UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Extensión de *xawtv* para la transmisión de flujos de video mediante ACE-TAO

AUTOR
Ramón García Soto

DIRECTORES
Juan Ángel Pastor Franco
Antonio Javier García Sánchez

Septiembre/2003

Autor	Ramón García Soto
E-mail del Autor	tasiete@teleline.es
Directores	Juan Ángel Pastor Franco Antonio Javier García Sánchez
E-mail del Director	Juanangel.pastor@upct.es
Codirector(es)	
Título del PFC	Ampliación de *xawtv* para la transmisión de flujos de video mediante ACE-TAO
Descriptores	

Resumen

Partiendo de la suite xawtv para la captura y reproducción de video en entorno Linux, se ha diseñado un sistema por el que la aplicación es capaz de enviar y recibir video a través de la red haciendo uso del protocolo UDP y de la implementación de las especificaciones de CORBA contenidas en ACE-TAO. Además, pueden negociarse parámetros sobre la calidad de servicio, y todo ello basado en una filosofía de diseño basada en plugins.

Titulación	Ingeniería Técnica de Telecomunicación, especialidad Telemática
Intensificación	
Departamento	Departamento de Tecnologías de la Información y Comunicaciones
Fecha de Presentación	Septiembre- 2003

ÍNDICE

1 Introducción

Desde la creación de Internet, las redes de datos han sufrido grandes cambios, evolucionando hasta alcanzar un considerable ancho de banda. Esto ha animado a los grupos de I+D a desarrollar software que proporcione cantidad y variedad de servicios, como transmisión de video, voz sobre IP... que requieren conexiones de alta velocidad.

Hoy en día es habitual encontrar en *Internet* este tipo de servicios que ofrecen conversaciones de voz en tiempo real, transmisión de *shows* en directo, videoconferencias, etc, lo que sugiere la importancia que tiene la transmisión de video en tiempo real en los campos lúdico y profesional.

Arrastrados por este interés sobre la transmisión de datos multimedia, han surgido protocolos (RTP), estándares (CORBA) y herramientas que permiten el desarrollo de aplicaciones distribuidas en cualquier lenguaje de programación, como son C, C++, Java...

1.1 Tecnologías basadas en transmisión de datos multimedia en tiempo real

A continuación se dará una breve descripción de las herramientas más idóneas para el desarrollo de software que permita la transmisión en tiempo real de grandes cantidades de datos ofreciendo una calidad de servicio.

En lenguaje C pueden encontrarse librerías útiles, por ejemplo las que implementan el protocolo de transmisión en tiempo real RTP. Prácticamente todas están basadas en el uso de *sockets* (RTP suele funcionar sobre UDP). A partir de ellas se puede conseguir una buena aplicación cliente/servidor para el envío de audio, video... incluso en comunicaciones *multicast*.

Otra gran ventaja de usar C viene dada por la cantidad de código de libre distribución existente para *linux,* que suele estar escrito en C. De esta forma se puede reutilizar parte de dicho código, a veces de gran calidad, para a partir de él construir aplicaciones de audio/video. Por ejemplo, si se está interesado en este último tipo de aplicación, podemos partir de un software que permita capturar y reproducir una imagen y centrarnos únicamente en las cuestiones relacionadas con el envío y la recepción de video a través de la red. Existen programas de este tipo que están disponibles en Internet, como por ejemplo *xawtv*, válido para la captura y reproducción de video (a través de la aplicación *pia* que incluye). Existen también otros reproductores como *XAnim* **[1]**.

Por último, una gran ventaja de C es poder enlazarlo con código C++, del que también existen multitud de librerías.

En lenguaje C++ también se pueden encontrar librerías para transmisión de audio/video en tiempo real. Existen implementaciones de RTP e incluso ORBs (*Object Request Broker*) para este proposito, que implementan la especificación de *Real-Time* CORBA. Un ORB que cumple estas características, es el que se encuentra en las librerías de ACE y se llama The ACE ORB (TAO). Es de libre distribución, de código abierto y sigue fielmente los estándares para las implementaciones de la especificación de Real-Time CORBA, que provee de una QoS (*Quality of Service*) *extremo a extremo* eficiente, previsible y fiable. Estas librerías están disponibles tanto para Sistema Operativo (SO) Windows como Linux.

En cuanto a utilidades para la captura de audio/video programadas en lenguaje C++, se encuentran mucho más facilmente para el SO *Windows*. El problema de estas librerías, normalmente desarrolladas por *Microsoft*, es que aunque son de libre distribución, no son de código abierto y no se ofrece una documentación gratuita "decente". Un ejemplo de esto son las librerías *DirectShow* **[2]**, donde todas sus clases se comportan como "cajas negras" que no están bien documentadas y realizan operaciones que no puede controlar el programador.

