Universidad de Alcalá

Escuela Politécnica Superior

Grado en Ingeniería Telemática

Trabajo Fin de Grado

Sistema de visualización de flujos de vídeo en Java en tiempo real

Autor: Francisco Javier López Márquez

Director: Roberto Javier López Sastre

TRIBUNAL:

Presidente: Pedro Gil Jiménez

Vocal 1º: Francisco Javier Escribano Aparicio

Vocal 2º: Roberto Javier López Sastre

CALIFICACIÓN:..... FECHA:.....

Sistema de visualización de flujos de vídeo en Java en tiempo real

Francisco Javier López Márquez

15 de septiembre de 2014

El secreto de la libertad radica en educar a las personas,
mientras que el secreto de la tiranía está en mantenerlos ignorantes.
-Maximilien Robespierre-

Agradecimientos

A todas las personas que han estado a mi lado en esta época bonita y dura a partes iguales, pues sólo el que se esfuerza sabe lo duro que es. En especial a mi familia, ya que sin ellos nada de esto habría sido posible.

Índice general

Lista de figuras

Lista de tablas

Resumen

Los fabricantes de sistemas de Circuito Cerrado de Television (CCTV) proporcionan a los desarrolladores un conjunto de utilidades (librerías, protocolos propietarios, etc.) para que puedan interactuar con los mismos. Sin embargo, no existe un estándar generalizado, ni tampoco una aplicación visor genérica, que permita interactuar con flujos de vídeo estándar, independientemente del fabricante.

Partiendo de esta problemática, este trabajo se plantea como principal objetivo, desarrollar un sistema en Java, formado por un componente, que permita construir un interfaz de visualización de vídeo. El componente se ha diseñado para poder visualizar el flujo de vídeo de cualquier fabricante, en un JPanel genérico de Java. Los formatos de vídeo que soporta son *H.264* y *MPEG*. Finalmente, para demostrar el funcionamiento del componente, se ha desarrollado una aplicación de ejemplo que permite manipular flujos de vídeo de cámaras de un fabricante concreto.

Palabras clave: sistema, vídeo, java, tiempo-real, CCTV

Abstract

Manufacturers of CCTV systems provide to the developers a set of tools (libraries, proprietary protocols, etc.) for interacting with the cameras and other video services. However, it does no exist a generic library to interact with any camera, regardless of the manufacturer.

The main objective of this project is to develop a Java component, which allows to build a generic viewer for IP video streams. It has been designed to display the video stream from any manufacturer, in a generic Java JPanel. This component supports the standard *H.264* and *MPEG* video formats. Finally, a sample application has been developed to demonstrate how to interact with the developed component.

Keywords: system, video, java, real-time, CCTV

Resumen Extendido

Hasta ahora, para poder visualizar flujos de vídeo desde cámaras de distintos fabricantes, se tenía que recurrir a sus propios visores, o a la realización de una aplicación mediante las librerías o protocolos propietarios que éstos facilitaban. Este paradigma se puede ver en la figura 1.

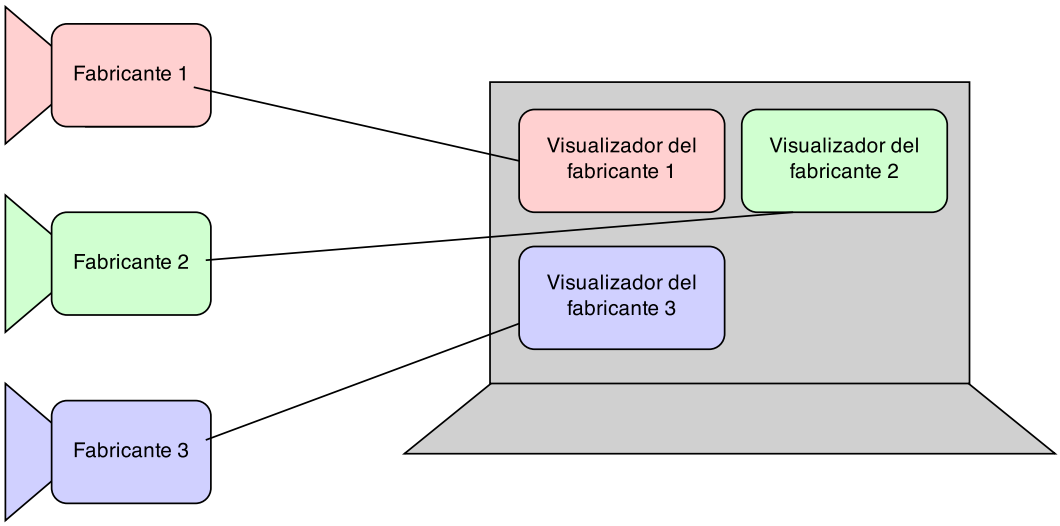

Figura 1: Visualizar flujos de vídeo

Eso resulta bastante tedioso y poco práctico, ya que se deben tener en cuenta todos los tipos de cámara a la hora de desarrollar una aplicación visor *genérica*. Además, al tener que usar librerías y protocolos propietarios de cada fabricante, tanto el mantenimiento como la portabilidad y distribución de la aplicación se vuelven inviables. Por ello, en este proyecto se ha construido un componente Java, denominado Jcctv, que, mediante el uso del protocolo RTSP, podrá conectarse a prácticamente cualquier cámara IP, ya que la mayoría de estas cámaras pueden hacer uso de ese protocolo para la retransmisión de flujos de vídeo en directo.

Se va a intentar conseguir lo que muestra la figura 2, es decir, que el componente, independientemente del fabricante, pueda visualizar flujos de vídeo RTSP. Como puede observarse, la diferencia con respecto a la figura 1 es clara, no necesitamos un software diferente para cada cámara.

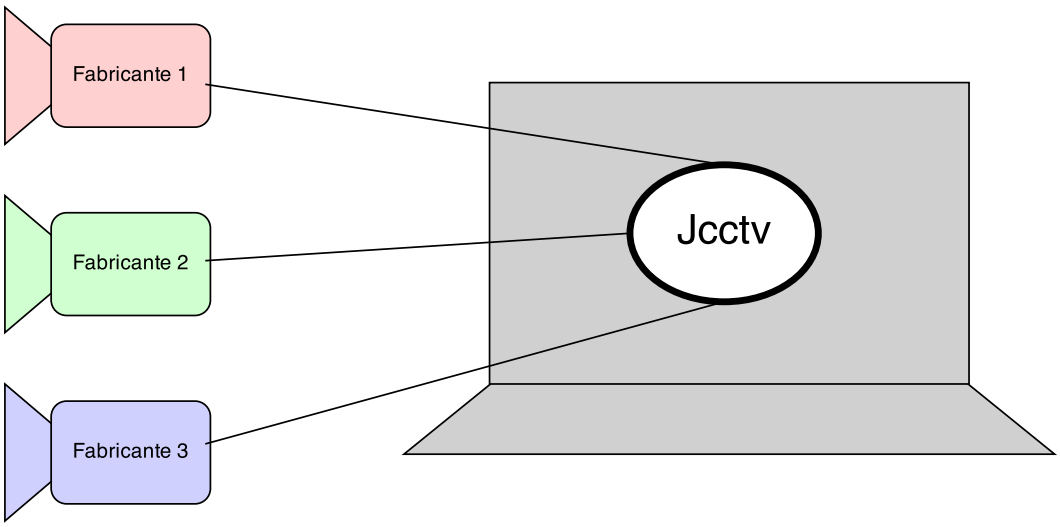

Figura 2: Comunicación de cámaras con Jcctv

Para que el componente sea lo más versátil posible, va a constar simplemente de un panel de visualización de vídeo. Las funcionalidades que soporta el componente son las siguientes: conectarse a una cámara para recibir vídeo de ésta y desconectarse de la misma. Finalmente, para evaluar el funcionamiento del componente, se ha desarrollado una aplicación contenedora con la que se puede configurar mínimamente el componente y se puede hacer un uso intuitivo del mismo. De esa forma se llegará al sistema representado en la figura 3, donde se tiene una aplicación que contiene al componente, el cuál puede dialogar con una cámara y mostrar vídeo de ésta. En la figura 4 se puede observar como queda la aplicación terminada mostrando flujo de vídeo de una cámara IP.

Cabe destacar que el componente ha sido desarrollado en Java. Para abordar el desarrollo se ha utilizado la librería vlcj [14], que cuenta con una serie de clases e interfaces que nos permiten utilizar los recursos de la librería VLC [12] mediante el lenguaje de programación Java. Esas clases e interfaces se explican en detalle a lo largo del trabajo en los capítulos 2 y 3. También se exponen fragmentos de código donde se muestran cómo y dónde deben ser utilizadas.

Debido a que el componente deberá ser usado en proyectos al igual que una librería o API de programación, se ha documentado utilizando la herramienta Javadoc.

Además el componente permite ser distribuido mediante la plataforma JWS [6], haciendo de su distribución e instalación un proceso muy sencillo e intuitivo.

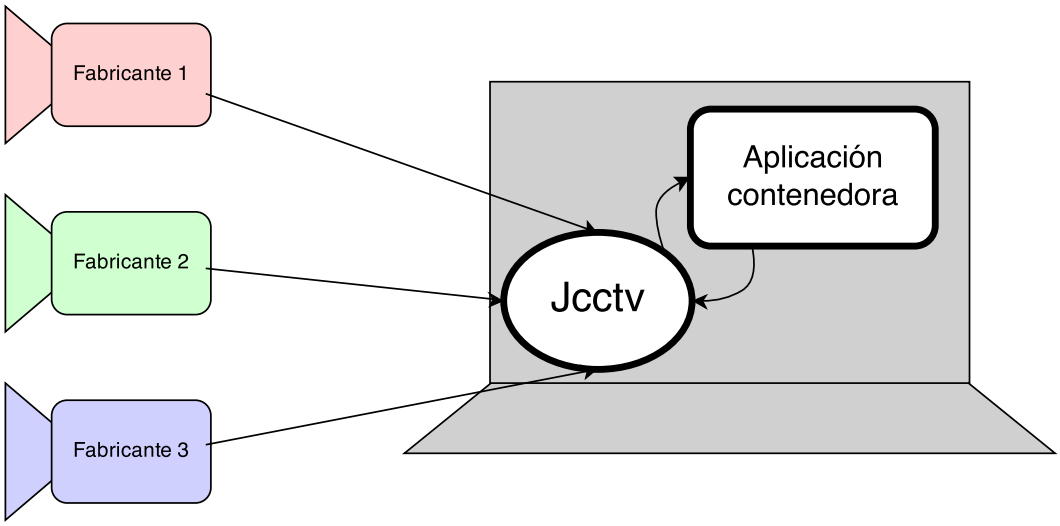

Figura 3: Visualizar flujos de vídeo

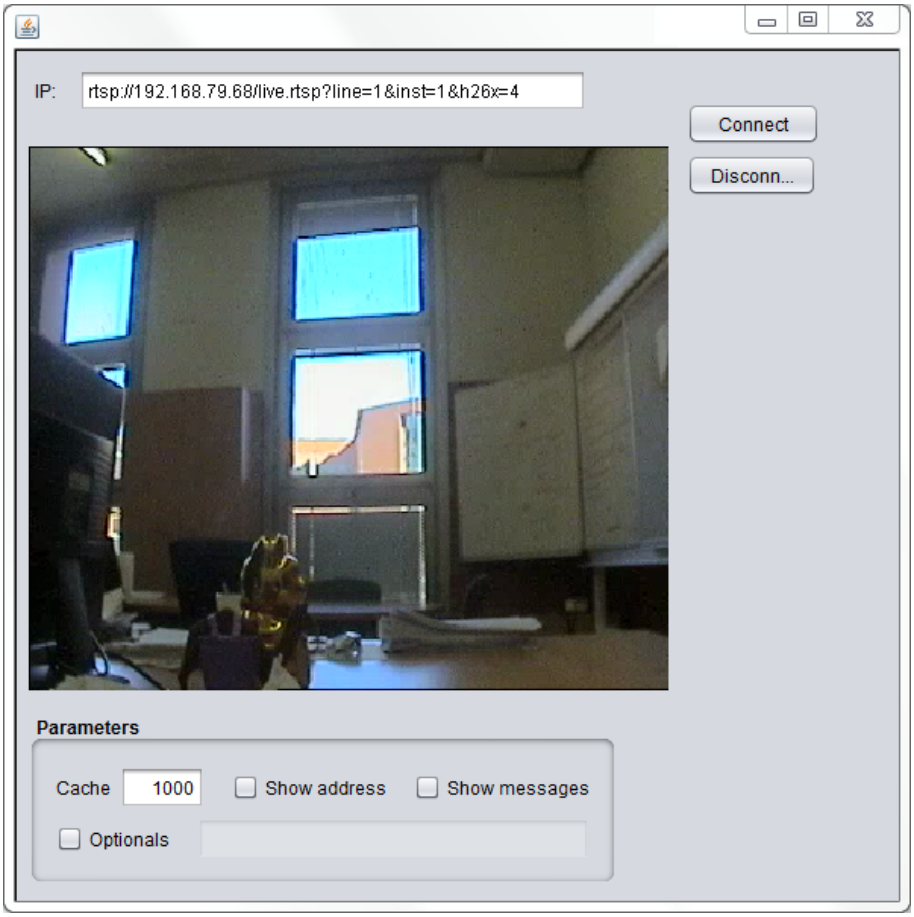

Figura 4: Aplicación contenedora mostrando flujo de vídeo de una cámara

Glosario

TFG Trabajo Fin de Grado

CCTV Circuito Cerrado de Television

RTSP Real Time Streaming Protocol

IETF Internet Engineering Task Force

RFC Request For Comments

GUI Graphical User Interface

RTP Real-time Transport Protocol

JWS Java Web Start

GPU Graphics Processing Unit

GPL General Public License

VOD Video on Demand

IPTV Internet Protocol Television

URL Uniform Resource Location

Capítulo 1

Introducción

En este capítulo se abordarán tres puntos clave de cualquier proyecto, como son la motivación para realizarlo, la descripción del mismo, y el uso que se puede hacer de éste en la vida real.

1.1. Motivación

Los fabricantes de sistemas de gestión de vídeo en circuito cerrado CCTV proporcionan a los desarrolladores un conjunto de utilidades (ya sea en forma de librerías o protocolos propietarios) para que éstos puedan interaccionar con los sistemas de cámaras. Esto condiciona los desarrollos ya que no existe un estándar generalizado que utilicen todos los fabricantes, así como tampoco una aplicación visor genérica que permita interaccionar con flujos de vídeo estándar, como por ejemplo RTSP. Así por ejemplo, si se quiere desarrollar una aplicación para visualizar flujos de vídeo de N fabricantes distintos, tendríamos que distribuir las N librerías correspondientes, dificultándose la distribución e instalación de la aplicación. Además, normalmente las librerías no se encuentran disponibles para los mismos lenguajes de programación, lo que vuelve inviable el desarrollo de una aplicación visor genérica. En las siguientes figuras se puede observar de forma gráfica la problemática descrita. En la figura 1.1, se puede observar como la cantidad de recursos que son necesarios para poder visualizar flujo de vídeo de cámaras de diferentes fabricantes es proporcional al número de estos, debiendo instalar software específico por cada uno.

