- Gestión de la lista de videos disponibles

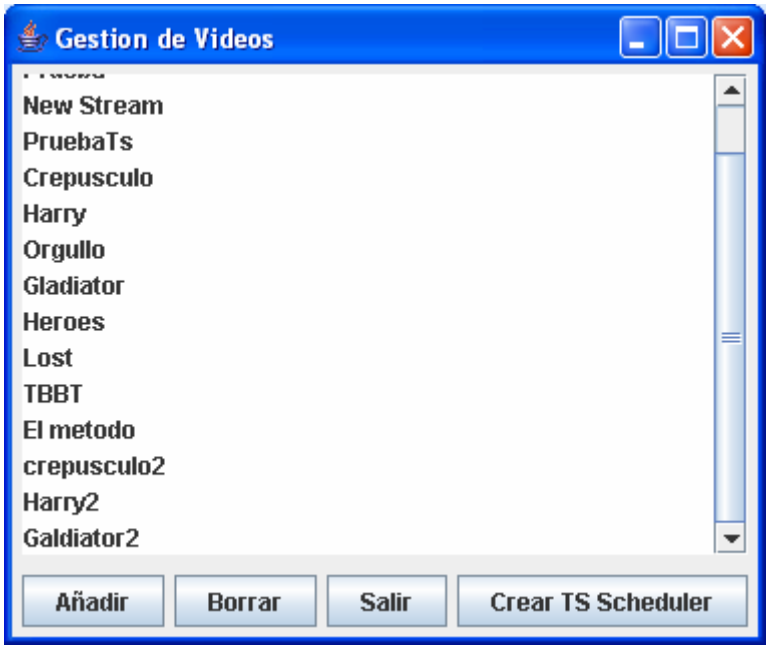


Imagen 2.8.- *Ventana Gestión de Vídeos*

En esta ventana se tiene una lista de todos los vídeos que hay disponibles en la aplicación en un determinado momento.

 Esta lista se actualiza de manera automática cada vez que se añade o borra un vídeo de la aplicación.

Desde esta ventana se pueden realizar 3 operaciones básicas:

- Añadir un nuevo vídeo a la aplicación
- Borrar un vídeo de la aplicación
- Acceder a la configuración de la escaleta de reproducción

2.3.1.1.- Añadir un nuevo vídeo

Si se quiere añadir un nuevo video a la aplicación se debe presionar el botón 

que se encuentra en la parte inferior izquierda de la ventana. Al presionar este botón se despliega la ventana Nuevo vídeo:

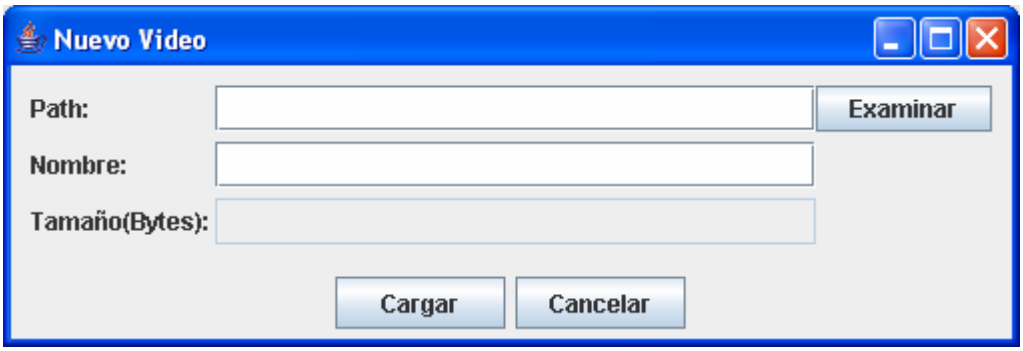


Figura 2.9.- *Ventana Nuevo vídeo*

Esta ventana contiene dos campos de texto editables donde se introducirán los datos del nuevo vídeo y uno más que no puede ser editado por el usuario.

- **Path:** En este campo se introduce la ruta donde se encuentra el fichero que se va a cargar. Con el botón Examinar, situado a la derecha del campo, se abre un explorador que facilita la tarea de introducir la ruta del fichero que se va a cargar. Una vez en la ventana del explorador se selecciona el archivo correspondiente y se presiona el botón Abrir de la ventana del explorador. De esta manera, el campo Path se rellena automáticamente con la ruta del fichero seleccionado.

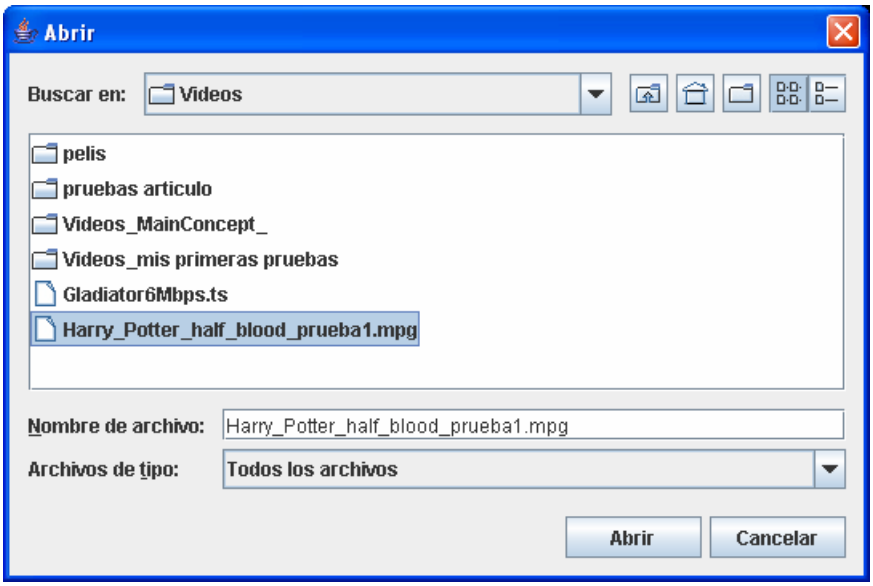


Figura 2.10.- *Explorador para seleccionar el video que se quiere añadir*

- **Nombre:** En este campo se introduce el nombre del video que se está cargando. Este nombre se utilizará para identificar al video en la aplicación, es decir, en la lista de videos disponibles aparecerá este nombre.

- **Tamaño:** Este campo es completado por la aplicación. Tras seleccionar el video que se quiere cargar, la aplicación automáticamente calcula el tamaño del fichero y completa el tercer campo de la ventana. Es, por tanto, un campo meramente informativo.

 Es importante tener en cuenta que los videos tienen que ser de tipo MPEG Transport Stream y con Bitrate constante para que funcionen correctamente. Si se quiere saber más sobre las características que deben cumplir los videos y de cómo convertir distintos formatos de video a un empaquetamiento Transport Stream se puede consultar el **Anexo** de esta memoria.

Una vez completados los campos correctamente se presiona el botón 

para finalizar la operación. Cuando termina el proceso de carga del video se cierra la ventana Nuevo vídeo automáticamente.

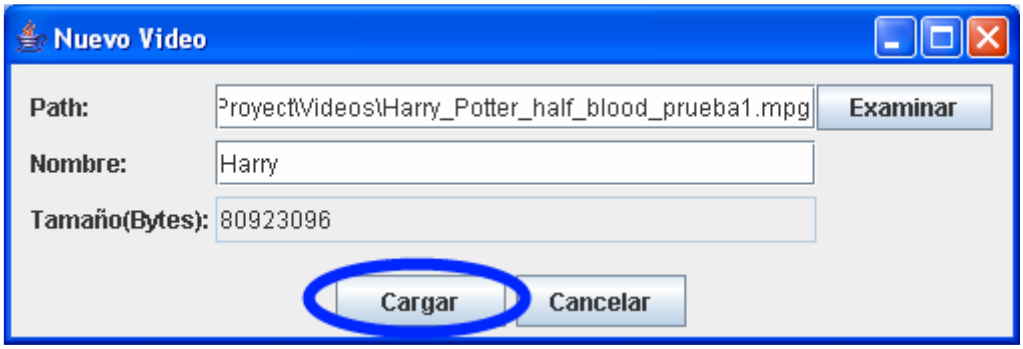


Figura 2.11.- *Ventana Nuevo Vídeo tras introducir los datos*

 Se debe tener en cuenta que no puede haber dos videos con el mismo nombre, de modo que al introducir el nombre del video en su campo correspondiente se debe comprobar que el nombre no esté ya en la lista. Si se le da un nombre que ya existe la aplicación cancela la operación de carga.

2.3.1.2.- Borrar un vídeo de la aplicación

Para eliminar un vídeo de la aplicación, en primer lugar se selecciona el vídeo que se quiere borrar, para ello basta con hacer clic con el ratón directamente en la lista sobre el nombre del vídeo.

Con el vídeo ya seleccionado se presiona el botón 

para confirmar la operación. Finalizada la operación se actualiza la lista, para que no sigua mostrando el vídeo borrado.

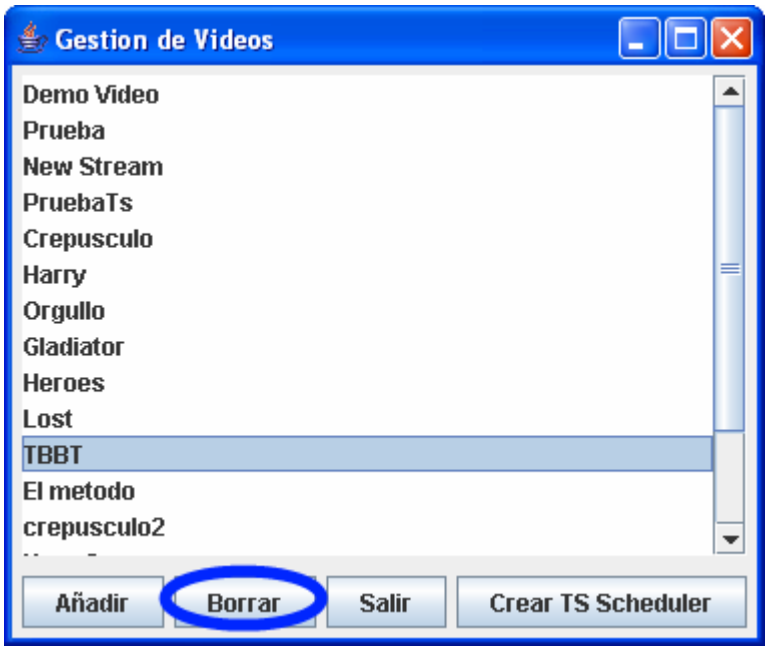


Figura 2.12.- *Borrado de un video de la aplicación*

2.3.1.3.- Acceso a la ventana Configuración de la escaleta de vídeos

Como ya se ha comentado antes, esta parte de la aplicación dedicada a la gestión de vídeos se divide en dos bloques. Hasta ahora se ha visto el bloque dedicado a la gestión de la lista de vídeos disponibles, que se maneja desde la ventana Gestión de vídeos. Para el otro bloque (Configuración de la escaleta de reproducción) se tiene una nueva ventana a la que se accede desde esta primera (Gestión de Vídeos) a través del botón Crear TS Scheduler.