Java nos ofrece la posibilidad de desarrollar nuestros propios programas de transmisión de datos multimedia utilizando *sockets*. También ofrece una API de libre distribución llamada The Java Media Framework (JMF) **[3]** que permite incluir flujos de audio, video y otras aplicaciones multimedia en aplicaciones Java y *applets*. Este paquete opcional se usa para capturar, reproducir, transmitir y codificar diferentes formatos de datos multimedia. Esta herramienta es muy útil, pues ahorra carga de trabajo a niveles más básicos de programación de flujos y *sockets*. El precio a pagar es un decremento de la capacidad de personalización y versatilidad.

Se encuentra también en desarrollo una implementación de un ORB para Java. Este ORB se llama ZEN, que es de libre distribución y abierto, y que sigue fielmente las características definidas en la especificación de CORBA 2.6. Su diseño se ha basado en muchas de las técnicas y patrones aprendidos de TAO.

1.2 Objetivos

Se quiere poder transmitir solo video en una LAN (Local Area Network). Para ello se desarrollará una aplicación aplicación Cliente/Servidor.

Partirá de una suite de libre distribución y código abierto llamada *xawtv*, ya comentada anteriormente. Para el extremo del cliente, se usará el reproductor de video *pia*. En el caso del extremo del servidor se partirá de la aplicación xawtv.
Debido a que el código fuente de ésta se encuentra escrito en C y preparado para ser compilado en una Plataforma Unix-Linux, se trabajará sobre un SO de este tipo (Suse, Redhat, Mandrake).

El servidor de la aplicación debe aceptar conexiones de varios clientes y poder negociar ciertos parametros de video y transmisión de éste. Por esto se debe crear una buena forma de crear, añadir y negociar nuevos parámetros. Dentro de los parámetros de transmisión se encuentra la infraestructura de comunicaciones a negociar, que podrá estar constituida por sockets o por un ORB basado en las especificaciones Real-Time CORBA. Este ORB será TAO, mencionado en el punto anterior.

Al contar con un código de partida en C y unas librerías a utilizar en C++, habrá que estudiar la forma de enlazar los dos tipos de código.

2 La aplicación *xawtv*

2.1 Introducción

xawtv **[4]**, fue la primera aplicación creada bajo *linux* para ver la televisión usando el driver *bttv*[1]. Desde entonces varios cambios han ocurrido, el driver *bttv* a dado paso a la nueva interfaz *video4linux*[2] y *xawtv* ya no es una única aplicación, sino una pequeña suite con software relacionado con *video4linux* **[5]**.

Actualmente *xawtv* es el programa más usado en la reproducción y captura de vídeo bajo *linux* y es actualizado constantemente, además destaca por su gran capacidad de configuración y ampliación. Ver figura 2.1

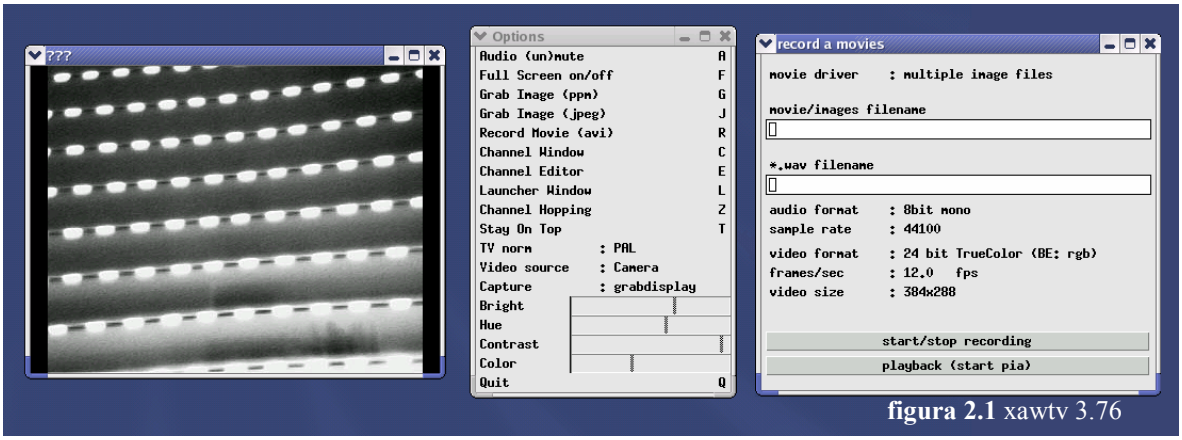


figura 2.1 xawtv 3.76

2.2 Características

Las principales características a tener en cuenta de *xawtv* son:

- Está programado en lenguaje C **[6]** y es *freesource*, lo que permite su ampliación o modificación por otros programadores
- Es equivalente a un *framework* **[7]** en *POO*[3] a pesar de estar escrito en C
- Gran capacidad de ampliación mediante *plugins* **[8]**
- Permite trabajar prácticamente con cualquier dispositivo de captura: tarjetas capturadoras de TV, webcams…
- Permite capturar diferentes tipos de formato y compresión de vídeo/audio (RAW, MJPEG…)
- Su autor lo actualiza constantemente
- Incluye varios subprogramas:
  - **v4l-conf,** para la configuración de *video4linux*
  - **xawtv-remote,** control remoto para *xawtv*
  - **fbtv,** programa de TV para la consola de *linux*
  - **ttv,** utilidad que renderiza la imagen de TV en cualquier terminal de texto
  - **v4lctl,** permite controlar un dispositivo *v4l* desde la línea de comandos y capturar imágenes

[1] *bttv* es el driver de linux para las tarjetas de TV con chips bt848 y bt878
[2] *video4linux* ofrece una interfaz común de programación para la mayoría de tarjetas capturadoras de TV, así como cámaras USB, por puerto paralelo, radio, teletexto…
[3] Programación Orientada a Objetos

- **streamer,** utilidad de captura de vídeo y audio desde la línea de comandos
- **radio,** aplicación que permite escuchar la radio desde la consola
- **videotext / teletext,** servidor *http* de teletexto
- **webcam,** captura imágenes de una *webcam*
- **pia,** reproductor propio
- **scripts** en *perl* para la captura de video
- **otros** programas

2.3 Estructura

Esta aplicación se compone de los siguientes directorios (ver figura 2.2):

DIRECTORIO	*CONTENIDO*
common	ficheros relacionados con la captura de vídeo y audio, así como de la webcam
console	código fuente de utilidades y aplicaciones que se ejecutan desde línea de comandos excepto el reproductor *pia*
contrib	ficheros de configuración según la región (Europa, Japón…) en la que se recibe vídeo, las frecuencias de sintonización de TV…
debian	ficheros para sistema *debian*
debug	código útil para debug de la aplicación
fonts	vacío
jwz	código fuente de la aplicación *xawtv-remote*
libng	ficheros y librerías relacionados con el vídeo. Durante todo el documento nos centraremos en los ficheros y directorios que contiene este directorio. Se verá más adelante
libng/plugins	contiene *plugins* para la aplicación *xawtv*. Se verá más adelante.
libvbi	vacío
man	ficheros de manual sobre la aplicación
scripts	scripts en perl para la captura de video
vbistuff	ficheros del demonio http para la aplicación *videotex*
x11	contiene aplicaciones *pia* y *xawtv* además de sus ejecutables