Y la finalidad del componente es, que no sea necesario instalar software del fabricante de la cámara a la cuál se quiere conectar, como se muestra en la figura 1.2.

1.2. Definición

Partiendo de esta problemática, este trabajo se plantea como principal objetivo, desarrollar un componente Java que permita construir un *interfaz* de visualización de vídeo genérico de modo que los desarrollos no dependan de las librerías y entornos de

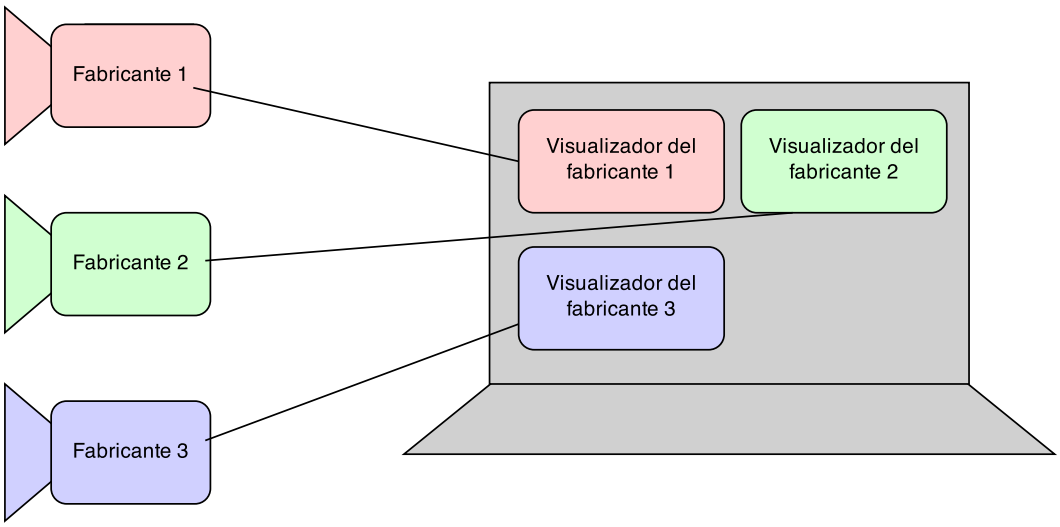

Figura 1.1: Visualizar flujos de vídeo

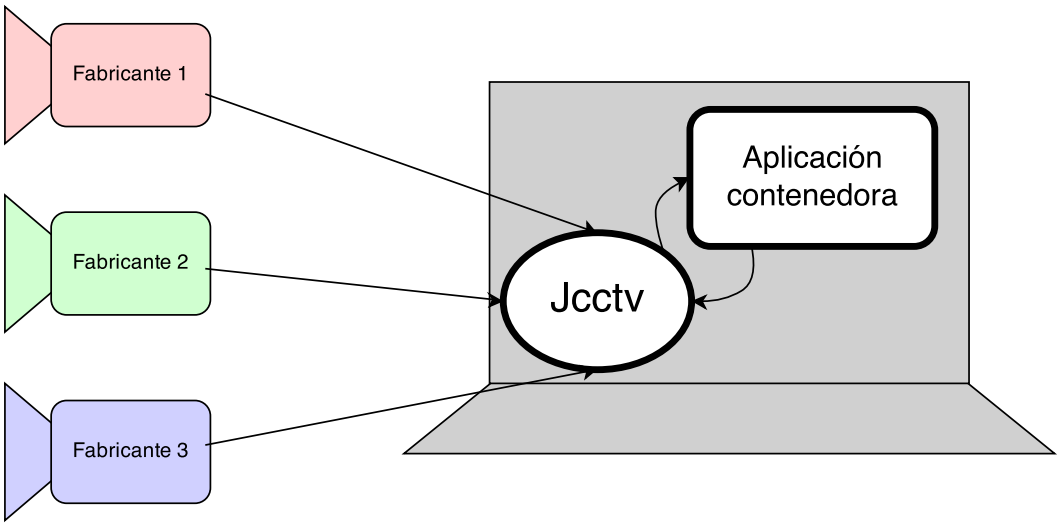

Figura 1.2: Visualizar flujos de vídeo

desarrollos proporcionados por los fabricantes de soluciones CCTV. En concreto, este componente, que ha sido denominado *Jcctv*, permite la conexión a flujos estándar de vídeo que siguen el protocolo RTSP definido por la Internet Engineering Task Force (IETF) en la Request For Comments (RFC) 2326 [10], y que es soportado por multitud de cámaras de CCTV de distintos fabricantes.

Este componente puede ser integrado en cualquier desarrollo de Graphical User Interface (GUI) escrito en Java, de modo que se pueda implementar un sistema de visualización de flujos de vídeo RTSP online con los formatos de codificación *H.264* y *MPEG*.

Además, el componente debe soportar el manejo de flujos Real-time Transport Protocol (RTP) *multicast*, de modo que el mismo cubra todas las posibilidades de interconexión con flujos de vídeo IP.

Por otro lado también se ha desarrollado una aplicación GUI que demuestre el funcionamiento del componente. Para esto, se hace uso de la librería Swing[7] de Java.

1.3. Campos de aplicación

El componente *Jcctv* es muy flexible, debido a que está desarrollado con la tecnología Java, y también es potente, puesto que utiliza protocolos estándar de retransmisión de vídeo. Además, el componente puede ser distribuido mediante la plataforma JWS, haciendo de su distribución e instalación un proceso muy sencillo e intuitivo. Gracias a su potencia y flexibilidad, con él se pueden crear aplicaciones complejas, como, por ejemplo:

- Aplicación de vídeo vigilancia. Teniendo acceso en la aplicación contenedora del componente a todas las cámaras que componen la seguridad de la entidad en cuestión, como una empresa o una vivienda particular. Esta aplicación podría verse como la figura 1.3.
- Aplicación de seguridad vial. Este componente se puede usar junto con un software de reconocimiento de infracciones en la red de carreteras.
- Retransmisión de eventos, pudiendo gestionar todas las cámaras en una misma aplicación, y aplicar cualquier tipo de procesamiento a los flujos de vídeo para una emisión de calidad.

1.4. Organización del documento

El resto del trabajo está estructurado de la siguiente forma. En el capítulo 2 se estudiará la librería utilizada en el proyecto vlcj, así como una breve introducción a la historia de VLC. En el capítulo 3 se profundizará sobre el desarrollo del componente, explicando su funcionamiento. En el capítulo 4 se puede ver cómo se ha hecho la aplicación contenedora, así como la forma de usarla y poder acceder al flujo de vídeo de una cámara desde la misma. En el capítulo 5 se expondrán las conclusiones del proyecto, así como futuras vías de trabajo que pueden quedar abiertas para mejorarlo. Este trabajo también cuenta con un apéndice en el que se muestra como documentar un proyecto con Javadoc.

Figura 1.3: Ejemplo de aplicación de videovigilancia donde podría utilizarse el componente Jcctv. Imagen tomada de [4]

Capítulo 2

Librería vlcj

Antes de poder estudiar la librería vlcj hay que saber que es una adaptación a Java del proyecto VLC. En este capítulo se describirá el proyecto VLC y a continuación se verán las características de la adaptación para el lenguaje Java, proporcionadas por la mencionada librería.

2.1. VLC media player

2.1.1. ¿Qué es VLC media player?

Como se puede ver en Wikipedia [13], VLC media player es un reproductor multimedia y *framework* multimedia libre y de código abierto desarrollado en el marco proyecto VideoLAN.

El punto fuerte de VLC es la cantidad de formatos y protocolos de streaming que puede reproducir. Además, es software libre, distribuido bajo la licencia General Public License (GPL) [8], con lo que se garantiza la libertad de compartirlo y modificarlo, así como usarlo en otros desarrollos (como el caso que nos ocupa).

VLC nació como un proyecto académico en 1996, desarrollado por estudiantes del École Centrale Paris, aunque desde 2009 es coordinado por la organización sin ánimo de lucro VideoLAN y mantenido por programadores de todo el mundo.

2.1.2. Versiones

A continuación se listarán las versiones de VLC media player, así como sus características más representativas de cada versión.

2.1.2.1. Rama 0.x

Fue liberado bajo licencia GPL el 1 de febrero del año 2001.

- *VLC 0.5*: Liberado en el año 2002, se empezó a experimentar con los "separadores"(*demuxers*) y códecs para poder transcodificar, aunque no estaban incluidas en

el programa.

- *VLC 0.7*: Capacidad para transmitir de varias fuentes en una sola instancia.
- *VLC 0.8*: Capacidad de transcodificar vídeos.

2.1.2.2. Rama 1.x

- *VLC 1.0.0.0*: Liberado el 7 de julio del año 2009. Soporta más formatos, puede reproducir ficheros multirar sin necesidad de descomprimirlos o unirlos. Además inicia el soporte para carpetas Blu-ray y AVCHD.
- *VLC 1.1*: Capacidad para aceleración por *hardware* de vídeo o Graphics Processing Unit (GPU) para los formatos H.264, VC-1 y MPEG-2.

2.1.2.3. Rama 2.x

- *VLC 2.0.1*: Liberado el 22 de marzo de 2012 con el nombre en clave *Twoflower*, mejorando la interfaz (incluyendo mayor compatibilidad para OS X), permitiendo la reproducción de discos Blu-ray e incluyendo más códecs.
- *VLC 2.1*: Soporte para resoluciones 4K o UltraHD además de un nuevo núcleo para reproducción de audio. Su nombre en clave es *Rincewind*.

2.1.3. Características principales

En este apartado se describen las características que han diferenciado al proyecto VLC con respecto a otros *framework* o reproductores multimedia.

Es un reproductor portable y multiplataforma con versiones para Microsoft Windows, GNU/Linux, Mac OS X, BeOS, BSD, eComStation, iOS y Android, entre otros.

VLC soporta muchos códecs de audio y vídeo, diferentes formatos de archivos como DVD o VCD y varios protocolos de streaming. También es capaz de transmitir datos en streaming a través de redes y convertir archivos multimedia a distintos formatos.

Muchos códecs de audio y vídeo están incluidos en VLC, utilizando la librería libavcodec del proyecto FFmpeg [3], aunque principalmente utiliza sus propios filtros de multiplexación. VLC incluye de forma nativa un gran número de códecs libres, evitando la necesidad de instalar o calibrar códecs propietarios.

VLC media player puede leer multitud de archivos dependiendo del sistema operativo. A continuación se puede observar los protocolos y medios extraíbles que soporta en la tabla 2.1, así como los formatos de contenedores en la tabla 2.2, de vídeo en la tabla 2.3, de audio en la tabla 2.4 y de subtítulos en la tabla 2.5.

2.2. vlcj

Una vez conocido el proyecto VLC ahora si se pueden introducir los conceptos de vlcj, qué es una librería que adapta el proyecto VLC al lenguaje de programación Java, permitiendo que pueda ser integrado en desarrollos con este lenguaje.

Como se puede ver en la página web del equipo encargado del proyecto, Caprica [1], vlcj es un proyecto de código abierto que ofrece una serie de interfaces en Java para acceder al *framework* VLC de VideoLAN. Mediante el uso de esas interfaces es posible construir un reproductor y servidor multimedia, usando sólo Java, para poder reproducir archivos de forma local e incluso a través de un servidor de vídeo bajo demanda Video on Demand (VOD) por *streaming*.

Hay muchas aplicaciones que están usando vlcj, proporcionando capacidades multimedia a los desarrollos en investigación oceanográfica y soluciones de Internet Protocol Television (IPTV) y de cine en casa a medida. También se está utilizando para crear software para la cámara de código abierto Elphel [2] y mapeo de vídeo para el proyecto Open Street Map [9].

2.2.1. Elementos de la librería utilizados durante el desarrollo

En esta sección se mostrarán las diferentes clases e interfaces de la librería vlcj que se han utilizado en este trabajo.

2.2.1.1. Interfaz RenderCallback

```
public interface RenderCallback
```

Esta interfaz pertenece al paquete uk.co.caprica.vlcj.player.direct.

Es una especificación para un componente al que se desea llamar para procesar los frames del vídeo. La devolución de llamada al render proporciona acceso al buffer de memoria nativa, que es la memoria solicitada por el sistema operativo que utiliza funciones como *malloc()* o *mmap()*.

Métodos

```
void display(DirectMediaPlayer mediaPlayer, com.sun.jna.
    Memory[] nativeBuffers, BufferFormat bufferFormat)
```

- La devolución de llamada es invocada cuando está preparado para mostrar un frame del vídeo.
- Parámetros:
  - mediaPlayer: objeto que reproduce el elemento multimedia.

- nativeBuffers: datos de vídeo de un frame.
- bufferFormat: información sobre el formato que usa el buffer.

2.2.1.2. Interfaz BufferFormatCallback

```
public interface BufferFormatCallback
```

Esta interfaz pertenece al paquete uk.co.caprica.vlcj.player.direct.

Permite que al cambiar el formato de vídeo se produzca una devolución de la llamada que es invocada por *DirectMediaPlayer*.

Métodos

```
BufferFormat getBufferFormat(int sourceWidth, int
    sourceHeight)
```

- Devuelve una instancia de *BufferFormat* especificando cómo debe estructurar sus buffers *DirectMediaPlayer*.
- Párametros:
  - sourceWidth: ancho del vídeo.
  - sourceHight: alto del vídeo.
- Devuelve:
  - el formato del buffer, no puede ser nulo.

2.2.1.3. Clase MediaPlayerFactory

```
public class MediaPlayerFactory extends Object
```

Pertenece al paquete uk.co.caprica.vlcj.player.MediaPlayerFactory.

Es una clase para instanciar reproductores de medios. Inicializa una instancia de libvlc y la usa para crear instancias de reproductores de medios. Se pueden crear múltiples instancias de clases, cada una con sus propias opciones de la librería libvlc si es necesario. Se debe liberar el objeto cuando la aplicación termina para poder limpiar los recursos nativos apropiadamente. La clase también proporciona acceso nativo a otros recursos de la librería libvlc como la lista de salidas de audio y la lista de filtros de audio y vídeo disponibles.

De entre todos los métodos que posee esta clase, estos son los que se utilizan en este trabajo:

Constructor

```
MediaPlayerFactory(String... libvlcArgs)
```

- Crea un nuevo objeto de reproducción de medios.
- Parámetros:
  - libvlcArgs: argumentos de la librería libvlc.

Métodos

```
DirectMediaPlayer newDirectMediaPlayer(BufferFormatCallback
    bufferFormatCallback, RenderCallback renderCallback)
```

- Crea un nuevo reproductor de medios para renderizar vídeo directo.
- Parámetros:
  - bufferFormatCallback: devolución de llamada para ajustar el formato del buffer deseado.
  - renderCallback: devolución de llamada para recibir los datos del frame del vídeo.
- Devuelve:
  - una instancia del reproductor de medios.

2.2.1.4. Interfaz DirectMediaPlayer

```
public interface DirectMediaPlayer extends MediaPlayer
```

Pertenece al paquete uk.co.caprica.vlcj.player.direct.

Es una especificación para un reprouctor de medios que proporciona acceso directo a los datos de los frames del vídeo. Un reproductor de medios directo es bastante útil en aplicaciones donde es necesario el acceso a los datos de los frames del vídeo para procesarlos en algún momento.