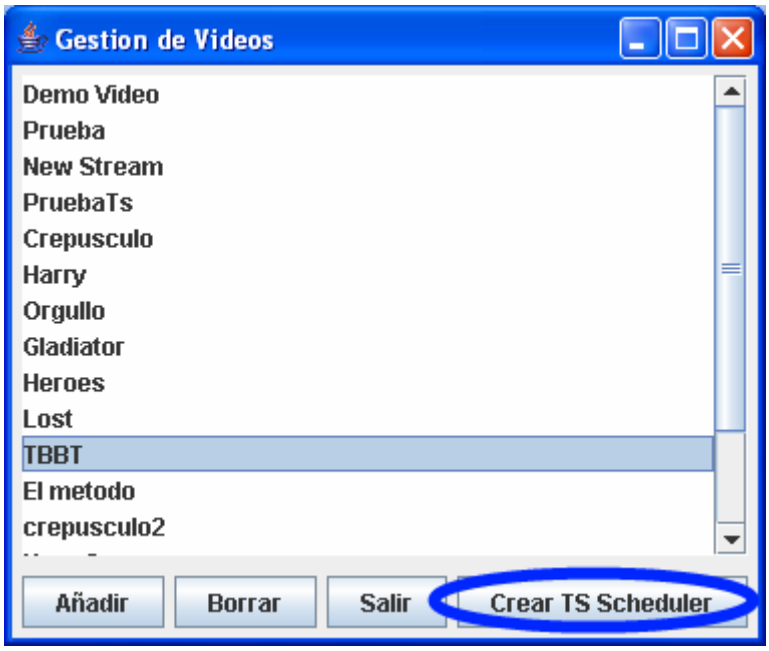


Figura 2.13.- *Acceso a la ventana Configuración de la escaleta de vídeo*

2.3.2.- Escaleta de reproducción

La gestión de la escaleta se realiza desde la ventana Configuración de la escaleta de vídeo, a la que, como se ha comentado ya, se accede desde la ventana Gestión de vídeo.

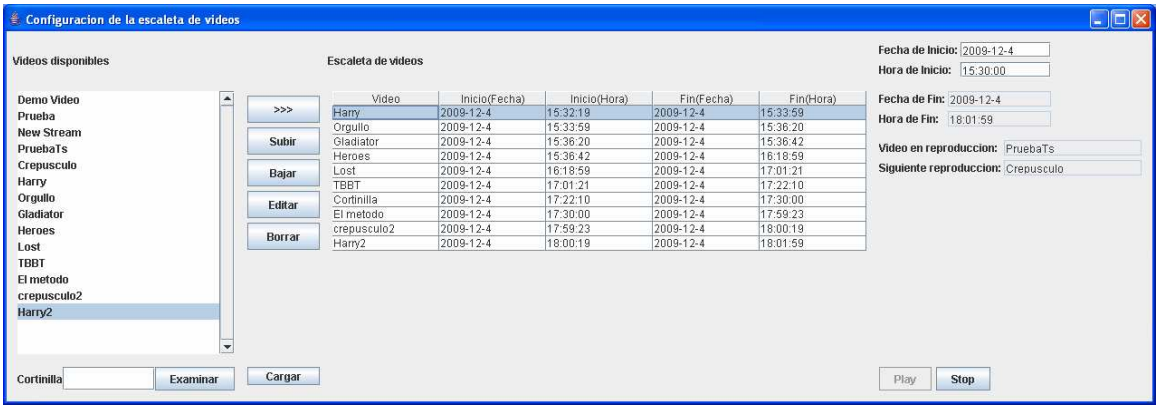


Figura 2.14.- *Ventana Configuración de la escaleta de vídeo*

En primer lugar se va a hacer una breve descripción de las partes que componen la ventana.

En la parte izquierda de la ventana se tiene la lista de videos disponibles igual a la que se tenía en la ventana anterior.

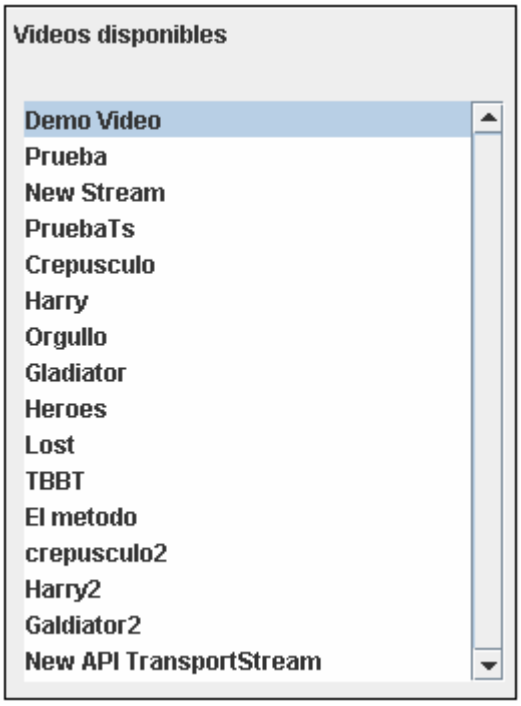


Figura 2.15.- *Lista de vídeos disponibles*

En la parte central de la ventana se sitúa la tabla que representa la escaleta de videos que se está gestionando en ese momento.

Escaleta de videos

Video	Inicio(Fecha)	Inicio(Hora)	Fin(Fecha)	Fin(Hora)
TBBT	2010-2-15	15:27:22	2010-2-15	15:48:11
El metodo	2010-2-15	15:48:11	2010-2-15	16:17:34
Heroes	2010-2-15	16:17:34	2010-2-15	16:59:51
Harry	2010-2-15	16:59:51	2010-2-15	17:01:31
Crepusculo	2010-2-15	17:01:31	2010-2-15	17:02:27
Gladiator	2010-2-15	17:02:27	2010-2-15	17:02:48

Figura 2.16.- *Tabla representativa de la escaleta de vídeos*

Esta tabla tiene las siguientes columnas:

- Nombre del vídeo
- Fecha de inicio de la reproducción
- Hora del inicio de la reproducción
- Fecha de fin de la reproducción
- Hora del fin de la reproducción

Entre estos dos elementos (la lista y la tabla) se encuentran los botones que dan acceso a las funciones que permiten configurar la escaleta de vídeos.

Figura 2.17.- *Botones para la configuración de la escaleta*

En la parte superior derecha de la ventana se encuentran varios campos con información sobre la reproducción de los vídeos.

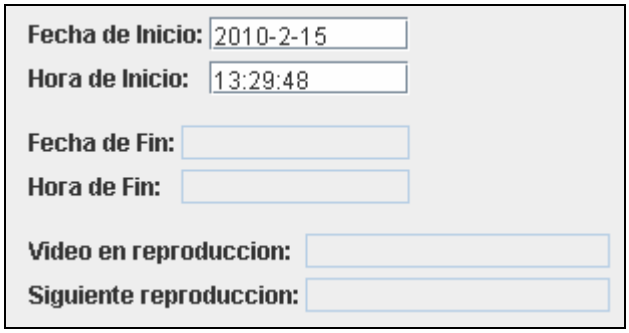


Figura 2.18.- *Campos con información de la escaleta en reproducción*

Los dos primeros campos indican el momento (Fecha y Hora) en que comienza la reproducción de la escaleta. Los dos campos centrales indican el momento (Fecha y Hora) en que finaliza la escaleta que se encuentra en reproducción.

Los dos últimos campos informan sobre el video que está en reproducción en ese momento y el siguiente vídeo que será reproducido.

En la parte inferior izquierda, se encuentra la parte dedicada a la modificación del vídeo por defecto que se reproducirá a modo de cortinilla cuando no haya ningún vídeo programado.

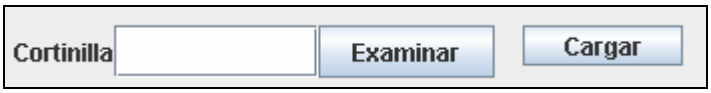


Figura 2.19.- *Campo para modificar el vídeo de la cortinilla*

Por último, en la parte inferior derecha de la ventana se encuentran los botones Play y Stop para iniciar y parar la reproducción respectivamente.

2.3.2.1.- Añadir un vídeo a la escaleta de reproducción

Para añadir un vídeo a la escaleta se selecciona primero el vídeo de la lista. Para ello se hace clic con el ratón directamente sobre la lista de vídeos disponibles.

No se puede añadir un vídeo que ya está en la escaleta.

Una vez que se ha seleccionado el vídeo que se quiere añadir se presiona el botón >>> El vídeo se añade al final de la tabla.

Figura 2.20.- *Añadir un video a la escaleta*

Si la escaleta está en reproducción, se completan todas las columnas de la tabla. Como el vídeo se añade al final de la tabla, la aplicación toma como hora de inicio del nuevo vídeo la hora de fin de la reproducción del vídeo anterior. Si no está en reproducción sólo se rellena la columna correspondiente al nombre, el resto se dejan en blanco hasta que comience la reproducción.

2.3.2.2.- Modificar la posición de un vídeo de la escaleta

Aunque los videos se añaden al final de la tabla, es posible que se desee modificar esta posición. Para ello se dispone de dos botones que permiten subir o bajar una posición de un vídeo de la tabla.

Cada vez que se presiona uno de estos botones el vídeo seleccionado se desplaza una posición hacia arriba o hacia abajo intercambiando su posición con el vídeo contiguo.

Para seleccionar un vídeo de la escaleta se hace clic directamente sobre la fila de la tabla que se quiere seleccionar.

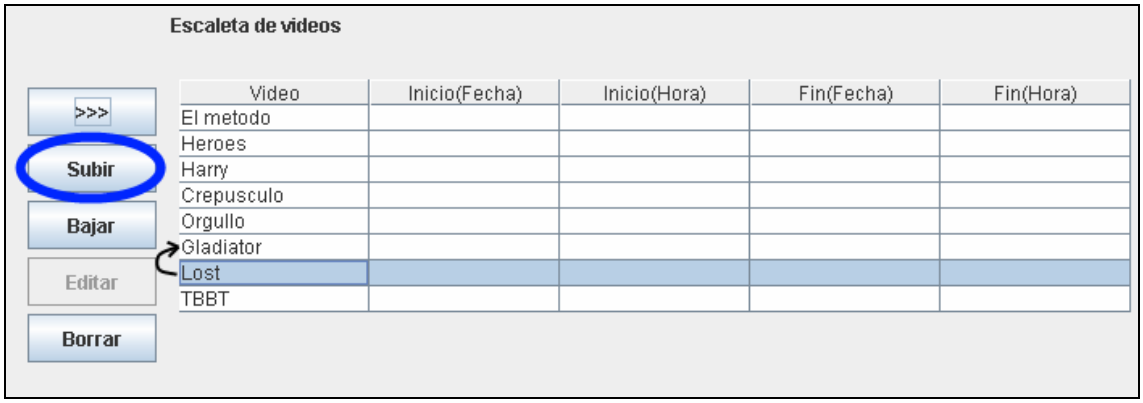


Figura 2.21- *Desplazar un vídeo de la escaleta una posición hacia arriba*

La posición de un vídeo en la tabla se puede modificar tanto si la escaleta está en reproducción como si está parada.

2.3.2.3.- Borrar un vídeo de la escaleta

Para borrar un vídeo de la escaleta, primero se selecciona la fila de la tabla correspondiente correspondiente al vídeo que se quiere borrar. Para seleccionar una fila de la tabla se hace clic con el ratón sobre la tabla.

A continuación se presiona el botón 

para confirmar la operación.

2.3.2.4.- Editar la hora de inicio de un vídeo de la escaleta

Esta opción sólo está disponible cuando la escaleta está en reproducción.

Para modificar la hora de inicio de un vídeo, en primer lugar se selecciona la fila de la tabla correspondiente al vídeo que se quiere editar. Para seleccionar una fila de la tabla se hace clic con el ratón sobre la tabla.

Una vez seleccionado el vídeo, se presiona el botón 

Esto hace que se abra una nueva ventana, Editar Vídeo.

Figura 2.22.- *Ventana Editar Vídeo*

En esta ventana sólo hay 2 campos editables, en ellos se introduce la nueva hora de inicio que se le quiere dar al vídeo.

La hora de inicio sólo se puede retrasar. No es posible adelantar la reproducción de un vídeo porque hay otros vídeos programados antes que él y si se adelanta la hora de inicio se solaparían.

Es importante añadir la nueva hora de inicio con el formato adecuado.

- Fecha → Año-Mes-Día (separados por un guión (-))
- Hora → Hora:Mes:Seg (separados por dos puntos (:))

Como referencia del formato se pueden tomar los valores de los campos Final.

Tras introducir la nueva hora de inicio en los campos correspondientes se presiona el botón 

para finalizar la operación.