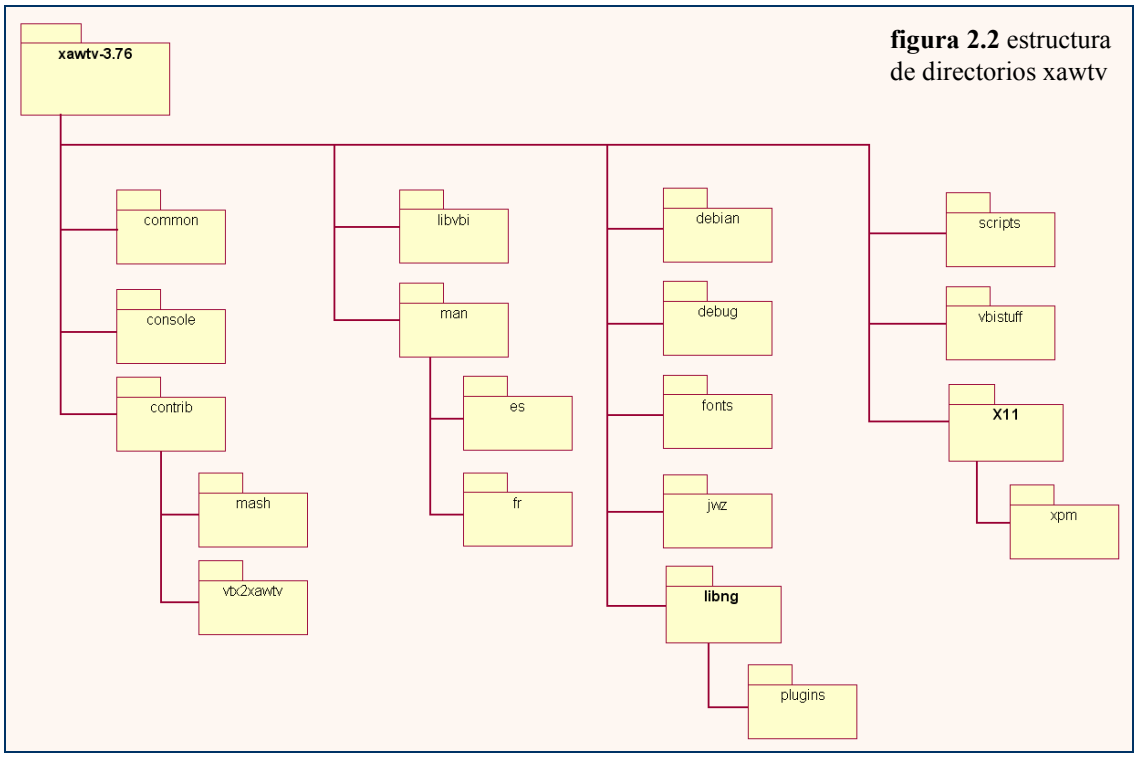


figura 2.2 estructura de directorios xawtv

2.4 Paquetes de interés

De todos los paquetes que contiene *xawtv*, nos centraremos solamente en tres, y en sus ficheros más importantes. Ver figura 2.2.1

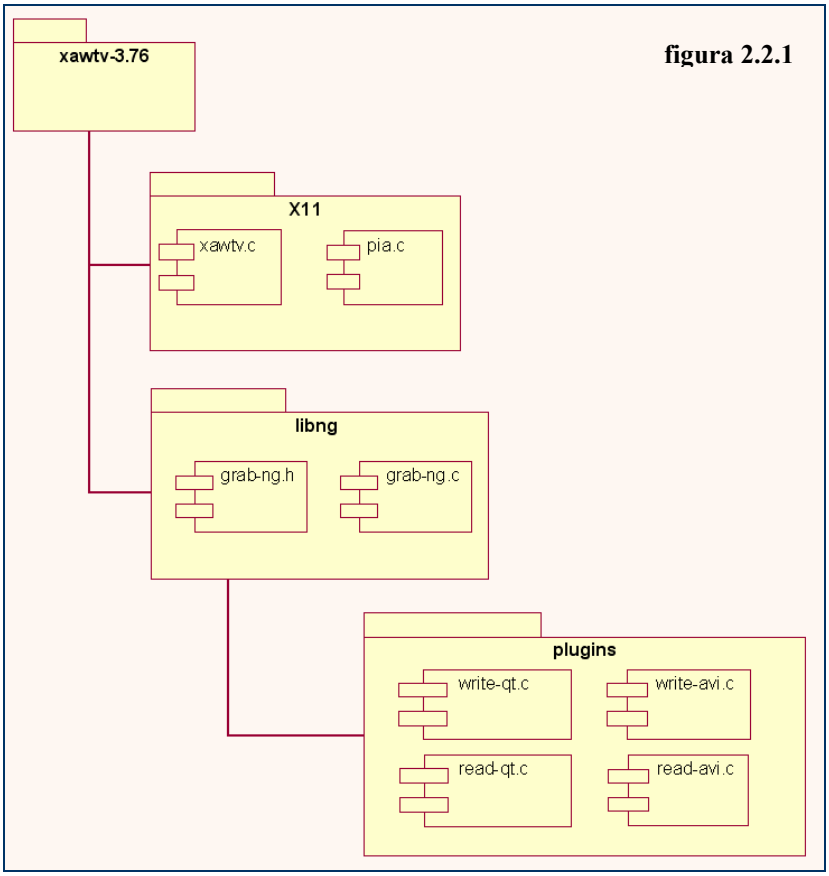


figura 2.2.1

2.4.1 libng

El directorio *libng* contiene los ficheros relacionados con el vídeo (y audio), su captura, formato, tamaño, compresión…

Los ficheros que se han estudiado con detenimiento se detallan a continuación.