Métodos

```
DirectMediaPlayer newDirectMediaPlayer(BufferFormatCallback
    bufferFormatCallback, RenderCallback renderCallback)
```

- Crea un nuevo reproductor de medios para renderizar vídeo directo.

- Parámetros:
  - bufferFormatCallback: devolución de llamada para ajustar el formato del buffer deseado.
  - renderCallback: devolución de llamada para recibir los datos del frame del vídeo.
- Devuelve:
  - la instancia del reproductor de medios.

```
boolean playMedia(String mrl, String... mediaOptions)
```

- Reproduce un nuevo elemento multimedia con las opciones establecidas. El elemento comenzará la reproducción de forma asíncrona. Al ser así, algunas funciones podrían no trabajar correctamente si se invocan inmediatamente después, pues en algunas circunstancias habrá que esperar que algún evento se dispare para que esas funciones tengan efecto.
- Parámetros:
  - mrl: localizador del recurso multimedia.
  - mediaOptions: cero o más opciones para el elemento multimedia.
- Devuelve:
  - true si el elemento multimedia ha sido creado, false en caso contrario.

```
boolean isPlayable()
```

- ¿Se está reproduciendo el elemento multimedia actualmente?
- Devuelve:
  - true si el elemento se está reproduciendo, false en caso contrario.

```
boolean start()
```

- Inicia la reproducción y espera por el elemento multimedia a que comience a reproducirse o se produzca algún error. Esta llamada es bloqueante.
- Devuelve:

- true si el elemento multimedia comenzó a reproducirse o false si dió algún error.

```
libvlc_state_t getMediaPlayerState()
```

- Obtiene el estado actual del reproductor de medios.
- Devuelve:
  - el estado

```
libvlc_media_stats_t getMediaStatistics()
```

- Obtiene las estadísticas actuales del reproductor de medios. Las estadísticas sólo se actualizarán si se está reproduciendo vídeo.
- Devuelve:
  - las estadísticas multimedia.

```
void release()
```

- Destruye el reproductor de medios, liberando todos los recursos asociados (incluso los nativos).

A modo descriptivo, se muestra en la figura 2.1 como se relacionan todos los elementos de la librería vlcj descritos hasta ahora.

La figura de la izquierda muestra el proceso de conexión:

1. Se crea una instancia de la clase *MediaPlayerFactory* con las opciones de configuración de la librería libvlc.
2. Se crea una instancia de la clase *DirectMediaPlayer*, mediante el método *newDirectMediaPlayer()* de la clase *MediaPlayerFactory*. Este objeto será el reproductor de medios.
3. Mediante el método *playMedia()* se le dice a donde se tiene que conectar para recibir el flujo de vídeo.
4. Se comprueba que el reproductor puede reproducir el flujo de vídeo.
5. Se configura el formato del buffer mediante el evento *getBufferFormat()* de la interfaz *BufferFormatCallback*.

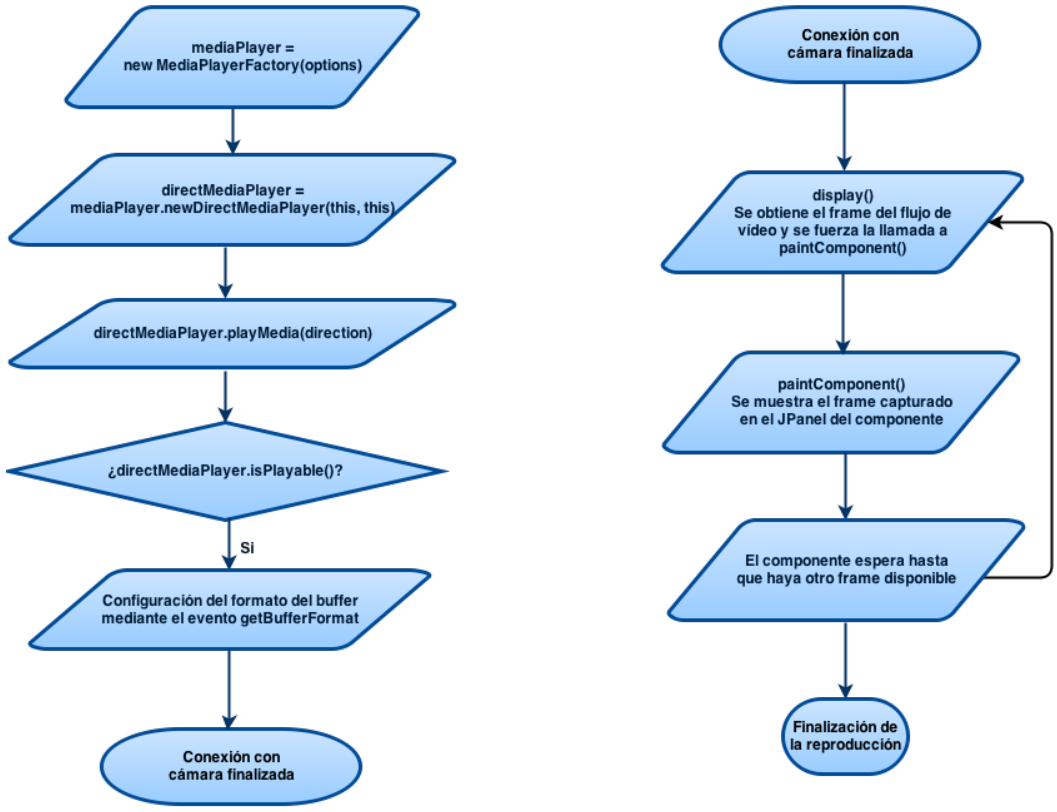

Figura 2.1: Descripción gráfica del comportamiento de la librería vlcj en el componente, a la izquierda en la conexión y a la derecha en la reproducción

Mientras que la figura de la derecha muestra el proceso de visualización del fujo de vídeo:

1. Cuando hay un frame disponible, automáticamente se llama al método *display()*, que obtiene ese frame y fuerza la llamada al método *paintComponent()*.

2. El método *paintComponent()* muestra el frame capturado pintándolo en el JPanel del componente.

3. Termina el proceso, cuando haya otro frame disponible, el método *display()* volverá a ejecutarse, volviendo a realizar el proceso mientras se mantenga la transmisión del flujo de vídeo.

Protocolo / Medio	Windows	Mac OS	Linux	BeOS	FreeBSD/ OpenBSD
UDP/RTP Unicast	Sí	Sí	Sí	Sí	Sí
UDP/RTP Multicast	Sí	Sí	Sí	No	Sí
HTTP / FTP	Sí	Sí	Sí	Sí	Sí
MMS	Sí	Sí	Sí	Sí	Sí
TCP/RTP Unicast	Sí	Sí	Sí	Sí	Sí
DVB (Satellite, Digital TV, Cable TV)	Sí part EyeTV 3	Sí	Sí	No	
Fuentes de vídeo	Sí Direct Show	Sí QTKit 5	Sí V4L, V4L2	No	No

Tabla 2.1: Protocolos y medios extraíbles

Formatos contenedores	Windows	Mac OS X	Linux	BeOS	FreeBSD/ OpenBSD
MPEG (ES,PS,TS,PVA, MP3)	Sí	Sí	Sí	Sí	Sí
AVI	Sí	Sí	Sí	Sí	Sí
ASF / WMV / WMA	Sí	Sí	Sí	Sí	Sí
MP4 / MOV / 3GP	Sí	Sí	Sí	Sí	Sí
OGG / OGM / Annodex	Sí	Sí	Sí	Sí	Sí
Matroska (MKV)	Sí	Sí	Sí	Sí	Sí
RealMedia	De forma parcial	De forma parcial	De forma parcial	De forma parcial	De forma parcial
WAV (including DTS)	Sí	Sí	Sí	Sí	Sí
Raw Audio: DTS, AAC, AC3/A52	Sí	Sí	Sí	Sí	Sí
Raw DV	Sí	Sí	Sí	Sí	Sí
FLAC	Sí	Sí	Sí	Sí	Sí
FLV (Flash)	Sí	Sí	Sí	Sin comprobar	Sí
MXF	Sí	Sí	Sí	Sin comprobar	Sí
Nut	Sí	Sí	Sí	Sin comprobar	Sí
Standard MIDI / SMF	Sí	Sí	Sí	Sí	Sí
Creative™ Voice	Sí	Sí	Sí	Sí	Sí

Tabla 2.2: Formatos contenedores

Formatos de vídeo	Windows	Mac OS X	Linux	BeOS	FreeBSD / OpenBSD
MPEG-1/2	Sí	Sí	Sí	Sí	Sí
DIVX (1/2/3)	Sí	Sí	Sí	Sí	Sí
MPEG-4 ASP, DivX 4/5/6, XviD, 3ivX D4	Sí	Sí	Sí	Sí	Sí
H.261	Sí	Sí	Sí	Sin comprobar	Sí
H.263 / H.263i	Sí	Sí	Sí	Sin comprobar	Sí
H.264/MPEG-4 AVC	Sí	Sí	Sí	Sí	Sí
Cinepak	Sí	Sí	Sí	Sí	Sí
Theora	Sí	Sí	Sí	Sí	Sí
Dirac/VC-2	Sí	Sí	Sí	Sin comprobar	Sí
MJPEG (A/B)	Sí	Sí	Sí	Sin comprobar	Sí
WMV 1/2	Sí	Sí	Sí	Sin comprobar	Sí
WMV 3 / WMV-9 / VC-1 1	Sí	Sí	Sí	Sí	Sí
Sorenson 1/3 (Quicktime)	Sí	Sí	Sí	Sí	Sí
DV (Digital Video)	Sí	Sí	Sí	Sí	Sí
On2 VP3/VP5/VP6	Sí	Sí	Sí	Sin comprobar	Sí
Indeo Video v3 (IV32)	Sí	Sí	De forma parcial	Sí	De forma parcial
Indeo Video 4/5 (IV41, IV51)	No	No	No	No	No
RealVideo 1/2	Sí	Sí	Sí	Sí	Sí
RealVideo 3/4	Sí	Sí	Sí	Sin comprobar	Sí

Tabla 2.3: Formatos de vídeo

Formatos de audio	Windows	Mac OS X	Linux	BeOS	FreeBSD / OpenBSD
MPEG Layer 1/2	Sí	Sí	Sí	Sí	Sí
MP3 - MPEG Layer 3	Sí	Sí	Sí	Sí	Sí
AAC - MPEG-4 part3	Sí	Sí	Sí	Sí	Sí
Vorbis	Sí	Sí	Sí	Sí	Sí
AC3 - A/52 (Dolby Digital)	Sí	Sí	Sí	Sí	Sí
E-AC-3 (Dolby Digital Plus)	Sí	Sí	Sí	Sí	Sí
MLP / TrueHD»3	Sí	Sí	Sí	Sí	Sí
DTS	Sí	Sí	Sí	Sí	Sí
WMA 1/2	Sí	Sí	Sí	Sí	Sí
WMA 3 1	Sí	Sí	Sí	No	No
FLAC	Sí	Sí	Sí	Sí	Sí
ALAC	Sí	Sí	Sí	Sí	Sí
Speex	Sí	Sí	Sí	Sin comprobar	Sí
Musepack / MPC	Sí	Sí	Sí	Sin comprobar	Sí
ATRAC 3	Sí	Sí	Sí	Sin comprobar	Sí
Wavpack	Sí	Sí	Sí	Sí	Sí
Mod (.s3m, .it, .mod)	Sí	Sí	Sí	Sin comprobar	Sí
TrueAudio (TTA)	Sí	Sí	Sí	Sí	Sí
APE (Monkey Audio)	Sí	Sí	Sí	Sí	Sí
Real Audio 2	De forma parcial	De forma parcial	De forma parcial	Sin comprobar	De forma parcial
Alaw/ulaw	Sí	Sí	Sí	Sí	Sí
AMR (3GPP)	Sí	Sí	Sí	Sí	Sí
MIDI 3	Sí	Sin comprobar	Sí	No	Sí
LPCM	Sí	Sí	Sí	Sí	Sí
ADPCM	Sí	Sí	Sí	Sí	Sí
QCELP	Sí	Sí	Sí	Sí	Sí
DV Audio	Sí	Sí	Sí	Sí	Sí
QDM2/QDMC (QuickTime)	Sí	Sí	Sí	Sin comprobar	Sí
MACE	Sí	Sí	Sí	Sí	Sí

Tabla 2.4: Formatos de audio

Formatos de subtítulos	Windows	Mac OS X	Linux	BeOS	FreeBSD / OpenBSD
DVD 1	De forma parcial	De forma parcial	De forma parcial	De forma parcial	De forma parcial
Text files (MicroDVD, SubRIP, SubViewer, SSA1-5, SAMI, VPlayer)	Sí	Sí	Sí	Sí	Sí
Closed captions	No	Sí	Sí	No	Sin comprobar
Vobsub	Sí	Sí	Sí	Sí	Sí
Universal Subtitle Format (USF)	Sí	Sí	Sí	Sí	Sí
SVCD / CVD	Sí	Sin comprobar	Sí	Sin comprobar	Sí
DVB	Sí	Sí	Sí	Sí	Sí
OGM	Sí	Sí	Sí	Sí	Sí
CMML	Sí	Sí	Sí	Sí	Sí
Kate	Sí	Sí	Sí	Sin comprobar	Sí
ID3 tags	Sí	Sí	Sí	Sí	Sí
APEv2	Sí	Sí	Sí	Sí	Sí
Vorbis comment	Sí	Sí	Sí	Sí	Sí

Tabla 2.5: Formatos de subtítulos y etiquetas

Capítulo 3

Implementación del componente Jcctv

Es este capítulo se detalla el desarrollo del componente Jcctv. Se explica todo el proceso, desde las decisiones iniciales relacionadas con el diseño del proyecto, hasta los problemas encontrados y soluciones aportadas.

Para hacer el texto más claro, el capítulo se dividirá en las siguientes partes:

- Desarrollo del componente. Aquí se puede ver como se ha diseñado y desarrollado el componente.
- Análisis de rendimiento. Aquí se realiza un estudio del rendimiento del componente bajo ciertas condiciones.
- Distribución del componente mediante JWS. Aquí se detallará el proceso de distribución del componente.

3.1. Desarrollo del componente

En esta primera sección se explicará cómo funciona el componente. Se detallará el proceso para conectarse a una cámara remota y obtener y mostrar flujo de vídeo así como la gestión de errores que proporciona.

3.1.1. Visualización del flujo de vídeo

El componente que se desarrolla en este proyecto está escrito en Java, y su función principal consiste en recuperar frames de un flujo de vídeo y visualizarlos en un JPanel.

Para poder visualizar esos frames en el componente se hará uso de la librería vlcj, como se describió en el capítulo 2.

Las fases por las que debe pasar el componente son principalmente dos, conexión con la cámara remota y recepción de vídeo, que se explican a continuación.