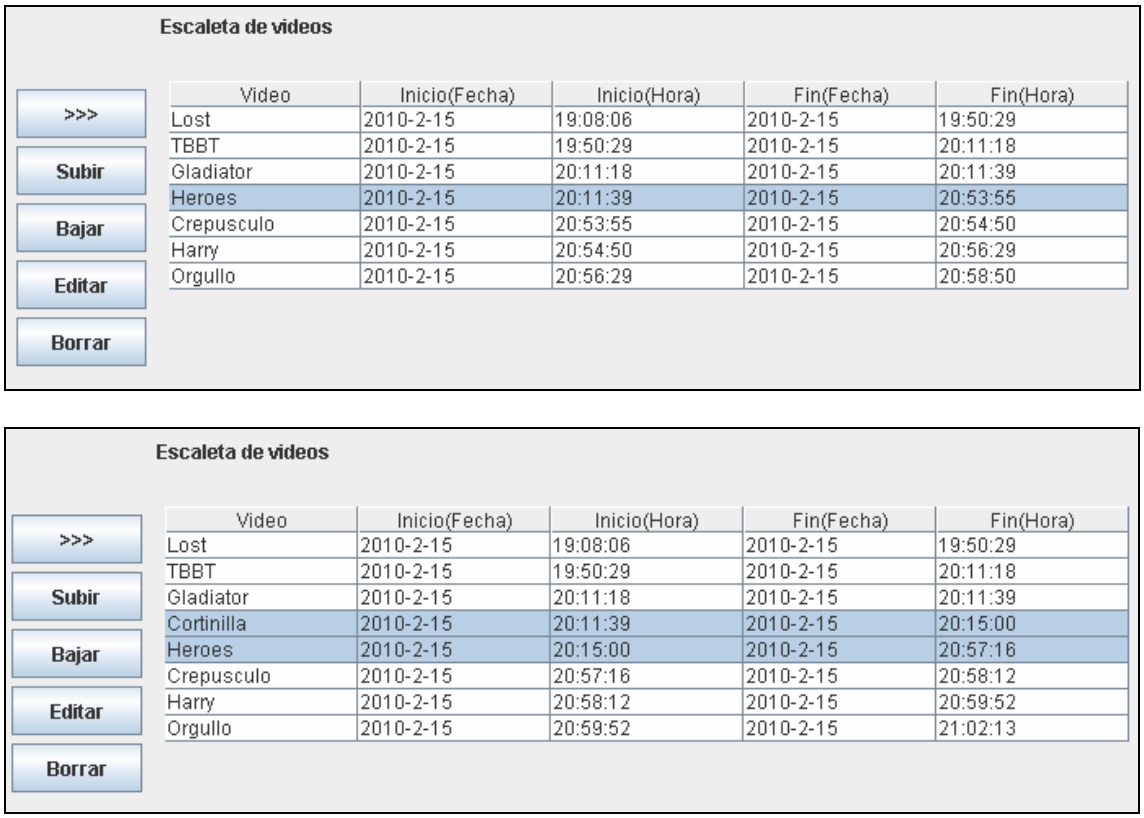


Figura 2.23- *Editar la hora de inicio de un vídeo*

 Tras modificar la hora de inicio de un vídeo la aplicación añade a la tabla una nueva entrada correspondiente a la cortinilla que se reproducirá en el espacio que queda vacío entre los dos vídeos.

2.3.2.5.- Modificar el vídeo de la cortinilla

Como se acaba de comentar, al modificar la hora de inicio de un vídeo, entre el fin de la reproducción de un vídeo y el comienzo del vídeo modificado se genera un espacio de tiempo que no tiene programada ninguna reproducción. Este espacio es rellenado con un vídeo por defecto que hay cargado en la aplicación. Éste vídeo se reproducirá de forma cíclica hasta que comience la reproducción del vídeo modificado.

Este vídeo se puede modificar en cualquier momento, para ello debajo de la lista de vídeos disponibles hay un campo de texto donde se puede introducir el nuevo vídeo de cortinilla.

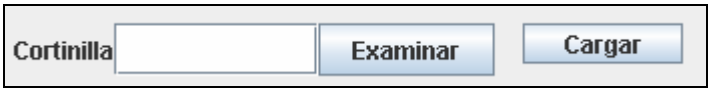


Figura 2.24.- *Modificar el vídeo de cortinilla*

En este campo de texto se debe introducir el Path del vídeo que se quiere cargar. Para facilitar esta operación se dispone de un botón Examinar que abre un explorador en el que se puede seleccionar el vídeo que se quiere cargar.

Una vez seleccionado el vídeo, se presiona el botón Cargar para finalizar la operación.

 Esta operación se puede realizar tanto con la escaleta en reproducción como parada.

2.3.2.6.- Iniciar la reproducción de la escaleta

Para iniciar la reproducción, una vez configurada la escaleta, se presiona el botón

Antes de presionar el Play, se debe asegurar de que la hora de inicio es correcta. La hora de inicio no puede ser anterior a la hora actual porque si no es así, la reproducción no comenzaría nunca. Si la hora de inicio no es correcta, la aplicación aborta la operación.

 Si no hay vídeos en la tabla no se puede iniciar la reproducción. Primero hay que añadir los vídeos que se quieren reproducir, y es deseable que se ordenen antes de iniciar la reproducción.

2.3.2.7.- Parar la reproducción

En cualquier momento de la reproducción se puede presionar el botón 

Esto provoca la interrupción inmediata de la reproducción.

 Al detener de este modo la reproducción se pierde el vídeo que se está reproduciendo y el que se encuentra a la espera para comenzar la reproducción a continuación.

2.4.- Aplicaciones

El otro gran bloque de esta aplicación corresponde a la gestión de las aplicaciones interactivas. Para acceder a esta parte se presiona el botón 

de la ventana de inicio.

Figura 2.25.- *Acceso a la gestión de aplicaciones*

Al presionar este botón, se despliega la ventana Gestión de aplicaciones

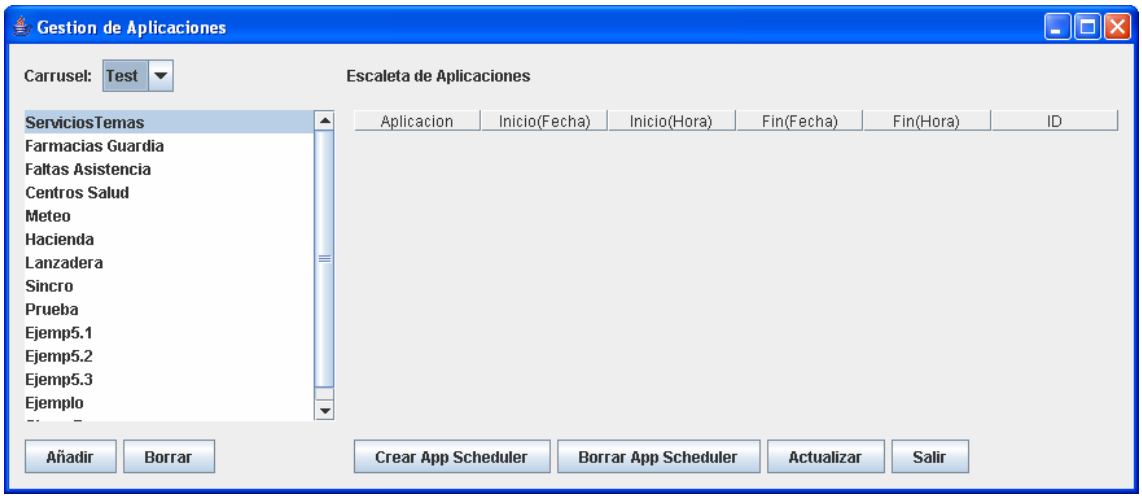


Figura 2.26.- *Ventana Gestión de aplicaciones*

Descripción de la ventana:

- En la parte superior izquierda hay un combo box donde se selecciona el carrusel con el que se va a trabajar.

Figura 2.27.- *Selección del carrusel*

- Bajo este combo box se encuentra la lista de aplicaciones disponibles, es decir, las aplicaciones que hay cargadas en el laboratorio en ese momento.

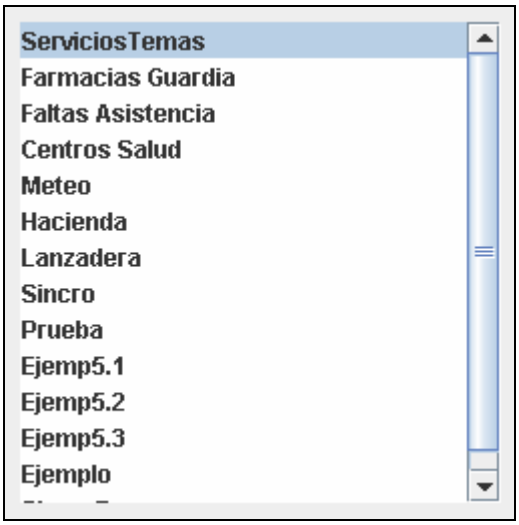


Figura 2.28.- *Lista de aplicaciones disponibles*

- Para terminar con la parte izquierda, en la parte inferior se encuentran los botones para la gestión de la lista de aplicaciones disponibles.
- En la parte derecha se sitúa la tabla que representa el *Schedule* de aplicaciones.

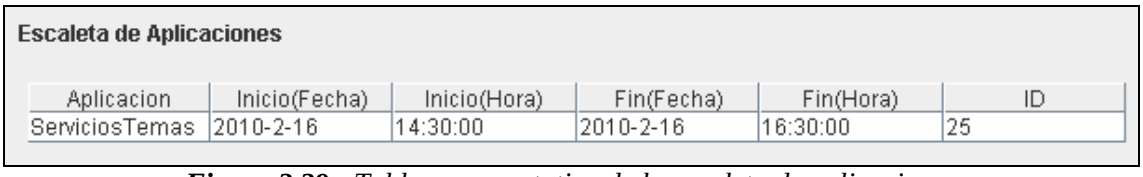


Figura 2.29.- *Tabla representativa de la escaleta de aplicaciones*

- Debajo de esta tabla se encuentran los botones con los que se gestiona la escaleta.

2.4.1.- Gestión de la lista de aplicaciones disponibles

La lista de aplicaciones disponibles se gestiona desde la parte izquierda de la ventana.

Para la gestión de esta parte se tienen 3 acciones posibles:

- Selección del carrusel
- Añadir una nueva aplicación
- Borrar una aplicación MHP

2.4.1.1.- Selección del carrusel

Es importante seleccionar el carrusel con el que se quiere trabajar ya que al cargar una nueva aplicación, los archivos de la aplicación se cargan en el carrusel.

Figura 2.30.- *Combo box para seleccionar el carrusel*

Se selecciona en el combo box el carrusel con el que se va a trabajar.

2.4.1.2.- Añadir una nueva aplicación

Para iniciar la operación se presiona el botón 

lo que provoca que se abra la ventana Nueva Aplicación MHP

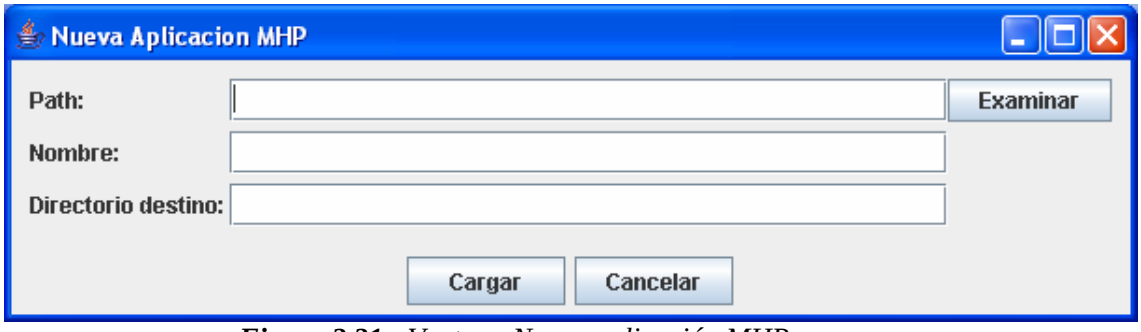


Figura 2.31.- *Ventana Nueva aplicación MHP*

Como se puede ver, esta ventana tiene tres campos que se deben completar con los datos de la nueva aplicación.