2.4.1.1 *grab-ng.h*

Fichero de cabecera de *grab-ng.c*. Define diversas variables constantes globales relacionadas con el formato de vídeo y audio*. Además define las siguientes estructuras (sólo las más importantes):

```
struct ng_video_fmt {
        int    fmtid;
        int    width;
        int    height;
        int    bytesperline;
};
```

La estructura **ng_video_fmt** engloba los parámetros pertenecientes al formato de video. Una imagen tiene un determinado formato de compresión cuyo identificador se guarda en **fmtid**, así como un ancho **width** y una altura **height** (definidos en píxeles). Cuando se trata de un formato de video que no requiere compresión, el valor **bytesperline** será distinto de cero, y valdrá cero cuando haya compresión.

```
struct ng_video_buf {
    struct ng_video_fmt  fmt;
    int                  size;
    unsigned char        *data;

    struct {
       long long         ts;
       int               seq;
       int               twice;
    } info;

    pthread_mutex_t      lock;
    pthread_cond_t       cond;
    int                  refcount;
    void                 (*release)(struct ng_video_buf *buf);
    void                 *priv;
};
```

ng_video_buf contiene una imagen (un *frame*[4]) e información sobre ella. La imagen se guarda en el *array* de caracteres ***data**, también se conoce su tamaño **size** y su

* NOTA: Nos centraremos únicamente en video
[4] *frame*: fotograma

formato de video **fmt**. Se pueden encontrar otros parámetros, como una estampa de tiempo **ts** (normalmente de cuando se captura la imagen), un número de secuencia **seq** útil para la detección de pérdida de *frames* y otros parámetros que no se han estudiado en profundidad.

```
struct ng_audio_fmt {
    int    fmtid;
    int    rate;
};
```

```
struct ng_audio_buf {
    struct ng_audio_fmt   fmt;
    int                   size;
    int                   written;
    char                  *data;

    struct {
      long long         ts;
    } info;
};
```

Las estructuras relacionadas con audio no se han estudiado, aún así **ng_audio_fmt** contiene información sobre el formato de audio, y **ng_audio_buf** contiene un *frame* de audio.

```
struct ng_format_list {
    char  *name;
    char  *desc;
    char  *ext;
    int   fmtid;
    void  *priv;
};
```

Contiene descripción sobre un formato de vídeo como puede ser MJPEG, RGB24, RGB15…

A lo largo de la documentación, nos centraremos fundamentalmente en las estructuras: **ng_writer** y **ng_reader**, ya que a partir de ellas podremos definir *plugins*.

2.4.1.1.1 ng_writer

```
struct ng_writer {
    const char *name;
    const char *desc;
    const struct ng_format_list *video;
    const struct ng_format_list *audio;
    const int combined;

    void* (*wr_open)(char *moviename, char *audioname,
                struct ng_video_fmt *video, const void *priv_video, int fps,
                struct ng_audio_fmt *audio, const void *priv_audio);
    int (*wr_video)(void *handle, struct ng_video_buf *buf);
    int (*wr_audio)(void *handle, struct ng_audio_buf *buf);
    int (*wr_close)(void *handle);
};
```

Imaginemos que deseamos capturar una secuencia de video en un fichero. Los pasos a seguir serían:

1- Crear el fichero
2- Obtener la imagen y guardarla en el fichero
3- Cerrar el fichero

La estructura **`ng_writer`** define una *interfaz* para llevar a cabo tales acciones, que debe "implementarse" mediante *plugins* de forma similar a la implementación de interfaces en POO. Ver figura 2.3.

En el caso que deseáramos realizar una implementación de captura de video a un fichero, se realizaría una implementación de manera que:

- **`(*wr_open)()`** permita crear el fichero donde se guardará la captura
- **`(*wr_video)()`** obtiene el video y lo guarda en el fichero
- **`(*wr_close)()`** cierra los flujos de captura y el fichero