3.1.1.1. Conexión con cámara remota

Para poder iniciar una petición de conexión con una cámara remota, hay que configurar el componente con la siguiente información:

- Dirección IP: será la dirección IP de la cámara a la cuál se conectará el componente.
- Caché: memoria que reservará el componente a modo de buffer para intentar mantener la estabilidad en la visualización del vídeo.
- Mostrar dirección: esta es una opción de la librería vlcj, que permite visualizar encima del vídeo la dirección IP de la cámara a la que está conectado.
- Mostrar mensajes: otra opción de la librería que permite mostrar mensajes por consola del códec. Esta opción puede ser útil a la hora de desarrollar tanto el componente como la aplicación contenedora final.

Además de estos cuatro parámetros configurables, hay otro que permite establecer parámetros opcionales extra para configurar directamente el reproductor de la librería.

Una vez que el componente está configurado se puede proceder a realizar la conexión. La conexión con la cámara se realiza utilizando la librería vlcj configurada con los parámetros anteriormente descritos, de la siguiente forma.

1. Primero se realiza un ping para asegurar que existe conectividad con la cámara remota. Si no se tiene acceso, el proceso de conexión finaliza de forma no satisfactoria.
2. Una vez se ha comprobado que se tiene acceso a la cámara remota, se crea un objeto de la clase *MediaPlayerFactory* de la librería vlcj, que es una instancia al reproductor multimedia de la librería. Al crear este objeto debe configurarse. A continuación se muestran los métodos que permiten configurar el componente:

   Estos son los métodos *set* y *get* de las variables que controlan las opciones de configuración. Esas opciones son la caché, mostrar la dirección IP como marca de agua en el vídeo, mostrar mensajes de la librería libvlc por consola y parámetros opcionales extra.

```
private MediaPlayerFactory grabber_direct;
private String[] options = {"", "", ""};
public void set_cache(int cache) {

    String cadena;
    int valor;

    if (cache < 0) {
                    valor = 0;
```

```
    } else if (cache > MAX_CACHE) {
        valor = MAX_CACHE;
    } else {
        valor = cache;
    }
    cadena = "--network-caching=" + Integer.toString(valor);
    this.options[0] = cadena;
}
public void set_show_address(boolean flag) {

    if (!flag) {
        this.options[1] = "--no-video-title-show";
    } else {
        this.options[1] = "";
    }
}
public void set_show_messages(boolean flag) {

    if (!flag) {
        this.options[2] = "--quiet";
    } else {
        this.options[2] = "";
    }
}
public void set_extra_options(String extra_options) {
    more_options = extra_options;
}
public void connect(String dir) throws InterruptedException,
    IOException, Jcctv.Excepcion {

    // ...

    if (more_options == null || more_options.isEmpty()) {
    // Si no tiene opciones extra
        grabber_direct = new MediaPlayerFactory(options);
    } else {    // Si tiene opciones extra
        grabber_direct = new MediaPlayerFactory((options[0]
          + "," + options[1] + "," + options[2] + "," +
          more_options).replace(" ", "").split(","));
    }
```

```
    // ...
}
```

3. Con la instancia de la clase *MediaPlayerFactory* se puede obtener el reproductor con el que se tendrá acceso directo al flujo de vídeo. Mediante el método *newDirectMediaPlayer()* se obtendrá como resultado un objeto de la clase *DirectMediaPlayer*.

```
private void player(String dir) throws Jcctv.Excepcion,
   InterruptedException {

    // ...

    grabber_direct_player = grabber_direct.
       newDirectMediaPlayer(this, this);

                // ...
}
```

4. El siguiente paso es establecer la dirección de la cámara remota a la que se desea conectar mediante el método *playMedia()*.

```
private bool started;
private void player(String dir) throws Jcctv.Excepcion,
   InterruptedException {

    // ...

    started = grabber_direct_player.playMedia(dir);

    // ...
}
```

5. A partir de aquí se realizan una serie de comprobaciones para verificar que el flujo de vídeo se puede reproducir y que la conexión se ha realizado con éxito.

   Mediante el método *isPlayable()* se verifica que el flujo de vídeo puede ser reproducido.

   Si es así, se comprueba la estabilidad del flujo de vídeo. Esto se puede saber comprobando el estado, el número de bytes leídos para saber si el formato es el correcto y el número de imágenes mostradas, para comprobar si la caché es suficiente.

```
private bool playable;
private bool flag_status;
private void player(String dir) throws Jcctv.Excepcion,
   InterruptedException {

    // ...

    // Se verifica que el flujo es reproducible
    playable = grabber_direct_player.isPlayable();

    // Se inicia el volcado de informacion en el buffer.
    if (playable) {
        if (flag_status == false) {
            int media_state;
            int displayed_pictures;
            int bytes;

            grabber_direct_player.start();
            wait_end(time_to_connect);
            libvlc_state_t mediaState =
               grabber_direct_player.getMediaPlayerState();
            libvlc_media_stats_t mediaStatistics =
               grabber_direct_player.getMediaStatistics();

            media_state = mediaState.intValue();
            // Devuelve el estado de la ejecucion.

            displayed_pictures = mediaStatistics.
               i_displayed_pictures;
            // Devuelve el numero de frames guardados en el
               buffer.

            bytes = mediaStatistics.i_demux_read_bytes;
            // Devuelve el numero de bytes leidos.

            if (media_state == STATE_OK && bytes == 0) {
                is_connecting = false;
                release();
                throw new Jcctv.Excepcion("Formato
                   incorrecto. El formato solicitado no
```

```
                    coincide con el configurado en la camara
                    o no se puede reproducir ese formato.");
            }
            if (media_state == STATE_FAIL) {
                is_connecting = false;
                release();
                throw new Jcctv.Excepcion("Cadena incorrecta
                    o contrasena incorrecta.");
            }
            if (displayed_pictures > DISPLAYED_PICTURES_MIN
               && displayed_pictures <
               DISPLAYED_PICTURES_MAX) {
                is_connecting = false;
                release();
                throw new Jcctv.Excepcion("Buffer
                   insuficiente, ejecucion detenida. Aumente
                    el tamano del buffer del visor mediante
                   el metodo set_cache()");
            }

            // ...
        }
    }
}
```

6. Si la conexión se ha podido realizar con éxito, la librería llamará al método *getBufferFormat()* para configurar el formato del buffer donde se colocarán los frames capturados por la cámara remota.

```
@Override
public BufferFormat getBufferFormat(int sourceWidth, int
   sourceHeight) {

    if (change_video_format++ > 0) {
        notifyEvent(CHANGE_VIDEO_FORMAT);
    }
    BufferFormat format = new BufferFormat("RGBA",
       sourceWidth, sourceHeight, new int[]{sourceWidth *
       4}, new int[]{sourceHeight});

    return format;
```

```
}
```

El método *getBufferFormat()* tiene dos funcionalidades bien diferenciadas. Por un lado, mediante la variable *change_video_format* se comprueba si ya se ha llamado a este método anteriormente, así se puede saber si se llama al método porque el componente se acaba de conectar a un flujo de vídeo o porque una vez conectado a éste se cambia el formato. Por otro lado ajusta la dimensión del buffer en función del formato a reproducir, necesitando la codificación de color y la dimensión del frame (ancho y alto) para ello.

En la figura 3.1 se puede ver el proceso completo de conexión.

1. Se establecen las opciones de configuración.
2. Se realiza un ping a la cámara destino para comprobar su disponibilidad. Si no está disponible termina el proceso de forma no satisfactoria.
3. Se crea una instancia de la clase *MediaPlayerFactory* con las opciones de configuración de la librería libvlc previamente establecidas.
4. Se crea una instancia de la clase *DirectMediaPlayer*, mediante el método *newDirectMediaPlayer()* de la clase *MediaPlayerFactory*. Este objeto será el reproductor de medios.
5. Mediante el método *playMedia()* se le dice a donde se tiene que conectar para recibir el flujo de vídeo.
6. Se comprueba que el reproductor puede reproducir el flujo de vídeo.
7. Se configura el formato del buffer mediante el evento *getBufferFormat()* de la interfaz *BufferFormatCallback*.

3.1.1.2. Recepción de vídeo

Para poder mostrar el vídeo capturado por la cámara remota, se definen dos métodos proporcionados por la librería vlcj, a saber:

- *display()*
- *paintComponent()*

El primer método mencionado, *display()*, es el encargado de obtener los datos de vídeo de la cámara y almacenarlos en una variable local, ya preparados para ser mostrados, como se indicó en el capítulo 2. A continuación se puede ver el código del método *display()*.

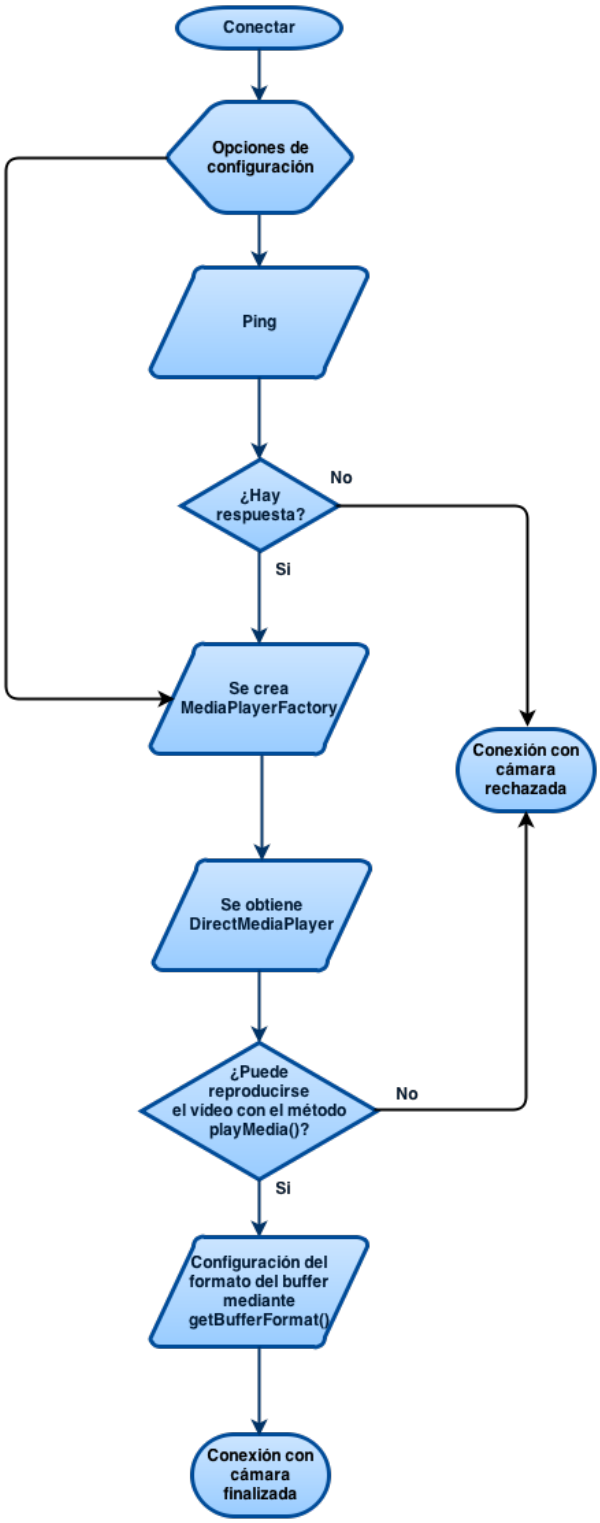

Figura 3.1: Comportamiento del componente en el proceso de conexión con una cámara remota

Lo primero que se puede ver en el método es una comprobación para saber si se ha cambiado el formato del vídeo mientras el componente lo está reproduciendo. Después se guarda el tiempo actual, esto será usado para comprobar si hay errores de conexión. Antes de proceder a copiar la imagen, se comprueba que la variable que se usará para tal fin, *img*, está iniciada. Si no es así se inicia, justando el tamaño en función del ancho y alto del frame del vídeo. A continuación se crea un buffer y se almacena en él la información del frame del vídeo, que después de unas operaciones, se copia en un array de píxeles perteneciente a la variable *img*, que es la que será mostrada en el método *paintComponent()*.

```
@Override
public void display(DirectMediaPlayer grabber_direct_player,
   Memory[] nativeBuffers, BufferFormat bufferFormat) {

    //Si el formato de video no ha cambiado se visualiza
    if (change_video_format == 1) {
        // Referencia temporal para controlar congelacion de la
           imagen o perdida de conexion.
        current_time_millis_display = System.currentTimeMillis()
           ;

        // Creacion de la imagen inicialmente vacia con el
           formato correcto del buffer de color
        if (img == null) {
            img = new BufferedImage((int) bufferFormat.getWidth
               (), (int) bufferFormat.getHeight(), BufferedImage
               .TYPE_INT_BGR);
        }

        // Crecion del buffer
        ByteBuffer buffer = nativeBuffers[0].getByteBuffer(0, (
           int) bufferFormat.getWidth() * (int) bufferFormat.
           getHeight() * 4);

        // Creacion del array de pixels que apunta directamente
           al array de datos de pixels de la imagen "img" creada
            antes.
        pixels = ((DataBufferInt) img.getRaster().getDataBuffer
           ()).getData();

        try {
```

```
                // Copia de la informacion del buffer al array de
                    pixels de cada elemento del buffer al de la
                    imagen a representar.
                buffer.asIntBuffer().get(pixels, 0, (int)
                    bufferFormat.getWidth() * (int) bufferFormat.
                    getHeight());
            } catch (Exception e) {
                notifyEvent(COPY_BUFFER_TO_PIXELS_ARRAY);
            }

            // Actualizacion del panel.
            repaint();
        }
    }
```

Al final del método se realiza una llamada a un método llamado *repaint()*. Este método es el que fuerza la llamada a *paintComponent()*, que muestra de forma visual el frame. El código de éste método es el que se muestra a continuación.

```
@Override
public void paintComponent(Graphics g) {

    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    // Visualizacion del la imagen correctamente actualizada
    g2.drawImage(img, 0, 0, this.getWidth(), this.getHeight(),
        null);
}
```

Lo que hace este método es pintar la imagen obtenida anteriormente por el método *display()* en el JPanel del componente.

Como se puede suponer este procedimiento se realizará por cada frame que capture de la cámara remota para poder mostrar una secuencia de vídeo.

En la figura 3.2 se puede ver gráficamente como se comporta el componente cuando se encuentra visualizando vídeo.