- **Path:** En este campo se introduce la dirección donde se encuentra el archivo .zip que se quiere cargar. Para ello, se dispone de un botón Examinar con el que abre un explorador que facilita la selección de la ruta correcta.

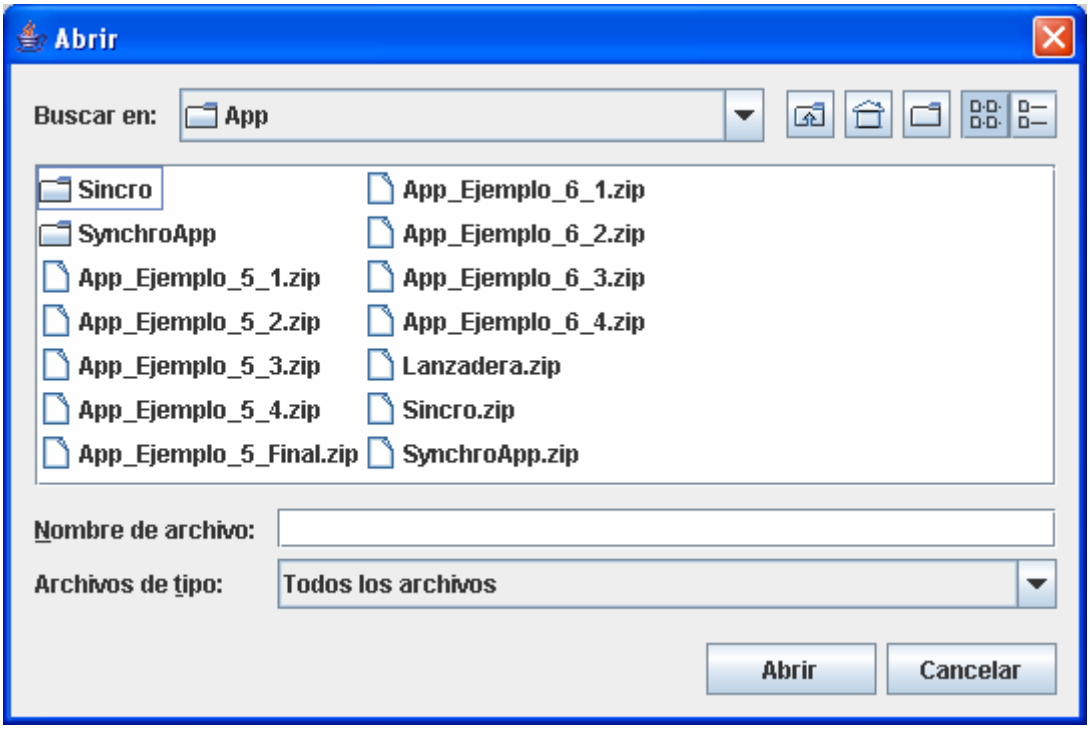


Figura 2.32.- *Explorador para seleccionar la nueva aplicación*

- **Nombre:** Nombre que se le quiere dar a la nueva aplicación. Hay que tener en cuenta que este nombre será el que identificará a la aplicación durante el uso del programa.
- **Directorio destino:** Al cargar la aplicación se descomprime el archivo .zip en el carrusel. Todo su contenido se coloca en el directorio que se especifica en este campo, si el directorio no existe se crea automáticamente. Esto ayuda a tener todos los ficheros contenidos en el carrusel más ordenados.

Una vez que se han rellenado los campos con los datos de la nueva aplicación, se presiona el botón Cargar para subir los ficheros de la aplicación.

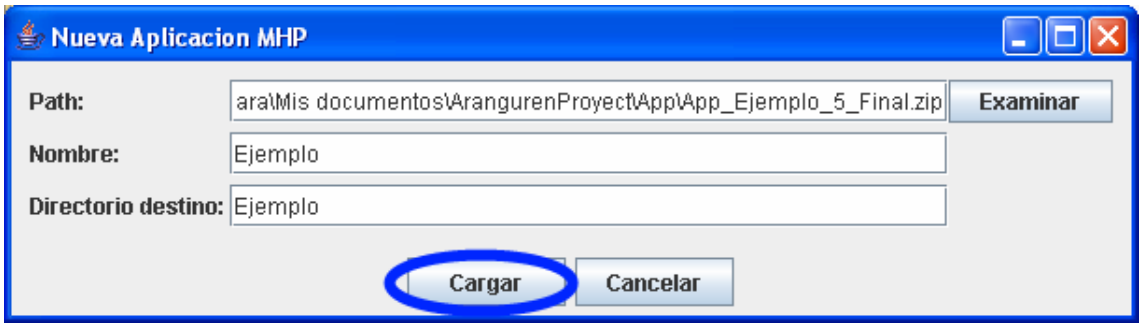


Figura 2.33.- *Carga de una nueva aplicación MHP*

Una vez que se ha descomprimido el archivo de la aplicación y se han subido los ficheros resultantes de la descompresión al carrusel, se abre una nueva ventana donde se selecciona la clase principal de la aplicación.

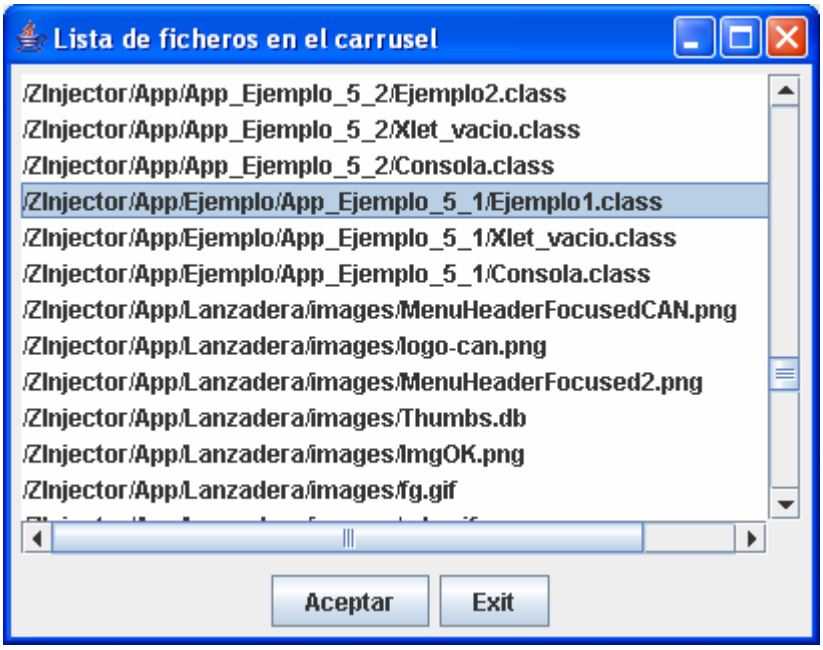


Figura 2.34.- *Ventana Lista de ficheros en el carrusel*

En esta ventana se selecciona la clase principal desde la que se va a lanzar la nueva aplicación. Para seleccionar la clase principal se hace clic con el ratón sobre la lista que contiene todos los ficheros que hay cargados en el carrusel. Una vez seleccionada se presiona el botón Cargar para finalizar la operación.

Para cargar una nueva aplicación se necesita tener un archivo .zip que contenga todas las clases necesarias para la ejecución de la aplicación. También debe contener todos los accesorios necesarios para su correcta visualización como pueden ser imágenes, archivos de propiedades,… Por último, es muy importante que el archivo que se

va a cargar tenga extensión .zip, ya que los ficheros con otras extensiones (como .rar) no son reconocidos por el laboratorio y por lo tanto no se cargan en el carrusel.

2.4.1.3.- Borrar una aplicación MHP

Primero se selecciona la aplicación que se quiere borrar haciendo clic directamente sobre la lista de aplicaciones disponibles.

A continuación se presiona el botón 

para confirmar la operación.

2.4.2.- Escaleta de aplicaciones

La configuración de la escaleta de aplicaciones se gestiona desde la parte derecha de la ventana.

En este caso también se dispone de 3 acciones posibles para la gestión de la escaleta:

- Crear un *Schedule* de aplicación
- Borrar un *Schedule* de aplicación
- Actualizar la escaleta de aplicaciones

2.4.2.1.- Crear un Schedule de aplicación

Para acceder a esta funcionalidad se presiona el botón 

que abre una nueva ventana (Nueva entrada en la escaleta de aplicaciones).

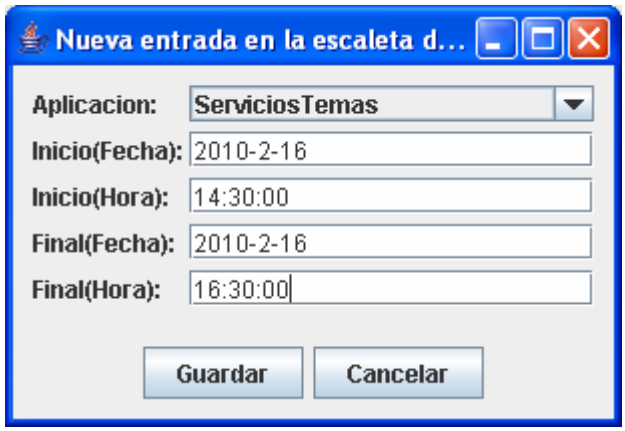


Figura 2.35.- *Ventana Nueva entrada en la escaleta de aplicaciones*

En primer lugar se selecciona la aplicación que se quiere programar en el combo box de la parte superior de la ventana.

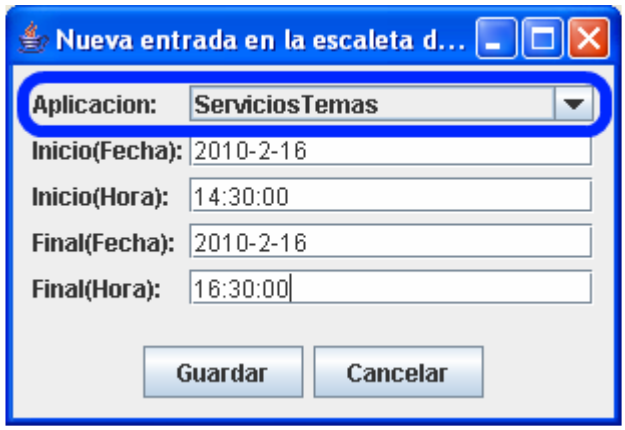


Figura 2.36.- *Selección de la aplicación que se a a programar*

Se debe tener en cuenta que para poder programar una aplicación, previamente se debe añadir la aplicación a la lista de aplicaciones disponibles. Si no existe en la lista no se puede programar.

A continuación se introducen las fechas y las horas entre las cuales se quiere que la aplicación esté disponible en los campos Inicio y Fin.

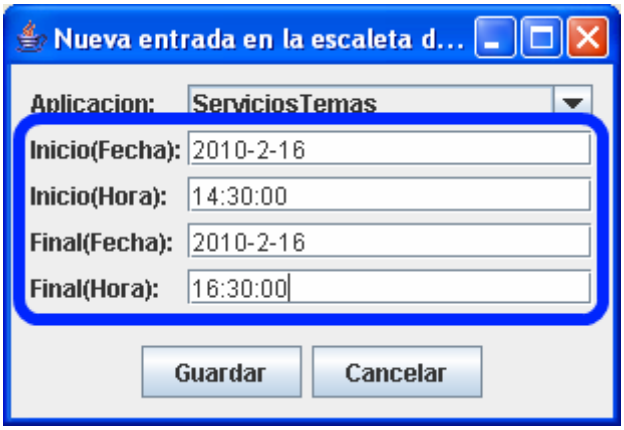


Figura 2.37.- *Campos Inicio y Fin del Schedule de aplicaciones*

 Es importante que las fechas y las horas se introduzcan con el formato adecuado:

- Fecha: Año-Mes-Día (separados por un guión (-))
- Hora: Hora:Min:Seg (separados por dos puntos (:))

Para finalizar se presiona el botón Guardar, de modo que si todos los datos son correctos, se cerrará la ventana y se añadirá una fila a la tabla.