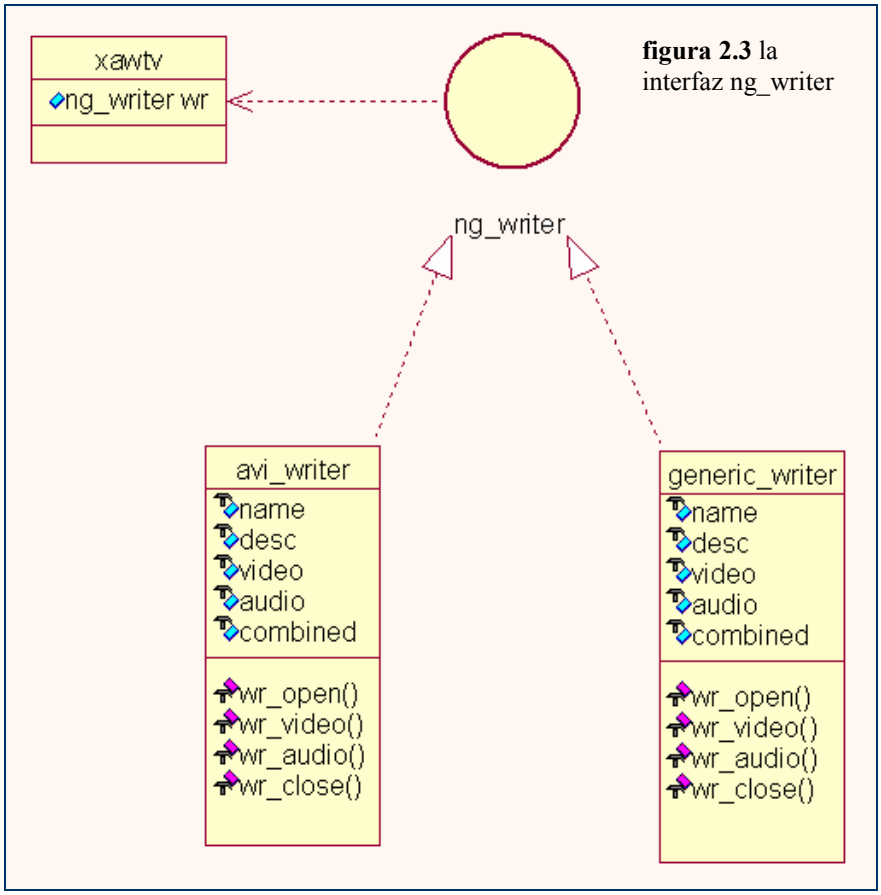


figura 2.3 la interfaz ng_writer

A tal implementación puede asignársele un nombre **`name`** y una breve descripción **`desc`** que será con la que reconozcamos nuestro *plugin* en *xawtv*.

Los formatos de video y audio soportados se guardan en ***`struct`*** *`ng_format_list`* **`*video`** y ***`struct`*** *`ng_format_list`* **`*audio`** respectivamente.

De una forma más general,

```
void* (*wr_open)(char *moviename, char *audioname,
                 struct ng_video_fmt *video,
                 const void *priv_video,
                 int fps, struct ng_audio_fmt *audio,
                 const void *priv audio)
```

donde ***moviename** es una cadena de texto, en *xawtv* es el nombre de fichero donde guardar la captura de video, pero también podría ser una dirección IP… depende de la implementación), lo mismo para ***audioname** solo que considerando el audio. Las estructuras ***struct*** *ng_video_fmt* ***video** y ***struct*** *ng_video_fmt* ***audio** contienen la información sobre el formato de video y audio que se va a capturar. **wr_open()** se ejecuta en el momento de arrancar el *plugin*.

```
int (*wr_video)(void *handle, struct ng_video_buf *buf)
```

La variable ***handle** puede ser de cualquier tipo, es usada internamente en los *plugins*, se estudiará posteriormente, ***buf** contiene el *frame* capturado y su información asociada.

```
int (*wr_close)(void *handle)
```

Se encarga de cerrar flujos (siempre de salida/escritura).

La aplicación *xawtv* que es la encargada de captura de video (flujos de salida), se encargará de manejar esta “interfaz” **ng_writer**.

2.4.1.1.2 ng_reader

```
struct ng_reader {
    const char *name;
    const char *desc;

    char  *magic[4];
    int   moff[4];
    int   mlen[4];

    void* (*rd_open)(char *moviename, int *vfmt, int vn);
    struct ng_video_fmt* (*rd_vfmt)(void *handle);
    struct ng_audio_fmt* (*rd_afmt)(void *handle);
    struct ng_video_buf* (*rd_vdata)(void *handle, int drop);
    struct ng_audio_buf* (*rd_adata)(void *handle);
    long long (*frame_time)(void *handle);
    int (*rd_close)(void *handle);
};
```

Supongamos un caso distinto al que se da en **`ng_writer`**, ya tenemos un fichero que contiene video y deseamos reproducirlo, tendríamos que:

1- Abrir el fichero
2- Reproducirlo
3- Cerrar el fichero

La estructura **`ng_reader`** define una interfaz para llevar a cabo tales acciones, que debe "implementarse" mediante *plugins* de forma similar a la implementación de interfaces en POO al igual que en el caso anterior **`ng_writer`**. Ver figura 2.4.

En el caso que deseáramos realizar una implementación de lectura de video de un fichero, se realizaría una implementación de manera que:

- **`(*rd_open)()`** permita abrir el fichero donde se guarda la captura
- **`(*rd_vdata)()`** obtiene el video contenido en el fichero y lo "devuelve" a *xawtv*
- **`(*rd_close)()`** cierra los flujos de lectura y el fichero