3.1.2. Control de errores

Durante la ejecución del componente pueden producirse ciertos errores o fallos que son necesarios controlar y notificar. Dependiendo de si estos fallos impiden que el componente pueda continuar funcionando o no, se van a tratar con notificaciones o excepciones, respectivamente. De este modo, si el error producido impide el correcto funcionamiento

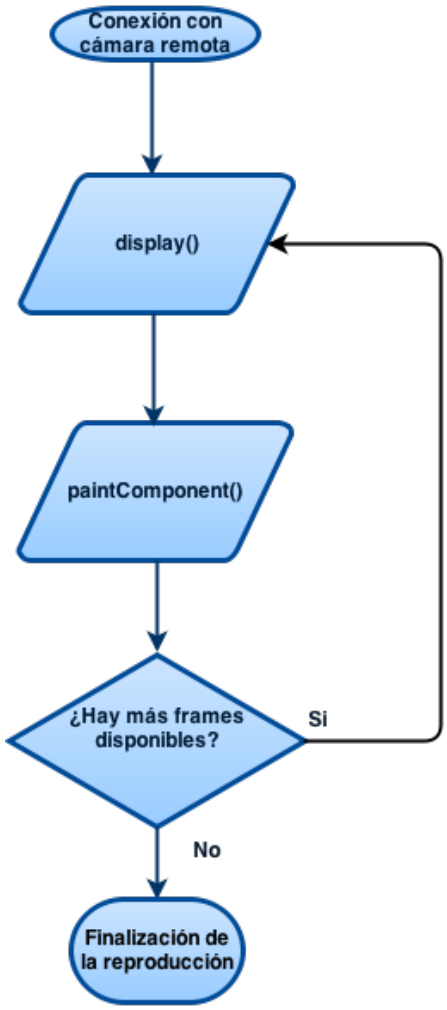

Figura 3.2: Comportamiento del componente al visualizar flujo de vídeo

del componente, éste será tratado como una excepción, interrumpiendo la ejecución del componente. En cambio si el error permite al componente seguir funcionando, el tratamiento que se aplicará será el de avisar a la aplicación contenedora y continuar la ejecución del componente.

3.1.2.1. Excepciones

En el caso de que al producirse un error, el componente no pueda continuar con la ejecución, se producirá una excepción. Para tratar este tipo de situaciones se ha creado una clase específica *Excepcion*, que hereda de *Exception*, y que será la encargada de notificar a la aplicación contenedora qué tipo de error se ha producido.

```
public class Excepcion extends Exception {

    private String msj;
```

```
    public Excepcion(String msj) {
        this.msj = msj;
    }

    @Override
    public String getMessage() {
        return msj;
    }
}
```

Estos son los tipos de excepciones que puede provocar el componente:

- Formato de vídeo incorrecto. Este error se produce cuando desde la aplicación contenedora se solicita a la cámara a la que se desea conectar, un formato de vídeo distinto al que la misma tiene configurado. Por ejemplo, si una cámara tiene dos flujos de vídeo posibles para conexiones, y el primero está configurado en formato *H.264* pero se le pide que se conecte al primer flujo con un formato *MPEG*.

```
if (media_state == STATE_OK && bytes == 0) {
    is_connecting = false;
    release();
    throw new Jcctv.Excepcion("Formato incorrecto. El
       formato solicitado no coincide con el configurado en
       la camara o no se puede reproducir ese formato.");
}
```

- Cadena o contraseña incorrecta. Si la cámara a la que se desea conectar necesita ser accedida mediante unas credenciales, es posible que las credenciales introducidas no sean las correctas, dando lugar a este error. El formato de una cadena de conexión que necesita credenciales es la siguiente: *rtsp://user:pass@ip_camara*.

```
if (media_state == STATE_FAIL) {
    is_connecting = false;
    release();
    throw new Jcctv.Excepcion("Cadena incorrecta o
       contrasena incorrecta.");
}
```

- Buffer insuficiente. Este error está directamente relacionado con el tamaño de caché con el que se ha configurado el componente. Si ese tamaño es demasiado pequeño,

la reproducción del vídeo no puede ser fluida, por lo que el componente responderá con este error e interrumpirá la conexión con la cámara.

```
if (displayed_pictures > DISPLAYED_PICTURES_MIN &&
   displayed_pictures < DISPLAYED_PICTURES_MAX) {
    is_connecting = false;
    release();
    throw new Jcctv.Excepcion("Buffer insuficiente,
       ejecucion detenida. Aumente el tamano del buffer del
       visor mediante el metodo set_cache()");
}
```

Por supuesto, y como se ha mencionado anteriormente, cualquiera de estos errores implica la notificación a la aplicación contenedora mediante una excepción, con lo que se detiene la ejecución del componente.

Puesto que estos tres errores dependen del estado del componente en el proceso de conexión, el componente sólo podrá lanzar excepciones en ese proceso.

3.1.2.2. Notificaciones

Este tipo de errores pueden darse durante la reproducción del flujo de vídeo de una cámara remota a la que ya se ha conectado el componente. El mecanismo empleado para el control de estos errores es la generación de eventos asíncronos. Estos permiten controlar aquellos errores que se producen una vez iniciada la visualización.

Para poder conseguir el comportamiento deseado se ha utilizado una interfaz, que se puede ver a continuación:

```
public interface EventListener {
    public void onNotificationEvent(Jcctv cn, int event);
};
```

Esta interfaz se implementa en la aplicación contenedora, y mediante el método *onNotificationEvent()*, definido a continuación, se pueden manejar los mensajes enviados desde el componente.

```
public void notifyEvent(int descEvent) {
    EventListener ea;
    Iterator<EventListener> it = listeners.iterator();

    while (it.hasNext()) {
        ea = it.next();
        ea.onNotificationEvent(this, descEvent);
```

```
        }
    }
```

Para poder enviar los mensajes desde el componente, se tiene una variable de tipo *LinkedList*, que almacena objetos que implementen la interfaz *EventListener*. La forma de pasarle al componente el objeto al que hay que notificar los mensajes es mediante el método *addListener()*.

```
public void addListener(EventListener ea) {
    listeners.add(ea);
}
```

De esta forma notificar un mensaje a la aplicación contenedora es tan fácil como llamar al método *notifyEvent()* con el código del mensaje a enviar como argumento.

A continuación se definen los diferentes errores de este tipo. El número de la lista representa el código de error:

1. Fallo en el buffer de vídeo. Esta notificación se envía a la aplicación contenedora cuando la copia de la información del buffer al array de píxeles falla.

2. Retraso excesivo en la visualización. Este error se produce cuando hay conexión con la cámara pero las imágenes tardan en llegar.

3. Pérdida de la conexión. Si se pierde la conexión con la cámara se produce este error y se notifica.

4. Error en el método encargado de la gestión de errores. Si el método encargado de la gestión de errores se ve interrumpido, se avisa a la aplicación contenedora mediante este error.

5. Cambio de formato de vídeo. Este error se produce cuando se detecta un cambio del formato de vídeo de la cámara mientras se está visualizando el flujo de vídeo.

   Este error merece una atención especial, puesto que durante el desarrollo del proyecto se ha descubierto que puede producir una finalización inesperada en la ejecución del componente debido a un fallo en la librería vlcj.

   Para solucionar este error, es necesario realizar una desconexión del componente en dos fases, de modo que la primera deje al componente bloqueado para prevenir un cierre inesperado de la aplicación y la segunda proceda a una desconexión normal del mismo.

   Esta solución deja una instancia ejecutándose en segundo plano, que es eliminada en cuanto el recolector de basura de Java se pone en funcionamiento.

6. Intento de conexión cuando el componente ya está conectado. Si se intenta una conexión y el componente está conectado a una cámara se notifica, para poder interrumpir este intento de conexión o desconectarse y conectarse a esta nueva cámara, por ejemplo.

3.1.2.3. Método *status()*

Se ha visto en la sección anterior que el componente es capaz de diferenciar entre retraso de visualización y pérdida de conexión, ¿cómo puede saber el componente si se está produciendo un retraso en la transmisión o directamente se ha perdido la conexión con la cámara remota? En cuanto se realiza una conexión con una cámara, el componente lanza un proceso en segundo plano encargado de la gestión de errores. Este método se llama *status()*, y su funcionamiento puede verse en la figura 3.3.

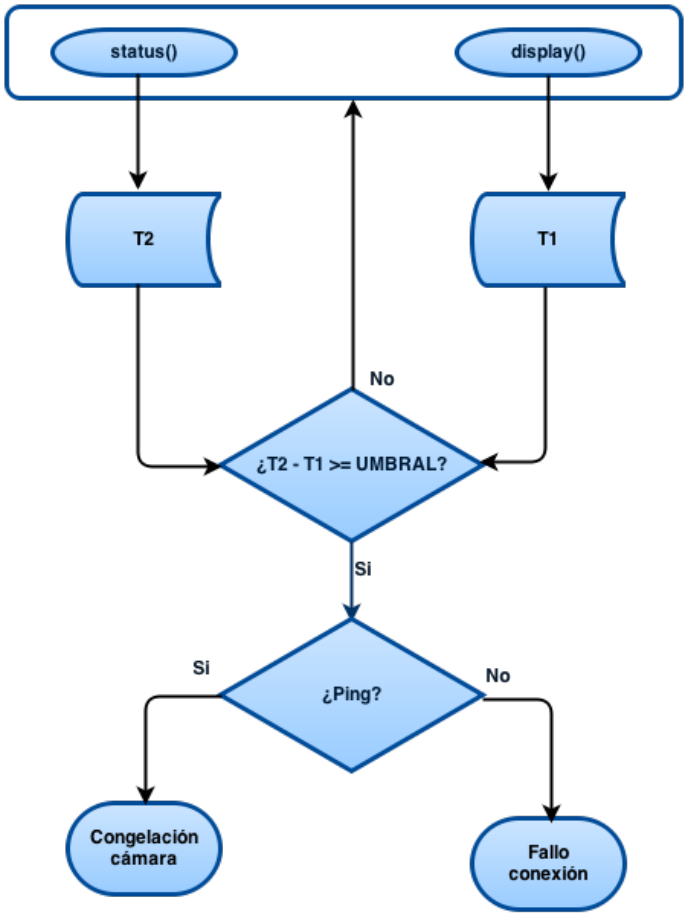

Figura 3.3: Funcionamiento del método status()

El método *status()* se ejecuta cada cierto tiempo, definido en la variable *time_limit*. Al ejecutarse guarda el valor del tiempo actual y lo compara con otro valor temporal, registrado por el método *display()* cuando éste se ejecuta. Si la diferencia entre ambas marcas temporales es mayor a un valor dado, UMBRAL, significa que el método *display()*

no se está ejecutando. Esto puede significar o que los frames del vídeo llegan con retraso, o que se perdió la conexión con la cámara. Para poder diferenciar entre estas dos situaciones se realiza un ping hacia la cámara, de modo que si responde, el problema es un retraso en el envío de frames, si no responde, significa que se ha perdido la conexión con la cámara.

3.2. Análisis de rendimiento

Se ha optimizado el componente reduciendo su consumo de memoria. Para ello, se han realizado algunas pruebas descritas a continuación, con una interfaz de cuatro paneles recibiendo vídeo, y se ha comprobado el consumo de memoria antes y después de cada prueba.

- Se ha realizado una simulación con cuatro paneles que contienen componentes Jcctv, apuntando el consumo de memoria en su inicio a las 13:22 con 71440 kb y finalizando a las 16:02 con 75856 kb. El consumo de memoria del componente ha sido de 4416kb.

  En la figura 3.4 se puede ver una captura del componente al inicio de la prueba, mientras que en la figura 3.5 se puede ver una captura una vez terminada la prueba de rendimiento mencionada anteriormente.

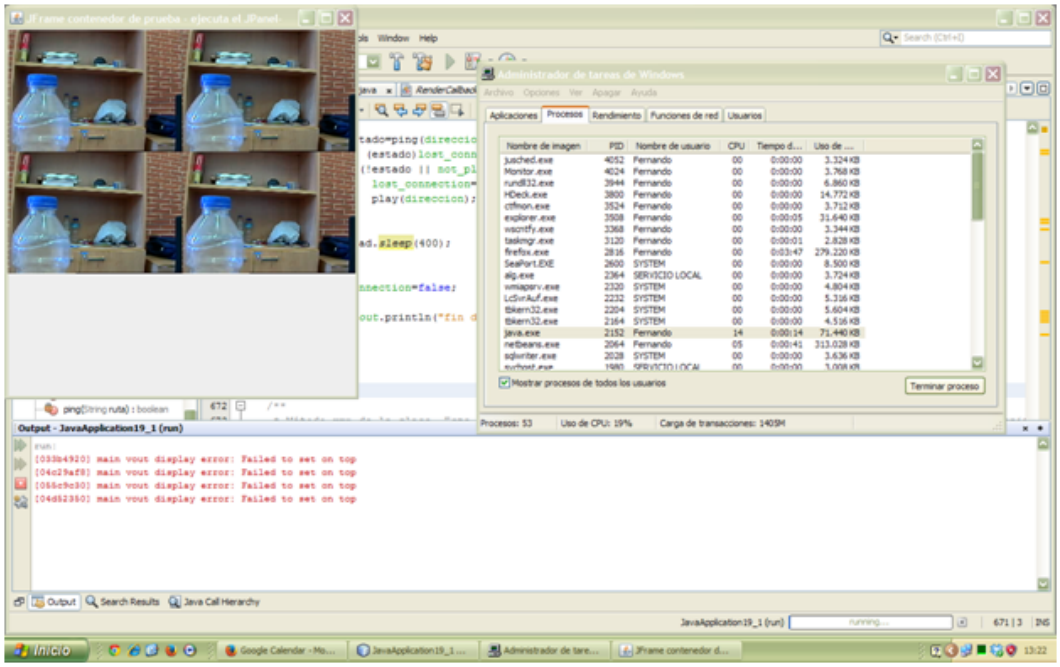

Figura 3.4: Análisis de rendimiento del componente Jcctv

- Otra prueba, con inicio a las 17:24 y fin al día siguiente a las 10:26 de la mañana. Como se puede apreciar en la figura 3.6, el consumo inicial es de 71456 Kb, mientras que el consumo final fue de 88904 kb, como se puede ver en la figura 3.7. El consumo total ha sido de 17448 kb.