2.4.2.2.- Borrar un Schedule de aplicación

Se selecciona el *Schedule* que se quiere borrar haciendo clic sobre la tabla.

Para confirmar la operación se presiona el botón 

2.4.2.3.- Actualizar la escaleta de aplicaciones

Al presionar el botón 

se eliminan de la tabla todas las entradas que hayan caducado, es decir, que su intervalo de disponibilidad ya ha pasado.

2.5.- Salir de la aplicación

Para salir de la aplicación, basta con presionar el botón Salir de la ventana de inicio. Automáticamente se cierran todas las ventanas de la aplicación que se tengan abiertas.

Figura 2.38.- *Salir de la aplicación*

Si hay una reproducción en curso es importante pararla antes de salir de la aplicación. Si no se hace así el vídeo que está en ese momento en reproducción se queda reproduciéndose de forma cíclica.

3.- CONCLUSIONES Y LÍNEAS DE FUTURO

3.1.- Conclusiones

En este capítulo se van a describir las reflexiones a las que se han llegado tras la realización de este proyecto. En primer lugar se va a hacer una valoración general de lo que ha supuesto para mi la realización de este proyecto. A continuación se comentarán las conclusiones particulares sobre el desarrollo del proyecto. Por ultimo, se presentarán las posibles líneas futuras que puede seguir este proyecto, es decir, como se puede mejorar lo ya hecho y algunos módulos complementarios que se pueden desarrollar.

En general, el desarrollo del proyecto ha supuesto la puesta en práctica de todas las habilidades adquiridas durante la carrera. Este es un trabajo donde no sólo se utilizan los conocimientos a nivel académico, sino que se prueba la capacidad de encontrar soluciones a problemas reales y saber como implementarlos o gestionarlos. A parte de poner en práctica muchas de las cosas aprendidas durante la carrera, este proyecto ha supuesto un continuo proceso de aprendizaje. En este proceso he aprendido distintas tecnologías que eran necesarias para el desarrollo de las diferentes partes del software que se ha implementado.

En mi caso, el hecho de que el proyecto esté basado en un problema real ha supuesto una motivación extra en todo este proceso. Esto le da más credibilidad a todo lo que estas haciendo, además, he tenido la suerte de poder testear todo lo que estaba haciendo en un entorno realista lo me ha permitido enfrentarme a problemas reales y no sólo a suposiciones. Además, la solución a la que se ha llegado está lista para ser utilizada en un entorno real, y esto resulta bastante gratificante.

Este proyecto me ha resultado muy interesante, entre otras cosas, porque me ha permitido profundizar en el mundo de la televisión digital. Este tema me ha interesado siempre, pero ahora con los conocimientos que he adquirido tras la realización de este proyecto me siento mucho más atraída por él. También hay que tener en cuenta que es un tema de mucha actualidad con la inminente llegada del apagón analógico, que abrirá las puertas a un nuevo concepto de televisión mucho más activo que el conocido hasta ahora.

Otro punto a destacar es que el hecho de haberlo desarrollado en una empresa como CEIN me ha abierto los ojos a la manera de trabajar en la empresa, que difiere bastante de lo vivido en la universidad. Esta ha sido una experiencia muy positiva, de la que he

aprendido mucho, tanto académicamente como desde el puto de vista de la vida laboral. También me ha servido para adquirir cierta experiencia, que no hubiera conseguido de no haber estado inmersa en una empresa.

A continuación se van a comentar las conclusiones más técnicas a las que se ha llegado tras el desarrollo del proyecto:

- Este proyecto me ha supuesto un importante aprendizaje en el lenguaje de programación Java, ya que la implementación del software se ha hecho en este lenguaje. Al inicio de este proyecto, mis conocimientos sobre la programación orientada a objetos eran bastante escasos, así que fue necesario invertir un poco de tiempo en adquirir conocimientos sobre este tema antes de comenzar de lleno con la parte creativa.
- También tuve que profundizar en el tema de la televisión digital, entre otras cosas en el estudio de la API MHP. Este tema me interesaba por dos razones, por un lado, necesitaba saber cómo gestionar y ejecutar aplicaciones MHP, para poder implementar la gestión de este tipo de aplicaciones en la herramienta que se ha desarrollado. Por otro lado, se ha creado una sencilla aplicación MHP como ejemplo de sincronización de las aplicaciones con los contenidos del canal, por lo que también me interesaba conocer cómo implementar una aplicación MHP.
- Otro de los conceptos importantes que he tenido que estudiar son los *Stream Events*. En el proyecto se ha implementado un ejemplo de sincronización de las aplicaciones MHP con los contenidos audiovisuales, para implementar esta parte primero tuve que investigar sobre el uso y las características de estos eventos ya que, como se ha comentado en algunas partes de la memoria, esta sincronización entre aplicaciones y contenidos se realiza mediante el uso de *Stream Events*.
- Para la realización de este proyecto, se ha utilizado el laboratorio multiplexor **iMux Multiplexer** de la empresa alemana MIT-Xperts. Así que también he tenido que aprender a manejarlo, saber lo que es capaz de hacer y, sobre todo, cómo lo hace para poder implementar las funcionalidades de la herramienta. También he tenido que investigar cómo funciona la Java API del laboratorio, que es el instrumento con el que se comunica el software con el laboratorio.
- Se ha hecho uso de la librería Java ***Jsch***, se trata de una implementación pura en Java de SSH2. Esta librería permite ejecutar comandos de Shell desde la aplicación Java. El uso de este paquete es necesario para solventar una de las

limitaciones impuestas por el **iMux Multiplexer**. Este laboratorio no permite obtener el tamaño de los vídeos que ya están guardados en la base de datos del laboratorio. Este dato es necesario para la implementación de la escaleta de vídeos, ya que a partir del tamaño se calcula la duración de los vídeos, y esta duración a su vez se utiliza para saber cuándo termina la reproducción de cada vídeo. Para solucionar este problema hubo que acceder al sistema de archivos del **iMux** mediante el uso de la librería ***Jsch***.

- También se ha realizado un estudio sobre las distintas herramientas que se pueden utilizar para la conversión de los vídeos a distintos formatos. Para poder subir los vídeos al **iMux Multiplexer** estos tienen que estar en un formato adecuado que el laboratorio pueda detectar, en concreto deben ser de tipo MPEG Transport Stream. Esta limitación provocó la realización del estudio sobre las herramientas de conversión de vídeo. Se tuvieron que hacer muchas pruebas con distintos software de conversión hasta dar con los parámetros adecuados para una correcta reproducción en los receptores.

Como se puede ver este proyecto ha contado con una importante etapa de aprendizaje en la que los resultados no eran muy visibles pero no por ello menos importantes. Sin este periodo de investigación y aprendizaje no hubiera sido posible desarrollar el resto del proyecto.

3.2.- Líneas de futuro

En cuanto a las líneas futuras del proyecto hay varias mejoras o complementos que se podrían realizar en un futuro:

- En primer lugar, yo creo que una ampliación interesante sería completar la parte referente a los Stream Event. En este proyecto se ha dado una breve pincelada a este tema, pero sería interesante implementar una tercera interfaz desde la que el usuario pudiera gestionar el envío de Stream Events, tanto de tipo *Do it now* (que ya implementa el iMux) como de tipo *Scheduler* (el iMux no implementa el envío de estos eventos).
- Otra mejora interesante sería extender el uso de esta aplicación a otros tipos de laboratorios. Para conseguir esto bastaría con modular el programa por medio de una clase abstracta o una interfaz Java que defina los métodos principales de la aplicación. Estos métodos se implementarían en paquetes distintos según el tipo de laboratorio para el que se esté trabajando.

- También podría resultar interesante hacer una versión web de la aplicación para aumentar la accesibilidad del software. La principal ventaja de este tipo de aplicaciones online es que el usuario puede acceder a ellas desde cualquier máquina que tenga acceso a Internet. Si se tiene una red local la solución podría estar en usar **Java Web Start**, con esto lo que se consigue es tener la aplicación instalada en un servidor web y que todos los usuarios puedan acceder a ella sin necesidad de tenerla instalada en su equipo. De esta forma también se consigue que todos los usuarios estén utilizando la misma versión, y a la hora de actualizarla sólo es necesario hacerlo en un único sitio. Esta seria la opción que supone menos cambios en la aplicación original, pero existen otras muchas opciones para migrar la aplicación de escritorio a un entorno web.
- Por otro lado, también se pueden realizar mejoras en el propio software. Por ejemplo:
    - Se podría implementar un sistema de ventanas informativas que indiquen al usuario de porque ha fallado alguna de las operaciones realizadas en el software. Por ejemplo, si se intenta comenzar la reproducción de una escaleta y la hora de inicio no es correcta se desplegaría una ventana que diga: “La hora de inicio es errónea”.
    - También se puede implementar un filtro para que en la lista de ficheros cargados en el carrusel sólo se muestren los ficheros que pertenecen al archivo .zip que se acaba de cargar. De este modo, se facilita la selección de la clase principal de la aplicación MHP que se está creando.
    - Otra mejora posible es la de mostrar las listas de vídeos y aplicaciones disponibles ordenadas alfabéticamente. Esto facilitaría el uso del software al usuario.

REFERENCIAS BIBLIOGRÁFICAS

Libros

[BLO 01] Joshua Bloch , *Effective Java*, Addison Wesley, 2001
ISBN: 0-20-131005-8

[ECK 03] Eckel Bruce *Thinking in Java*, Prentice Hall, 2003
ISBN: 0-13-100287-2

[LOY 02] Marc Loy… [et al], *Java Swing*, O´Reilly, 2002
ISBN: 0-596-00408-7

[MOR 05] Steven Morris, Anthony Smith-Chaigneau, *Interactive TV Standards (A Guide to MHP, OCAP and Java TV)*, Focal Press, 2005
ISBN: 0-240-80666-2

[ROB 03] Matthew Robinson, Pavel Vorobiev, *Swing*, Second Edition, Manning Publications Co., 2003
ISBN: 1-930110-88-X

Otras referencias y enlaces de interés

- Interactive TV Standars: http://www.interactivetvweb.org/
- Licencia BSD: http://es.wikipedia.org/wiki/BSD_license
- Librería Jsch (Java Secure Channel) : http://www.jcraft.com/jsch/
- MHP: http://www.mhpkdb.org/publ/mhp-guide.pdf
  - http://www.mhp.org/
- TVDI (Universidad de Vigo): http://tvdi.det.uvigo.es/es/index.html
- Xlet: http://www.xlet.net/

GENERACIÓN DE TRAMAS DE TRANSPORTE

Como ya se comentó en la memoria, uno de los problemas surgidos en el desarrollo de este proyecto tenía que ver con la codificación de video, más concretamente con la generación de tramas de transporte. Estas tramas debían ser cargadas en un laboratorio multiplexor el cual se encarga de transmitirlas como una señal de televisión convencional. La señal llega a los receptores de TDT que tras descodificarla la muestra por pantalla. El problema que surge es que las tramas que se estaban generando se veían correctamente cuando se reproducían en el PC, pero cuando se intentaban ver en la televisión dejaban de funcionar. Así que se decidió centrar los esfuerzos en resolver este problema antes de continuar.

Lo primero que se tenía que hacer era diseñar una estrategia de cómo abordar el problema, ya que hasta el momento se estaban dando palos de ciego alrededor del problema sin ver una regla de comportamiento clara. En este caso se decidió tomar un video de prueba y probar únicamente con el, además se intentó buscar otras herramientas que nos permitiesen modificar más parámetros de los que habíamos probado hasta ahora.

Descripción de las soluciones estudiadas.