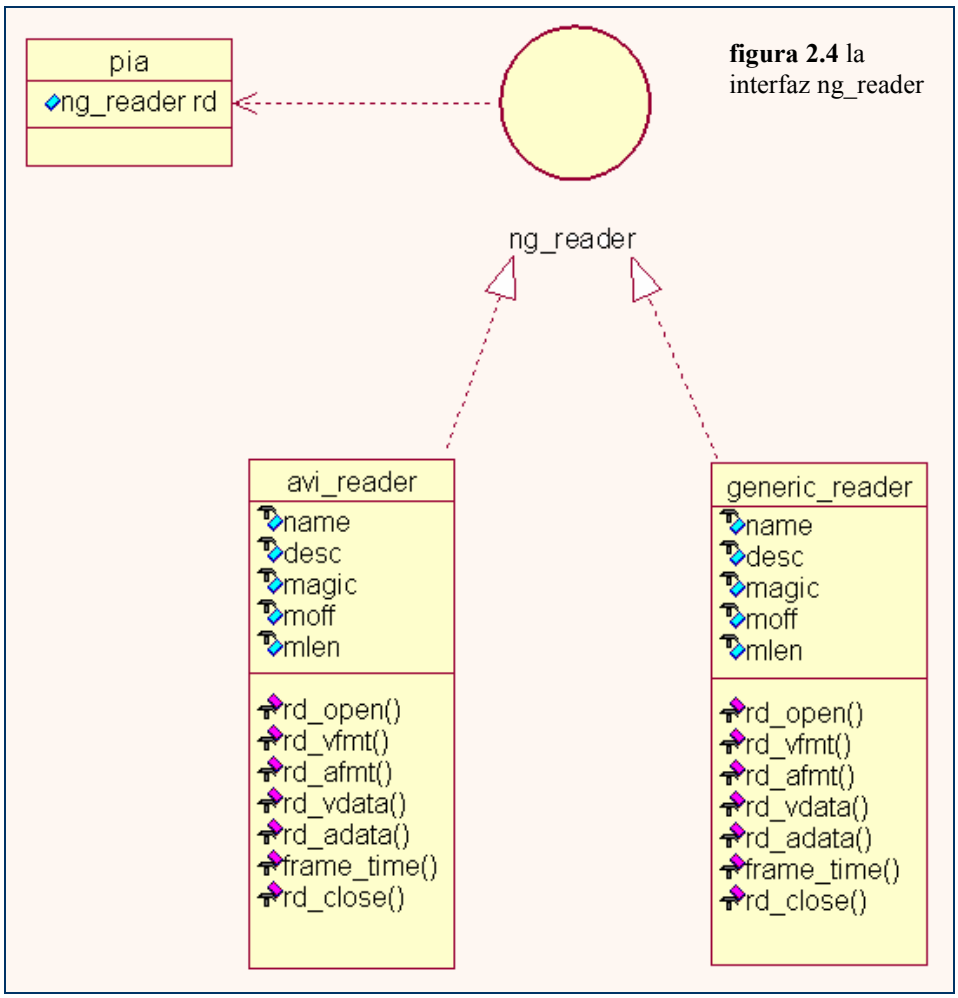


figura 2.4 la interfaz ng_reader

A tal implementación puede asignársele un nombre **name** y una breve descripción **desc**.

De una forma más general,

```
void* (*rd_open)(char *moviename, int *vfmt, int vn)
```

donde ***moviename** es una cadena de texto (normalmente será el nombre de fichero donde está el video, pero también podría ser una dirección IP… siempre dependiendo de la implementación). La variable ***vfmt** contiene el valor entero asociado a un formato de video. **rd_open()** se ejecuta en el momento de arrancar el *plugin*.

```
struct ng_video_fmt* (*rd_vfmt)(void *handle)
```

Esta función devuelve una estructura de tipo ***struct** *ng_video_fmt** con el formato de video que se va a reproducir. La variable ***handle** puede ser de cualquier tipo, al igual que en el caso de **ng_writer**.

```
struct ng_video_buf* (*rd_vdata)(void *handle, int drop)
```

Devuelve una estructura ***struct** *ng_video_buf** que contiene el *frame* leído del fichero. **drop** contiene el número de *frames* que se han eliminado durante la captura.

```
long long (*frame_time)(void *handle)
```

Se utiliza para manejar los frames eliminados (*dropped*). Si el tiempo transcurrido entre dos *frames* es excesivo, se eliminarán *frames*.

```
int (*wr_close)(void *handle)
```

Se encarga de cerrar flujos (de entrada/lectura).

xawtv no se encarga de manejar flujos de lectura (reproducir ficheros…), de ello se encarga la aplicación *pia* incluida en la suite.