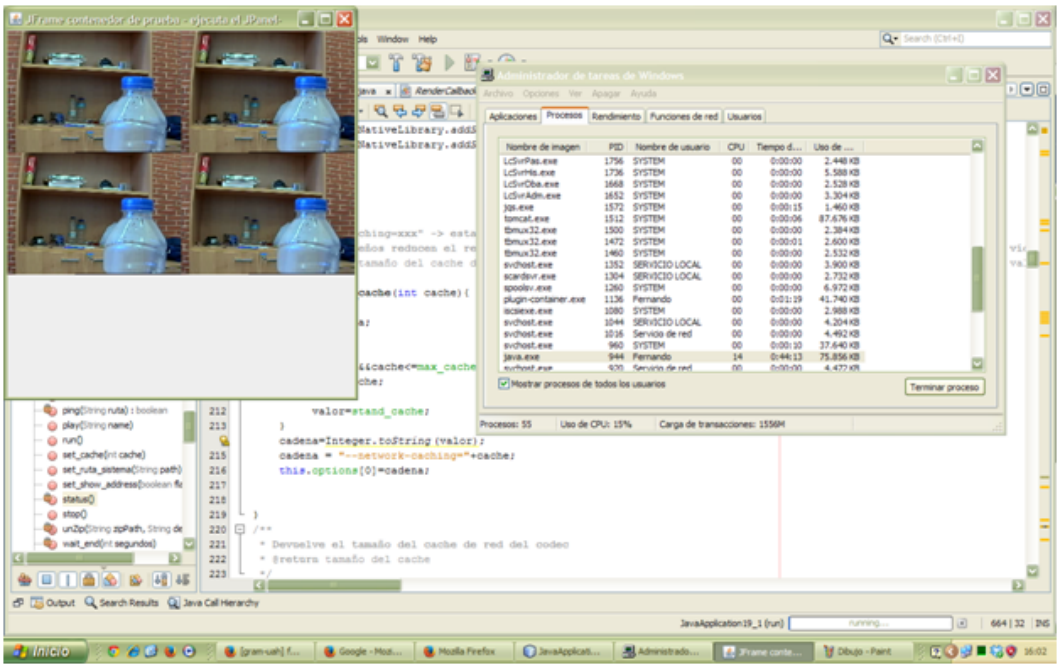

Figura 3.5: Análisis de rendimiento del componente Jcctv

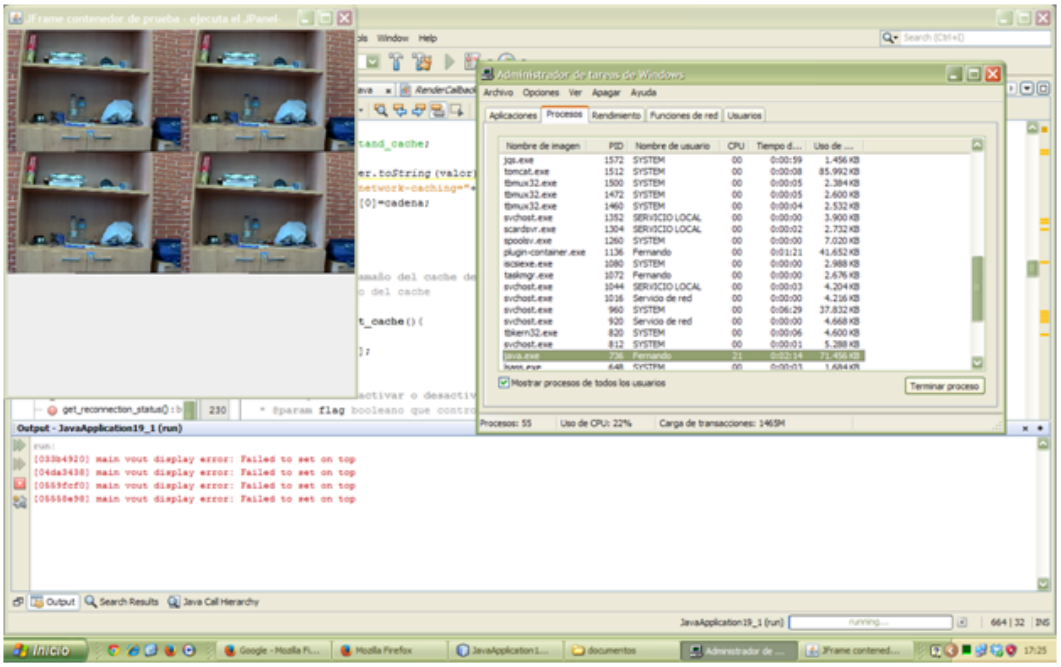

Figura 3.6: Análisis de rendimiento del componente Jcctv

- Se prueba también, como puede verse en la figura 3.8, el comportamiento del componente ante la pérdida de comunicación con la cámara, sin apreciarse un aumento en el uso de CPU durante la realización de los intentos de reconexión.

  Para conseguir esto, se añade una pausa en el bucle que controla la petición

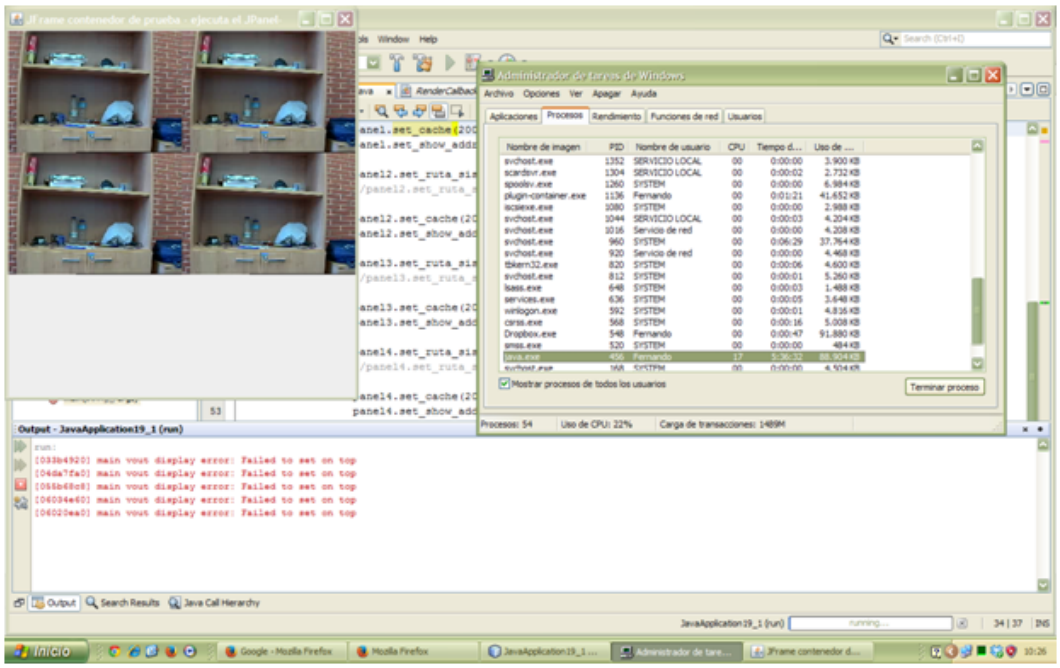

Figura 3.7: Análisis de rendimiento del componente Jcctv

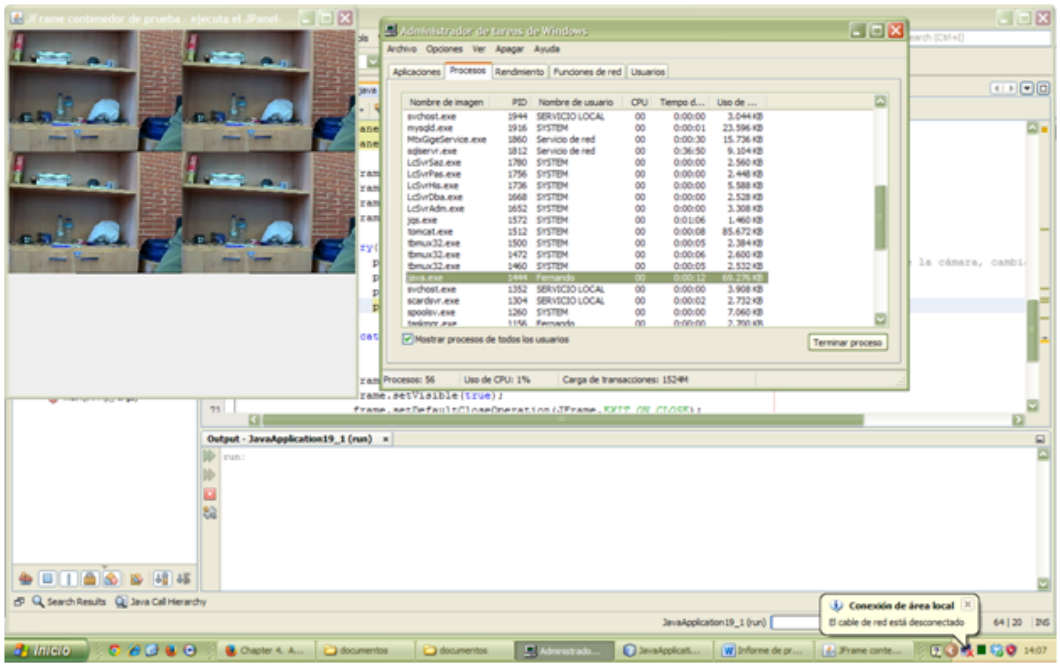

Figura 3.8: Análisis de rendimiento del componente Jcctv

de ping para evitar uso de CPU innecesario.

Como se puede ver en las distintas pruebas, se ha conseguido que el componente tenga un consumo de memoria bastante ajustado, aún en situaciones de pérdida de conexión y uso prolongado.

3.3. Distribución del componente mediante JWS

Para una distribución del componente más sencilla, se utiliza el sistema JWS de Java, pero, ¿en que consiste este sistema?

El software JWS, una vez se ha descargado por primera vez la aplicación, guarda una copia en caché, de modo que se ejecuta localmente, y cada vez que se ejecuta, comprueba si hay actualizaciones. De este modo se asegura que se esté ejecutando siempre la última actualización de la aplicación.

Para poder utilizar este sistema, se ha codificado un método, *getFileUrl()*. Este método primero obtiene la ruta del origen de la distribución realizada con JWS, desde el campo *codebase* de un archivo *.jnlp* de la aplicación. En esa ruta en el servidor se encontrará un archivo para descargar con todas las dependencias, librerías y ficheros de configuración necesarios para poder ejecutar la aplicación. Después se descarga el fichero, que está comprimido en *zip*, y lo extrae en la ruta marcada por la aplicación. A continuación se muestra el código que realiza lo descrito anteriormente.

```
private static void getFileUrl(String container, String
   system_path) throws UnavailableServiceException,
   MalformedURLException, IOException {

    final BasicService bs;
    final String codeBase;

    bs = (BasicService) ServiceManager.lookup("javax.jnlp.
       BasicService");
    codeBase = bs.getCodeBase().toString() + container;
    URL url = new URL(codeBase);
    // establecemos conexion
    URLConnection urlCon = url.openConnection();
    InputStream is = urlCon.getInputStream();
    FileOutputStream fos = new FileOutputStream(system_path +
       container);
    byte array[] = new byte[4096]; // buffer temporal de lectura
       .
    int leido = is.read(array);
    while (leido > 0) {//comienza la descarga
        fos.write(array, 0, leido);
        leido = is.read(array);
    }
    is.close();
    fos.close();
```

```
}
```

También se ha escrito un método, llamado *load_libs()*, que es el encargado de buscar los archivos necesarios para ejecutar la aplicación, y si no los encuentra llamar a *getFileUrl()* para proceder a su descarga y extracción.

```
public void load_libs() throws IOException,
   UnavailableServiceException {

    File file = new File(system_path);

    if (file.exists() == false) {//comprueba si existe la
       carpeta y la crea solo si es necesario.
        file.mkdirs();
        getFileUrl(container, system_path);
        unZip(system_path + container, system_path);
    }
}
```

En caso de que la aplicación tuviera que descargarse esos archivos, se descargan comprimidos. Para poder descomprimirlos se utiliza el método *unZip*, descrito a continuación.

```
private static void unZip(String zipPath, String destinationDir)
    throws IOException {

    InputStream is;
    FileOutputStream fos;
    ZipInputStream zis;

    // Crea el directorio de destino
    File file = new File(destinationDir);
    if (file.exists() == false) {
        file.mkdirs();
    }
    is = new FileInputStream(zipPath);
    zis = new ZipInputStream(is);

    ZipEntry entry;
    byte data[] = new byte[4096];
    int count;
    while ((entry = zis.getNextEntry()) != null) {//se recorre
```

```
        la estructura del archivo comprimido.
         String fileName = destinationDir + File.separator +
            entry.getName();
         //creacion de directorios
         if (entry.isDirectory()) {
             file = new File(fileName);
             if (file.isDirectory() == false) {
                 file.mkdirs();
             }
             //creacion de los archivos contenidos en los
                directorios
         } else {
             fos = new FileOutputStream(fileName);
             while ((count = zis.read(data)) != -1) {
                 fos.write(data, 0, count);
             }
             zis.closeEntry();
             fos.flush();
             fos.close();
         }
     }
     zis.close();
     is.close();
     zis.close();
}
```

Y con esto se tiene una aplicación, en este caso el componente Jcctv, listo para distribuirse e instalarse de una forma rápida y sencilla para el usuario.

Capítulo 4

Aplicación contenedora

La principal función de la aplicación contenedora es validar la funcionalidad del componente Jcctv de una forma gráfica y lo más simple posible.

Teniendo esta idea en mente, en la figura 4.1 se puede ver un boceto de los elementos que debe tener esta aplicación para cumplir con su funcionalidad.

Figura 4.1: Interfaz de la aplicación contenedora

4.1. Desarrollo de la aplicación contenedora

Al ejecutarse, la aplicación contenedora crea una nueva instancia del componente Jcctv y se establece a sí misma como *EventListener* mediante el método *addListener()* del componente. Esto lo consigue de la siguiente manera:

```
private Jcctv video;
public ContainerApp() {
    // ...
    initJcctv();
}
private void initJcctv() {
    video = new Jcctv();
    video.addListener(this);
    video.set_system_path(SYSTEM_PATH);
    video.set_cache(1000);
    video.set_show_address(false);
    video.set_show_messages(false);
}
```

Además, el método *initJcctv()* proporciona unos valores de configuración por defecto.

De esta forma permitimos al componente que envíe mensajes a la aplicación contenedora y ésta pueda responder automáticamente a determinadas situaciones, cómo por ejemplo una transmisión lenta del vídeo de una cámara o la desconexión con la misma. También se establecen algunos parámetros de ajuste del componente con valores por defecto, para que sea más sencillo usar la aplicación.

Una vez establecida la aplicación contenedora como receptora de notificaciones del componente, se va a estudiar qué acciones realiza la misma a la hora de proceder a una conexión con una cámara.

```
private String direccion;
private void connect() throws InterruptedException, IOException,
    Excepcion {
    direccion = tf_ip.getText();
    int cache;
    try {
        cache = Integer.parseInt(tf_cache.getText());
    } catch (Exception ex) {
        changeCache(ex.getMessage());
        return;
    }
    if (cache < 0 || cache > 2000) {
```

```
            changeCache("");
            return;
        }
        boolean address = cb_address.isSelected();
        boolean messages = cb_messages.isSelected();
        video.set_cache(cache);
        video.set_show_address(address);
        video.set_show_messages(messages);
        if (cb_optionals.isSelected()) {
            video.set_extra_options(tf_param.getText());
        }
        video.connect(direccion);
        video.setSize(jcctv_panel.getWidth() - 2, jcctv_panel.
           getHeight() - 2);
        video.setBounds(1, 1, jcctv_panel.getWidth() - 2,
           jcctv_panel.getHeight() - 2);
        jcctv_panel.add(video);
        jcctv_panel.setVisible(true);
    }
```

Para poder explicar este código hay que mostrar la interfaz de la aplicación contenedora, para poder establecer la relación entre los elementos de la interfaz donde se introducen los datos necesarios para configurar el componente y establecer una conexión con la cámara.

- Como se puede ver en el figura 4.2, hay un cuadro de texto para introducir la dirección IP, con valor *192.168.79.68*. En la primera línea del método *connect()* lo que se hace es obtener la dirección IP de ese cuadro de texto y almacenarla en la variable *direccion*.

- En las siguientes líneas se está comprobando si la caché introducida se encuentra dentro de los límites marcados por el componente. Por defecto y como se puede ver en la figura 4.2, este valor es de 1000, puesto que en las pruebas en el laboratorio es el valor que mejor resultado ha dado.

- Seguidamente se obtienen los valores de dos *CheckBox*, *Show address* y *Show messages*.

- Después se configura el componente con los valores de caché, mostrar dirección y mostrar mensajes, leídos anteriormente.

- A continuación se comprueba si hay parámetros opcionales, mediante el *CheckBox Optionals*, en cuyo caso pasan a formar parte de las opciones del componente.