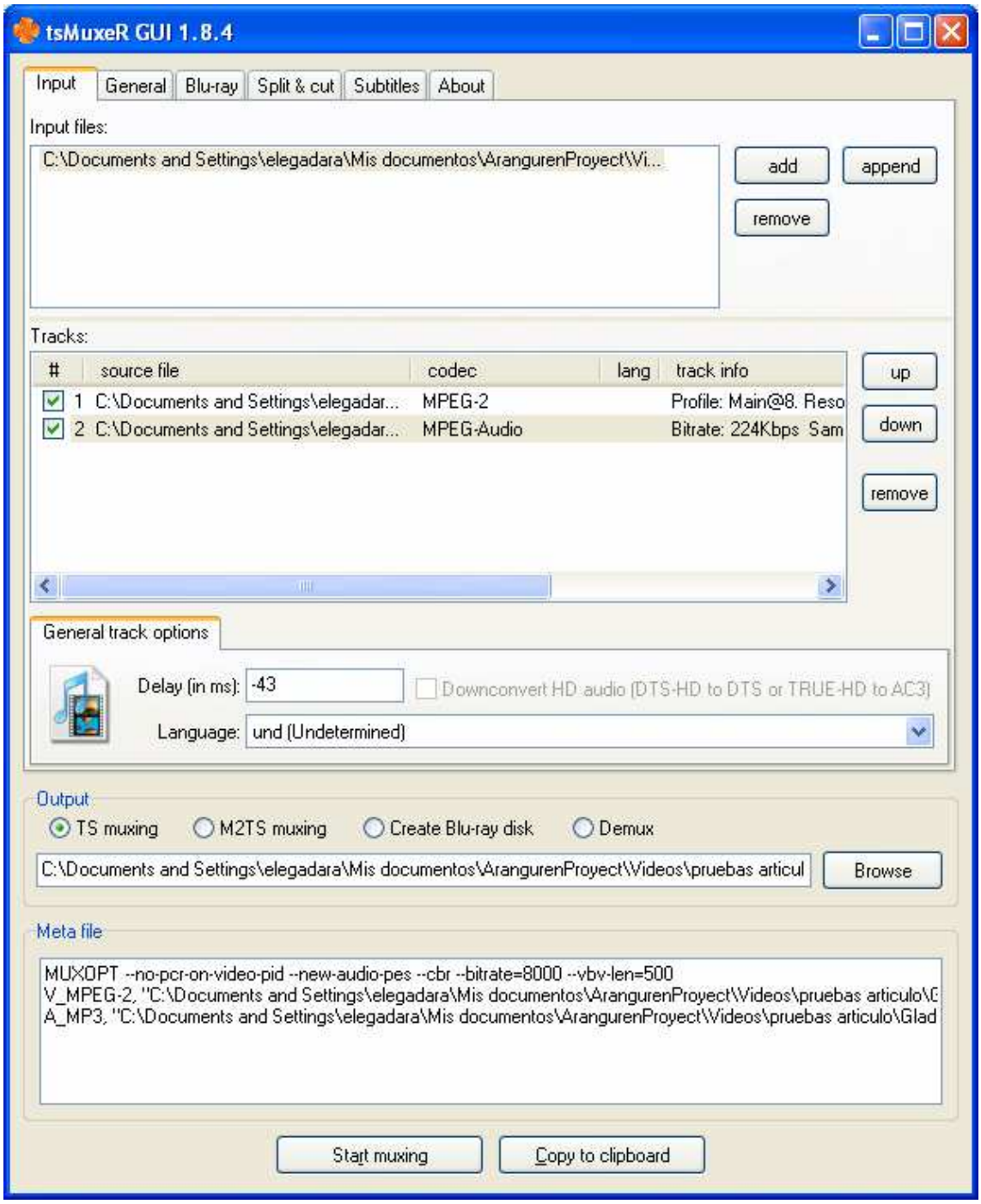
Como ya se ha comentado, se va a tomar un video de prueba con el que se probarán todas las soluciones que se contemplen. Este video va a ser una muestra de corta duración para que los procesos de codificación y encapsulado sean más rápidos y así poder realizar más pruebas. Como la idea era llegar a una configuración lo más general posible, una vez que consigamos hacer funcionar el video de test, se va a probar con videos de otros formatos para ver si se conservan los resultados, y así ser capaces de transmitir videos provenientes de cualquier fuente.

A continuación se van a describir 3 soluciones distintas a este problema:

- tsMuxer v.1.8.4: software libre.
- Super de eRightSoft (versión 2008): software libre
- Main Concept Reference Encoder: Software con licencia

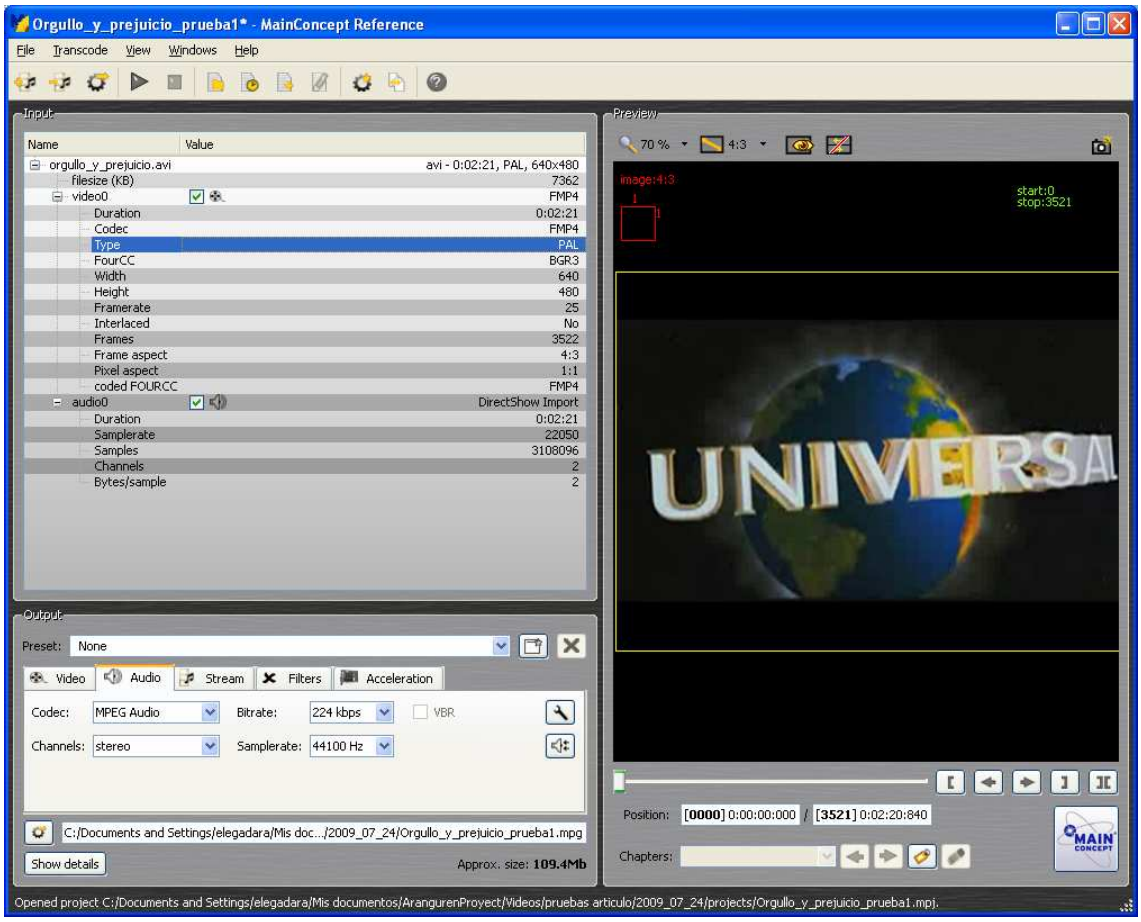
1.- El primero de los software que se ha usado para generar las tramas de transporte es el tsMuxer v.1.8.4. Se trata de un software de licencia libre que encapsula los datos de un archivo de video generando una trama de transporte. Este software genera ficheros .ts

(tramas de transporte) con el audio, el video y alguna tabla (PMT, PAT y SIT) estos archivos están ya listos para subirlos al laboratorio. Uno de los problemas que plantea este software es que sólo acepta archivos de entrada con formato mpeg, de modo que si se quiere codificar videos que estén en cualquier otro formato se va a necesitar necesitaremos otro programa que haga el paso previo de transformar el video a mpeg desde cualquier otro formato. Como la idea era buscar la solución más general posible, se va a necesitar un software que prepare los videos para ser multiplexados por el tsmuxer.



2.- Otra posible solución que se ha probado es el software Super de eRightSoft (versión 2008) que si que permite con una sola herramienta codificar cualquier tipo de formato de video en una trama de transporte. Se trata de un programa gratuito para la conversión de archivos de audio y video a diferentes formatos, obteniendo una amplia gama de formatos a elegir para convertir.

3.- Como tercera y última solución se ha probado el software Main Concept Reference Encoder, en este caso no es un software gratuito, pero en comparación con el resto es bastante más completo, sobre todo en lo que respecta a la elección de parámetros para la configuración del fichero de salida, da muchas más opciones y muchos más parámetros con los que probar para una codificación de mejor calidad.



Resultados obtenidos con las distintas soluciones

Las primeras pruebas, en todas las soluciones, se van a realizar con un video que tiene los siguientes parámetros:

- Video: Codec mpeg-2,
  - Relación de aspecto 4:3,
  - Resolución 720x576,
  - Bitrate 6Mbps,
- Audio: Codec mpeg-audio (MP3),
  - Bitrate 224Kbps,
  - simple rate 48000

Descripción de los resultados obtenidos con el tsMuxer:

Se carga el video se prueba y se comprueba que detecta correctamente todos los parámetros del fichero de entrada. Esta aplicación tiene muy pocos parámetros con los que jugar, así que se ajustan los que nos permite, entre ellos esta el bitrate, de modo que ponemos un Bitrate constante a 4Mbps. Se transmite el fichero de salida, pero vemos que no funciona, al video le falta continuidad y el audio se oye muy mal. Se observa el fichero de metainformación que ha generado la aplicación junto con la trama de transporte y dice que no genera PCRs en la PID de video, así que difícilmente va a funcionar sin PCRs. Con esta característica queda prácticamente desechada esta solución.

Por varios motivos se puede decir que no es una buena solución. En primer lugar, las opciones que tiene para configurar el fichero de salida son bastante escasas, y no permiten hacer prácticamente nada. Para obtener una trama de transporte con los parámetros que queremos, el fichero de entrada debe estar configurado ya con los parámetros adecuados.