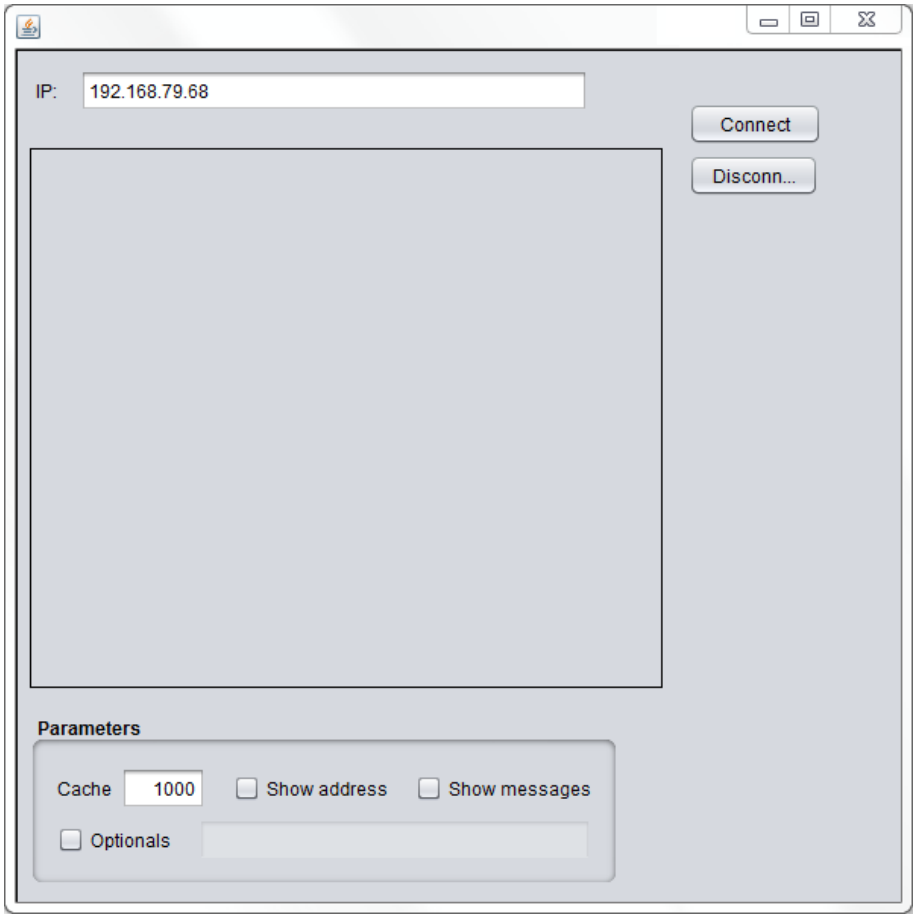

Figura 4.2: Aplicación contenedora

- Llegados a este punto se intenta establecer conexión con la cámara remota mediante el método *connect()* del componente.
- Si el componente pudo conectarse a la cámara, ya sólo queda establecer el tamaño del panel donde se visualizarán las imágenes, añadir el componente Jcctv a la lista de componentes de ese panel y hacerlo visible.

Para dejar de recibir el flujo de vídeo de una cámara basta con pinchar en el botón *Disconnect* de la aplicación.

4.2. Gestión de errores

A continuación vamos a estudiar cómo se comporta la aplicación contenedora ante las situaciones no deseadas que pueden provocar que el componente le envíe un mensaje. Cómo se explicó en el capítulo anterior, el componente enviará un mensaje a la aplicación contenedora en los siguientes casos:

1. Fallo en el buffer de vídeo.

2. Retraso excesivo en la visualización.
3. Pérdida de la conexión.
4. Error en el método encargado de la gestión de errores.
5. Cambio de formato de vídeo.
6. Intento de conexión cuando el componente ya está conectado.

En esta aplicación no se han codificado respuestas para todos los eventos, y para otros se ha tenido que forzar la desconexión del componente debido a errores en las librerías utilizadas. Teniendo en cuenta eso, y que la aplicación está diseñada para intentar mantener el flujo de vídeo activo el mayor tiempo posible, la respuesta a los eventos generados por el componente son tratados de la siguiente forma:

- Si se produce un retraso excesivo en la visualización, la aplicación intentará una desconexión del componente para después aumentar el caché y volver a conectar.
- Si se pierde la conexión con la cámara, la aplicación intentará una reconexión, desconectando el componente de la cámara e intentando una nueva conexión.
- Si se intenta una conexión cuando el componente ya está conectado, se supondrá que se intenta conectar a una cámara diferente, por lo que se procederá a una desconexión de la cámara actual y se procederá a realizar conexión con la nueva cámara.
- Si se produce un cambio de formato en el vídeo que se está recibiendo se realizará una desconexión forzosa. Esto es debido a un error en las librerías utilizadas en el desarrollo que ya se explicó en el capítulo anterior.
- Si se produce un error en el método encargado de la gestión de errores la aplicación no hará nada.
- Si se produce un fallo en el buffer de vídeo al copiar los píxeles la aplicación no hará nada.

Se tomó la decisión de no tratar algunos eventos debido a que la aplicación contenedora se ha codificado básicamente para poder probar el correcto funcionamiento del componente, y durante su desarrollo estos eventos nunca se dieron.

4.3. Manual de usuario

4.3.1. Elementos de la interfaz

En la figura 4.3 se puede ver la forma que tiene la aplicación contenedora una vez que su desarrollo ha llegado a su fin.

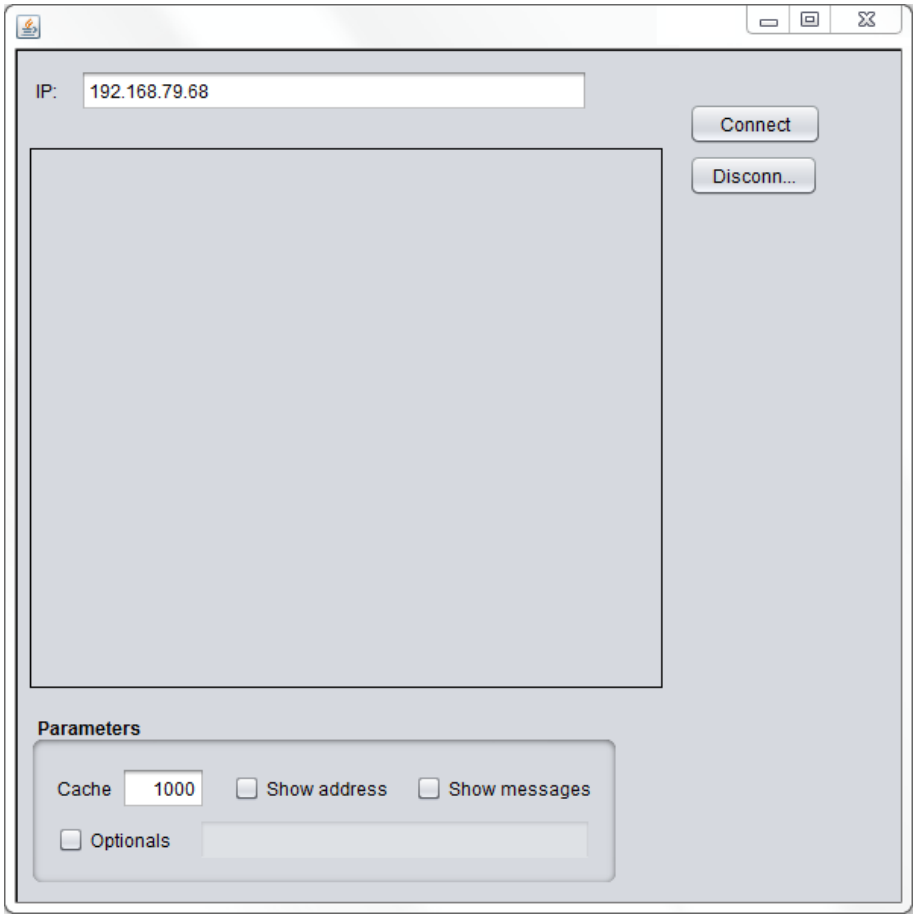

Figura 4.3: Aplicación contenedora

En este capítulo se expondrá al lector una guía para el uso de la aplicación, explicando cada elemento de ésta y los pasos que han de seguirse para poder usarla.

Primero se definirá cada elemento de la interfaz. Para ello se comenzará con la caja de texto para introducir la dirección IP de la cámara a la cuál se desea conectar, que se puede ver en la figura 4.4.

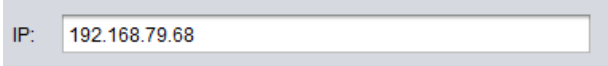

Figura 4.4: Cuadro para la introducción de dirección IP

Este cuadro sirve para introducir la dirección IP de la cámara a la que se desea conectar. En la figura 4.4 sólo aparece una dirección IP, pero se pueden enviar ciertos parámetros como el flujo específico que queremos visualizar o el formato de vídeo. Estos parámetros vienen definidos por el protocolo RTSP y limitados por la cámara.

La figura 4.5 representa los parámetros con los que se puede configurar el componente.

Como se puede ver, se puede establecer el valor de la caché, si se quiere ver la

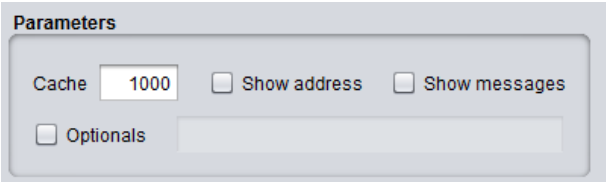

Figura 4.5: Parámetros de configuración del componente

dirección IP como marca de agua, o si se quiere que el componente muestre por consola mensajes relacionados con la ejecución del mismo, ésto último es útil para desarrolladores. También aparece un cuadro de texto deshabilitado por defecto, en el cuál se pueden añadir más opciones al componente definidos por la librería vlcj. Estos parámetros auxiliares no son necesarios y se recomienda no usarlos a no ser que se sepa con seguridad lo que se está haciendo ya que puede provocar inestabilidad y que la ejecución del componente, y por tanto de la aplicación, falle.

A parte de los elementos ya definidos, la caja de texto para introducir la dirección IP y el cuadro de parámetros para configurar el componente, sólo restan por mencionar los botones de conexión y desconexión y el cuadro de visualización, todos ellos fácilmente identificables en la figura 4.2.

4.3.2. Cómo conectarse a una cámara

En este apartado se hará una simple demostración de cómo conectarse a una cámara. Para ellos, los pasos a seguir son los siguientes:

1. Se introduce la dirección IP de la cámara de la que queremos visualizar flujo de vídeo. Al final de este capítulo se explicará la estructura de la URL de conexión RTSP.

2. Se configura el componente con los parámetros establecidos, en este caso se ha establecido una caché de 1000, porque como ya se mencionó anteriormente en este capítulo es el valor que mejores resultados ha dado, y no se desea que se muestre la dirección IP como marca de agua, ni que el componente muestre mensajes por consola. Tampoco se ha introducido ningún parámetro auxiliar.

3. Ahora se pincha sobre el botón de conexión, comenzando así con el proceso de conexión del componente hacia la cámara.

Llegados a este punto, y como puede verse en la figura 4.6, la aplicación ya se encuentra visualizando el flujo de vídeo de la cámara remota.

Para detener la visualización sólo hay que pulsar sobre el botón de desconexión, de modo que se limpie el cuadro de visualización y la aplicación vuelva al estado inicial.

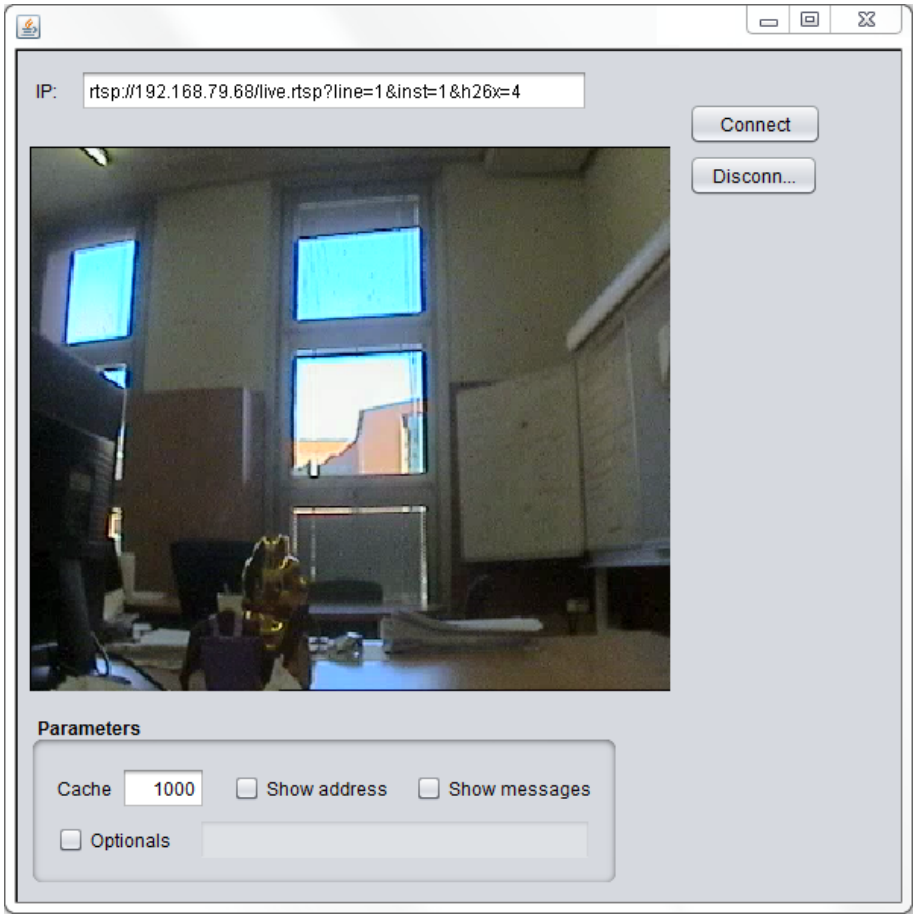

Figura 4.6: Aplicación contenedora mostrando flujo de vídeo de una cámara

4.3.3. Formato de una cadena URL RTSP

Como en el caso de este trabajo se ha utilizado una cámara de Bosch, se explicarán los diferentes campos de la cadena URL para la misma.

La cadena está formada por los siguientes campos:

- *rtsp://* es el protocolo, que se va a utilizar.
- *192.168.79.68/* es la dirección IP de la cámara.
- *live.rtsp* es el recurso de la cámara que transmite el vídeo en directo. Se pueden añadir argumentos, para ello hay que añadir el carácter *?* al final.
- Argumentos:
  - Flujo: se elije a cuál de los dos flujos se quiere conectar, para conectar al flujo 1, *inst=1*, y para acceder al flujo 2, *inst=2*.
  - Cámara: si se tiene una unidad multicanal, la forma de elegir la cámara a la que se quiere conectar es mediante el argumento *line*.