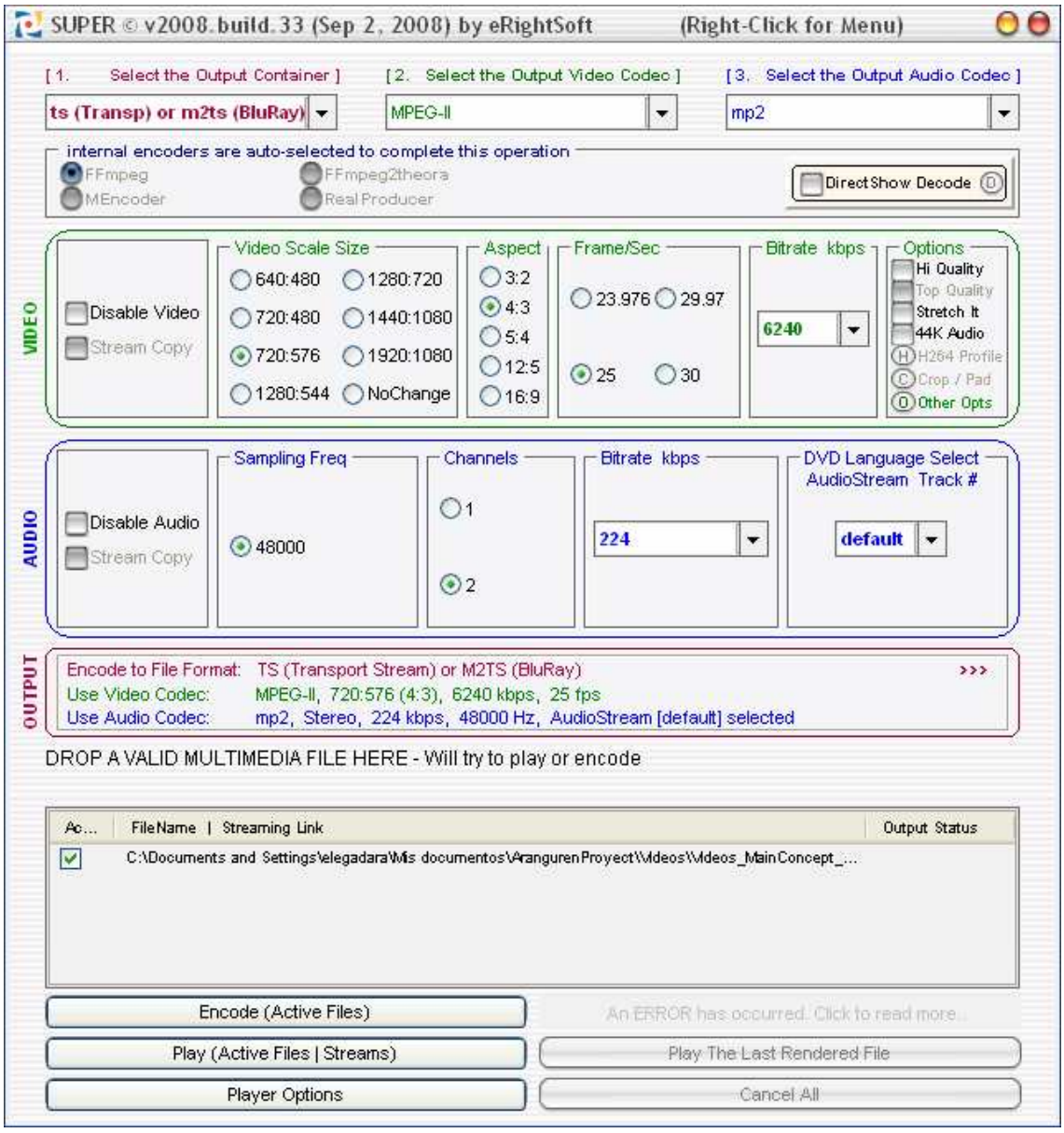
Por otro lado esta el problema de los PCRs, el hecho de que la aplicación no genere los PCRs para la PID de video hace del todo inservible la herramienta para el uso que le queremos dar en este caso. Si la reproducción de esos archivos se va a realizar, por ejemplo, desde un pc esto no tiene tanta importancia ya que este lo puede reproducir perfectamente sin necesidad de PCR, pero en el caso que nos ocupa que es la recepción en decodificadores de TDT para su posterior visualización en una televisión no es posible la reproducción sin PCRs.

Descripción de los resultados obtenidos con Super:

Otra herramienta de Software libre que se ha probado es el Super de eRightSoft. Esta aplicación tiene más opciones de configuración que el tsMuxer, con lo que permite un número mayor de posibilidades a la hora de generar las tramas de transporte. Además, otra ventaja de este software sobre el tsMuxer es que los ficheros de entrada y los de salida pueden tener prácticamente cualquier formato, a diferencia del tsMuxer que sólo aceptaba ficheros mpg como entrada, y a la salida sólo te permitía 4 opciones (ts, m2ts, Blu ray disck y demux).

Se carga el fichero de prueba, y se comprueba que lo carga sin problemas, reconoce todos los parámetros del fichero de entrada. Definimos los parámetros del fichero de salida:

- Output: Contenedor de salida: Transport Stream;
  - Video: resolución 720x576,
    Relación de aspecto 4:3,
    25 Frames/Sec,
    Bitrate 6240kbps;
  - Audio: Sample rate= 48000Hz,
    Bitrate = 224kbps.



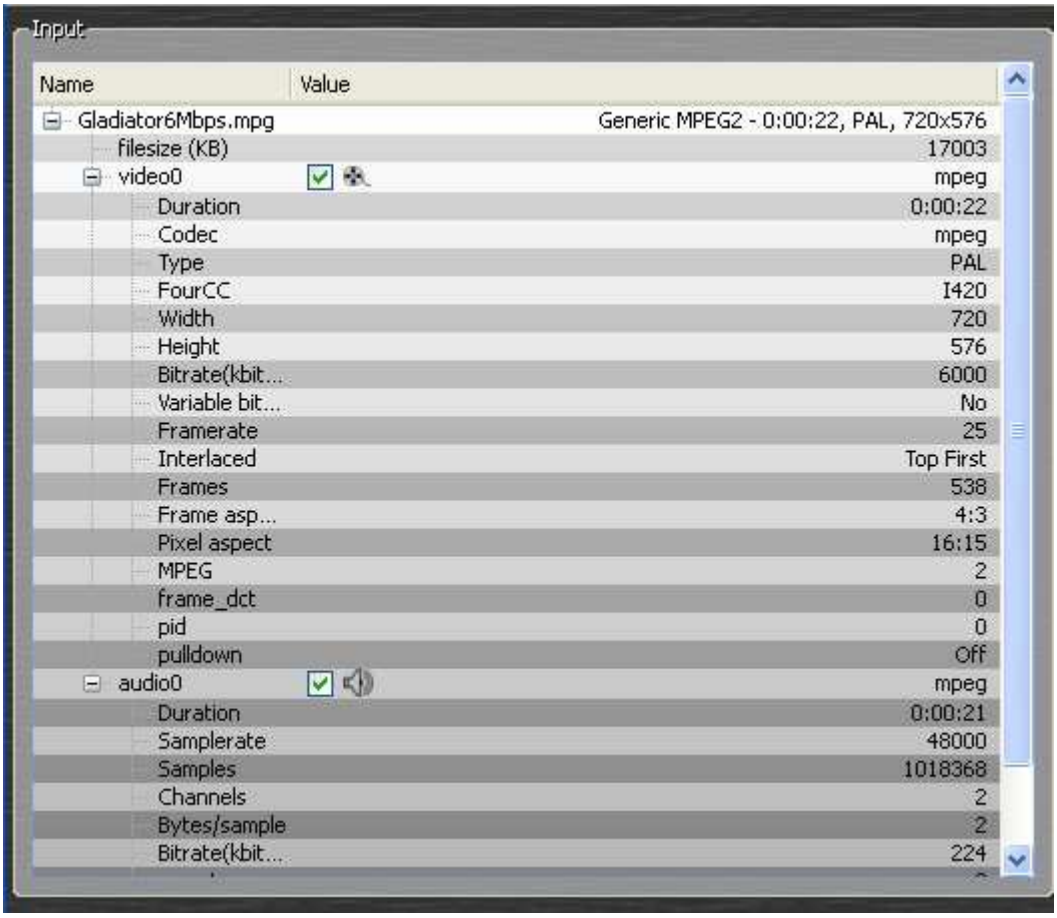
A la salida se comprueba que se ha codificado correctamente y se visualiza con un reproductor multimedia en el PC, todo parece correcto así que se prueba a transmitirlo a ver que nos encontramos en el receptor. El resultado no es el esperado, ya que el receptor no es capaz de decodificarlo correctamente, tanto el audio como el video tienen problemas de decodificación.

No se sabe muy bien porque esta trama de transporte no funciona, en principio los parámetros son los que deberían ser, de todas formas se hacen más pruebas variando los parámetros a ver si se da con los que hacen que funcione, pero no se consigue nada. Analizando los parámetros que te permite variar se ve que faltan algunos importantes, por

ejemplo el Bitrate, te deja seleccionar el valor pero el modo no, como se verá más adelante es muy importante que el Bitrate este en modo CBR (Constant Bitrate). Es muy probable que esta aplicación tome por defecto el modo VBR (Variable Bitrate), por la experiencia adquirida con otras aplicaciones se sabe que la mayoría toma por defecto el modo variable, de manera que si en este caso no permite cambiar el modo del bitrate podemos pensar que se está tomando un bitrate variable, así que este puede ser uno de los motivos por los que no funcionan las tramas de transporte generadas con esta herramienta. Por otro lado, tampoco se sabe que hace con los PCRs, recordemos que la aplicación anterior (tsMuxer) no los generaba lo que implicaba que las tramas no funcionasen, y tuvieran problemas con la sincronización, en este caso es posible que tampoco genere los PCRs, pero no se puede asegurar porque el software no nos da ningún tipo de información sobre estos. Aunque no se pueden asegurar los motivos por los que no funciona, porque nos falta información de cómo funciona el software, estas dos razones tienen muchas posibilidades de ser las culpables de que los TS no funcionen.

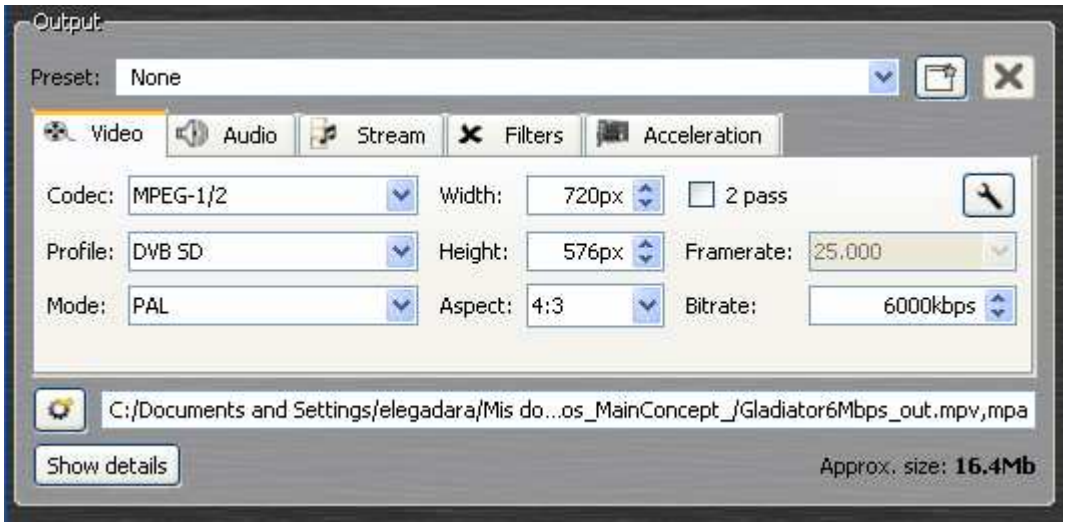
Descripción de resultados para Main Concept:

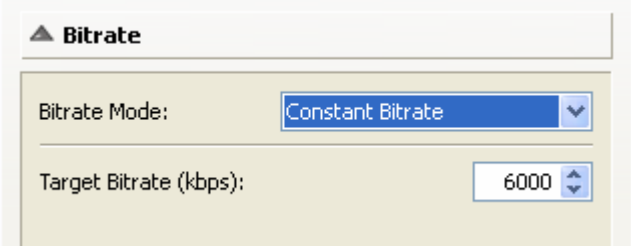
Como en los casos anteriores se comienza probando con el video de test (video Mpeg-2, con un Bitrate de 6Mbps).