- Códec: mediante el argumento *h26x* se pueden elegir los siguientes formatos:
  - *0. JPEG*
  - *3. MPEG-4 SH++*
  - *4. H.264*

Capítulo 5

Conclusiones y futuras líneas de trabajo

5.1. Conclusiones

Con la realización de este Trabajo Fin de Grado (TFG) se han conseguido los siguientes objetivos:

1. Desarrollar un componente potente y flexible, capaz de conectarse a flujos de vídeo de cámaras IP usando protocolos estándar.

2. Hacer de su distribución e instalación un proceso muy sencillo e intuitivo gracias a la plataforma JWS.

3. Desarrollar una aplicación contenedora con la que demostrar el comportamiento del componente y tener una aplicación plenamente funcional capaz de conectarse a una cámara IP y mostrar el flujo de vídeo.

Que son los objetivos marcados al inicio de este TFG.

5.2. Futuras líneas de trabajo

Hay dos vías principales que se pueden seguir para mejorar este TFG, una de ellas es centrarse en el componente y aumentar su funcionalidad, mientras que la otra es usar el componente para desarrollar una aplicación con unos objetivos concretos.

5.2.1. Mejoras del componente

- Se puede ampliar la funcionalidad del componente añadiendo protocolos propietarios para determinadas cámaras de determinados fabricantes que puedan ofrecer así funcionalidades exclusivas.

- Junto con otras aplicaciones y arquitecturas de los fabricantes, se puede completar el componente de forma que pueda acceder a un flujo de vídeo grabado previamente y visualizarlo.

5.2.2. Creación de aplicaciones basadas en el componente

Como se mencionó en el capítulo de introducción, se puede utilizar el componente Jcctv para desarrollar aplicaciones concretas, como por ejemplo:

- Aplicación de vídeo vigilancia.
- Aplicación de seguridad vial.
- Aplicación de retransmisión de eventos.

Apéndice A

Javadoc

A.1. ¿Qué es Javadoc?

Javadoc es una herramienta de Sun para documentar el código de una aplicación mediante los comentarios de la misma. Esta herramienta localiza estos comentarios y los extrae de forma que sean útiles para el usuario de esa aplicación.

Utilizando ciertas reglas en los comentarios de la aplicación se crea un fichero HTML con la documentación correspondiente aportada por el autor del código.

El formato utilizado para que el entorno de desarrollo, en este caso NetBeans, sepa que el comentario que se acaba de escribir debe ser incluido en la documentación, es el siguiente:

```
/**
*
* @author Autor
* @version 1.0
*
*/
```

Como puede observarse en el código, el comentario comienza con /**, doble asterisco, en lugar de uno sólo.

Dentro de los comentarios se puede escribir código HTML (por ejemplo para escribir en negrita se debe de introducir el texto dentro de las etiquetas ...texto...) y operadores, que deben escribirse precedidos por @ para que el intérprete de Javadoc pueda identificarlos.

El ejemplo del comentario para crear la documentación en Javadoc mostrado anteriormente indica quién es el autor del código y cuál es la versión del mismo. A continuación se muestra como documentar una clase:

```
public class Ejemplo {
    int numero;
```

```
    String cadena;

    /**
    * Constructor de la clase
    *
    * @param numero elemento de tipo entero
    * @param cadena elemento de tipo String
    */
    public Ejemplo(int numero, String cadena) {
        this.numero = numero;
        this.cadena = cadena;
    }

    /**
    * Indica si el numero almacenado en numero es cero
    *
    * @return true si numero es cero
    */
    public boolean esCero() {
    return (numero == 0);
    }
}
```

Como nota adicional hay que mencionar que por defecto esta herramienta sólo muestra documentación de elementos públicos *public* o protegidos *protected* del código, con lo que los elementos marcados como privados *private* no aparecerán en la documentación generada. Aún así se puede forzar para que Javadoc también muestre la documentación relacionada con los elementos privados.

A.2. Cómo documentar una aplicación con Javadoc

Una vez se ha descrito la herramienta Javadoc, en este capítulo se explicará como crear esa documentación en el entorno de desarrollo utilizado en este proyecto, NetBeans.

Primero se escribe el código y se documenta con las normas mostradas en el apartado anterior, de modo que el código quede más o menos como se muestra en la figura A.1.

Después en el menú Run hay una opción llamada Generate Javadoc, como se puede ver en la figura A.2. Al pinchar sobre esa opción, NetBeans compilará y generará la documentación.

Por último, en la figura A.3 se puede ver el resultado de generar la documentación.

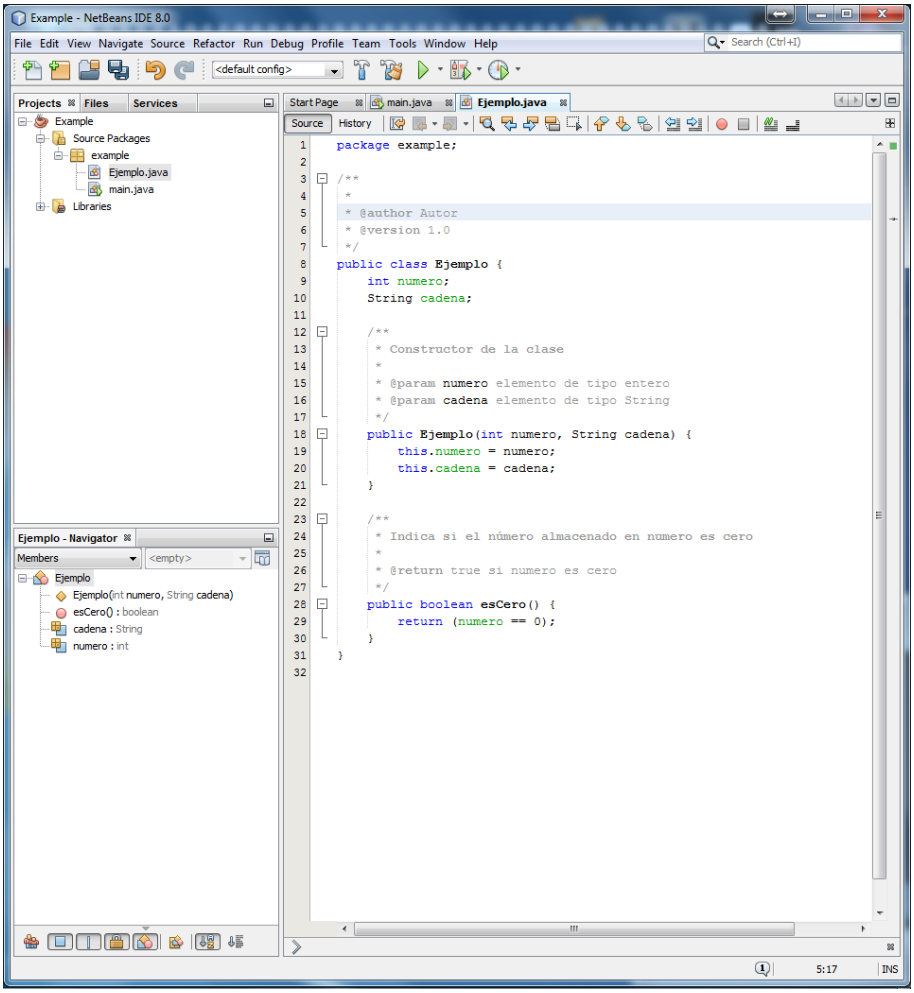

Figura A.1: Código fuente

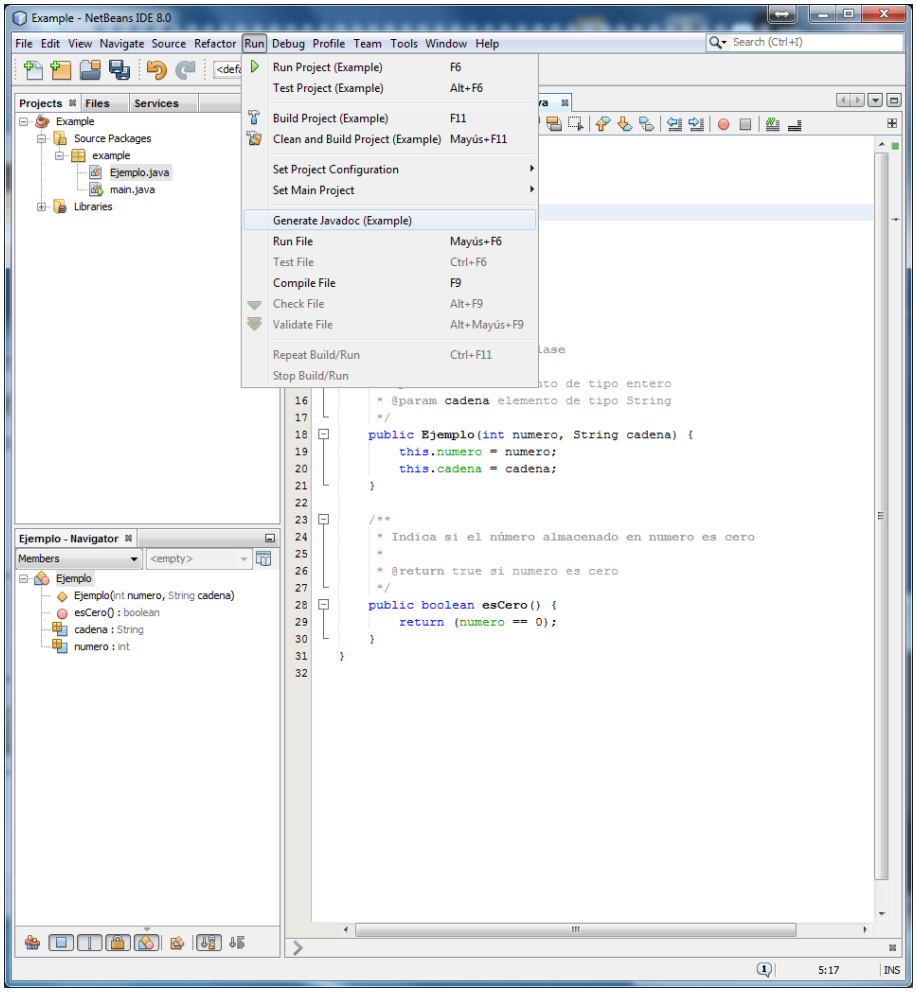

Figura A.2: Crear Javadoc

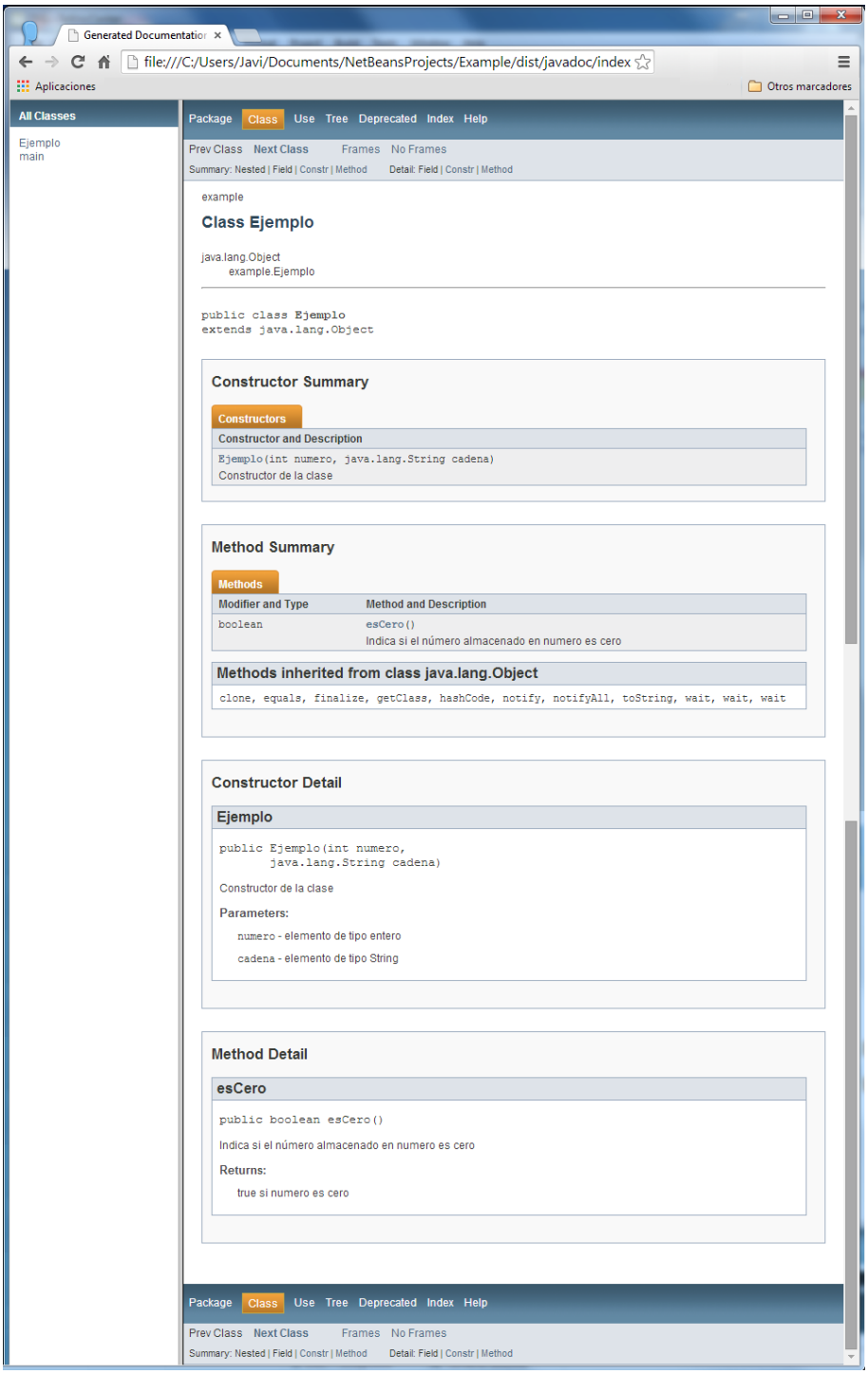

Figura A.3: Resultado de crear la documentación del código con Javadoc

Bibliografía

[1] Caprica software. http://www.capricasoftware.co.uk/projects/vlcj/index.html.

[2] Elphel. http://www3.elphel.com/.

[3] Ffmpeg. http://www.ffmpeg.org/.

[4] Fuente de imagen. http://www.revistaenie.clarin.com/ideas/VideovigilanciaControl-Centro-Monitoreo-Policia-Metropolitana_CLAIMA20120414_0012_19.jpg.

[5] Java native access. https://github.com/twall/jna.

[6] Java web start. http://www.java.com/es/download/faq/java_webstart.xml.

[7] Librería java swing. http://docs.oracle.com/javase/tutorial/uiswing/.

[8] Licencia gpl. http://es.wikipedia.org/wiki/Licencia_GPL.

[9] Open street map. http://es.wikipedia.org/wiki/OpenStreetMap.

[10] Rfc 2326 - real time streaming protocol. http://www.ietf.org/rfc/rfc2326.txt.

[11] Sourceforge. http://sourceforge.net/.

[12] Videolan. http://www.videolan.org/.

[13] Vlc en wikipedia. http://es.wikipedia.org/wiki/VLC_media_player.

[14] vlcj - java bindings for libvlc. https://code.google.com/p/vlcj/.