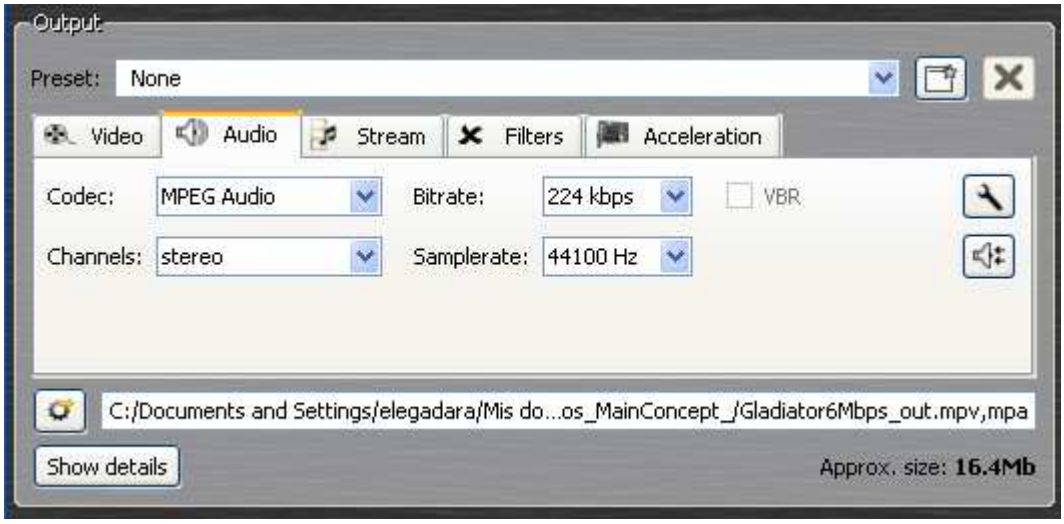


A continuación se describen algunas de las pruebas realizadas con el Main Concept:

Prueba1: el fichero de salida es una trama de transporte con Constant Bitrate a 6Mbps para el video, el audio va a 224kbps y 44.1kHz, perfil DVB SD y resolución 720x576. Tras subir esta trama al laboratorio para que sea transmitida se comprueba que funciona a la perfección.







Prueba2: la misma configuración que la anterior pero con Bitrate Variable. Se comprueba que la trama deja de funcionar, se ve y se oye mal. Se prueba con dos tipos de

receptores, un Engel TDT 6000i y un iCAN Osmosis 1.02 (MHP Receiver T75-DEV) y ninguno de los dos detecta el Bitrate Variable.

Prueba3: la misma configuración que en la prueba1 pero cambiando la relación de aspecto a 16:9. Igual que antes se prueba en los dos receptores, en el Engel se ve con menos resolución vertical y en el iCAN se ve igual que con relación de aspecto 4:3. Es decir, que el primero detecta la relación de aspecto y el segundo no la considera.

Prueba4: Una vez comprobado que funciona correctamente con el video de prueba, se intenta tomar otros formatos para el archivo de entrada para ver si se ha llegado a una solución general.

Se prueba con dos nuevos archivos con formatos diferentes:

Formato1: archivo .flv,
- Video: codec flv1
  - PAL
  - Resolución 300x240
  - Relación de aspecto 5:4
  - Frame rate 25 fps
- Audio: codec DirectShow Import
  - Sample rate 22050Hz

Formato2: archivo .avi
- Video: codec fmp4
  - PAL
  - Resolución 640x480
  - Relación de aspecto 4:3
  - Frame rate 25 fps
- Audio: codec DirectShow Import
  - Sample rate 22050Hz

En ambos casos se configura el fichero de salida con los mismos parámetros:

Output: MPEG Transport Stream
- Video: codec MPEG-1/2
  - Perfil DVB-SD
  - PAL
  - Bitrate 6Mbps CBR
  - Resolución 720x576
  - Relación de aspecto 4:3
  - Frame rate 25fps
- Audio: codec MPEG Audio
  - Bitrate 224KBps
  - Sample rate 44100Hz

Se transmiten para comprobar su correcto comportamiento, en este caso no hay sorpresas y se reciben perfectamente en los receptores.

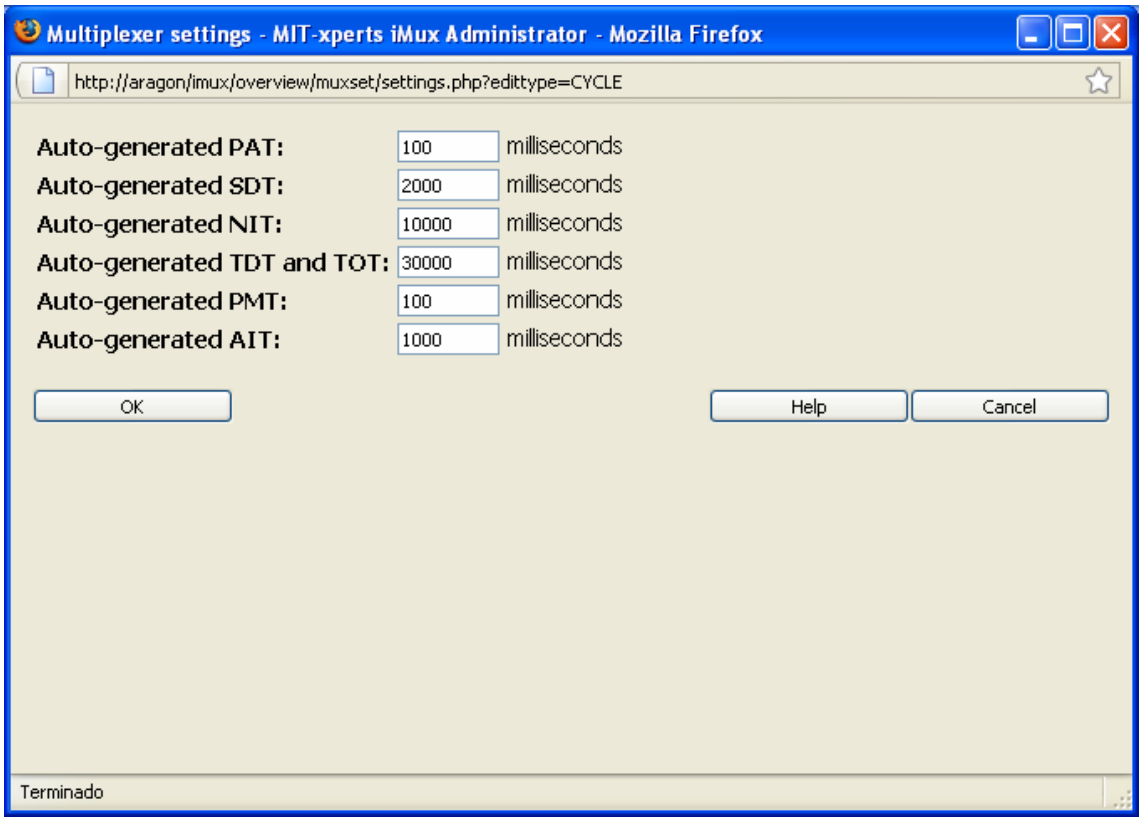
Comprobando que funciona para estos otros dos formatos, se concluye que no es necesaria ninguna otra herramienta para hacer una transformación intermedia, es decir, sólo con el Main Concept puedes tomar tu fichero multimedia de entrada, esté en el formato que esté, y el software te devuelve la trama de transporte lista para ser cargada en el laboratorio, quién se encargará de transmitirla

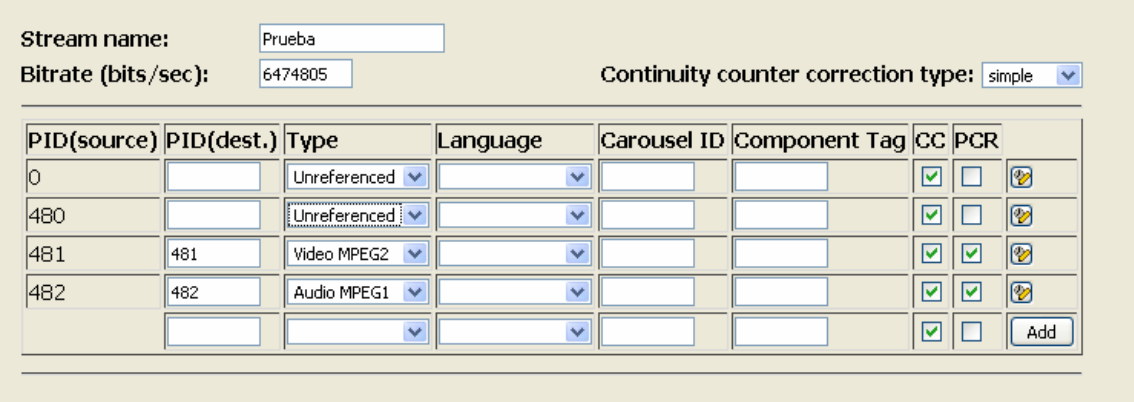
Para finalizar, se ha probado un último software, el E.M. Total Video Converter v3.21. Es también un software libre, en este caso se han hecho un par de pruebas con los parámetros usados en las pruebas anteriores, pero no se ha conseguido hacerlas funcionar. Por lo que se ha desechado esta herramienta también.

Algunas pruebas con la configuración del iMux

Aunque nos hemos centrado en los software de conversión de video, otro bloque importante de la cadena es el propio laboratorio. En este caso se trata del iMux (versión 1.4.0) de Mit-Xperts, un laboratorio de multiplexación de videos y aplicaciones MHP.

Se han hecho algunas pruebas para ver como influye la configuración del laboratorio en la transmisión de las tramas. Tal como se tiene configurado el laboratorio sólo necesita que el Transport Stream contenga los tracks de video y de audio ya que el propio laboratorio autogenera las tablas necesarias para la transmisión. Aunque el software al crear las tramas de transporte incluya alguna tabla, la opción más acertada es deshabilitarlas en la pestaña “Transport Stream” del iMux. En el caso de la PMT, si no se deshabilita el laboratorio tomará la que se ha incluido con la trama, pero en ese caso puede que se pierda información porque si, por ejemplo, se incluye alguna aplicación MHP en la transmisión esta no estará incluida en la tabla ya que se ha añadido después de cargar la trama en el laboratorio. Por esta razón, en este caso, es más seguro dejar que el laboratorio regenere las tablas. Para ello, una vez cargado el archivo en el laboratorio, se borran todas las PID destino, excepto las de los tracks de video y audio.





Por otro lado está la selección de la PID que lleva los PCR. Tras realizar varias pruebas seleccionando distintas PIDs se llega a la conclusión de que no importa que esta PID seleccionada no sea correcta, ya que el laboratorio tiene la capacidad de detectar la PID que contiene los PCRs. No se ha podido probar, pero es posible que también los regenere al igual que hace con las tablas.

Conclusiones

Tras el estudio realizado, se concluye que se optará por el uso del software Main Concept como solución definitiva. Aunque se trata de un software que precisa del pago de una licencia, los resultados son notablemente mejores que las soluciones compuestas por software libre, las cuales no han dado resultados aceptables para las características buscadas en este proyecto.

Estos software libres están bastante limitados en la configuración de los ficheros de salida. La percepción que se tiene de este tipo de herramientas es que están configuradas con los parámetros por defecto definidos para que se puedan usar sin tener mucha idea de la manipulación de ficheros multimedia, su uso está dirigido sobre todo a gente que busca por ejemplo cambiar el formato de una película que quiere ver en una videoconsola (PS, XBOX, PSP,….), en un dispositivo móvil como puede ser un teléfono móvil,… pero que en ningún caso tienen porque tener conocimientos técnicos sobre los distintos formatos usados por estos dispositivos.

Después de estudiar las pruebas que se han realizado y analizar detenidamente los resultados obtenidos podemos concluir que los parámetros más importantes o cuyos efectos son más visibles son dos:

- el tamaño de los frames (que deben ser 720x576 que es el tamaño indicado para el estándar DVB en un sistema PAL)
- y por otro lado el Bitrate, que debe ser constante porque el Bitrate variable no lo detectan los receptores. Este ha sido uno de los principales problemas que se han tenido con el resto de los software ya que por defecto te lo codifica con Bitrate variable. De hecho el Main Concept también toma por defecto el Bitrate Variable y hay que ir a las opciones avanzadas de video para cambiarlo a constante.

Por otro lado, hemos comprobado que la influencia de la relación de aspecto depende del receptor, hay receptores que la detectan y otros que la obvian. Dependiendo de las necesidades del usuario se podría jugar con la configuración de los receptores para solventar este problema pero la transmisión (que es lo que interesa en este momento) no se ve afectada por este parámetro.