2.4.1.2 *grab-ng.c*

Contiene funciones relacionadas con las "interfaces" **`ng_reader`** y **`ng_writer`**. Se encarga de:

- Cargar
- Registrar
- Inicializar

} *plugins*[5]

Además contiene funciones que controlan el formato de video. Centrándonos en el uso de los *plugins,* las funciones más importantes son:

```
int ng_writer_register(int magic, char *plugname, struct ng_writer *writer)
```

Registra un **`ng_writer`** en una lista de **`ng_writers`**. De esta forma *xawtv* puede acceder a los *plugins* de escritura. Esta función es llamada desde los *plugins*.

```
int ng_reader_register(int magic, char *plugname, struct ng_reader *reader)
```

Registra un **`ng_reader`** en una lista de **`ng_readers`**. De esta forma *pia* puede acceder a los *plugins* de lectura. Se llama desde los *plugins.*

```
static int ng_plugins(char *dirname)
```

Carga todos los *plugins* que se encuentran en el directorio **`*dirname`** y los registra. Un *plugin* se compila como librería dinámica[6]. Esta función carga todas las librerías dinámicas de un directorio. Ver figura 2.5.

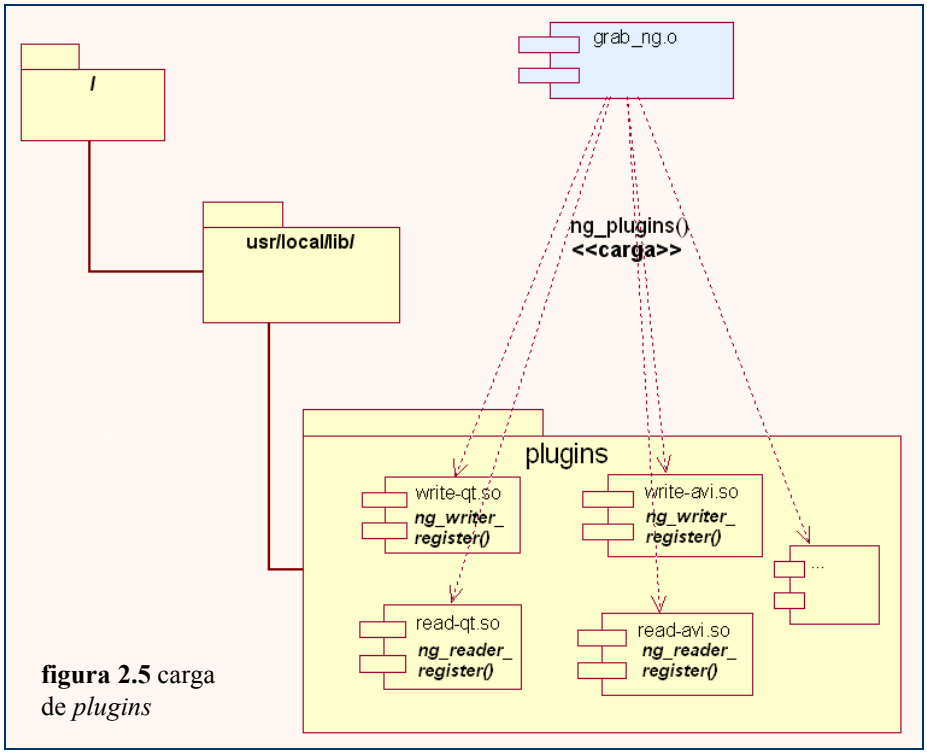


figura 2.5 carga de *plugins*

[5] Ahora que conocemos las estructuras dedicadas a manejar el flujo de la información, podemos decir que *plugin*: implementación de **`ng_reader`** o de **`ng_writer`**

[6] librerías que se cargan y ejecutan en tiempo de ejecución

```
void ng_init(void)
```

Función de inicialización. Inicializa los formatos de video, los dispositivos de captura y se encarga de llamar a **`ng_plugins()`**. La función **`ng_init()`** es llamada cuando se arranca *xawtv* o bien la aplicación *pia*.

2.4.2 libng/plugins

Es el directorio que contiene los *plugins*. Por defecto *xawtv* contiene:

PLUGIN	*IMPLEMENTA*
write-avi.c	ng-writer
read-avi.c	ng-reader
write-qt.c	ng-writer
read-qt.c	ng-reader

Nos centraremos en *write-avi.c* y *read-avi.c*. Los otros dos *plugins write-qt.c* y *read-qt.c* realizan la misma función, salvo que trabajan con compresión en formato *QuickTime.*

2.4.2.1 write-avi.c

En este fichero es un *plugin*. Implementa la estructura **`ng_writer`** (ver figura 2.6).

Es compilado como librería dinámica y cargado desde *xawtv* al inicio de la aplicación. Se registra con la descripción *Microsoft AVI (RIFF) format* mediante la cual podremos reconocerlo desde *xawtv*. Ver figura 2.7.

Captura video desde la aplicación *xawtv* y la guarda en un fichero.

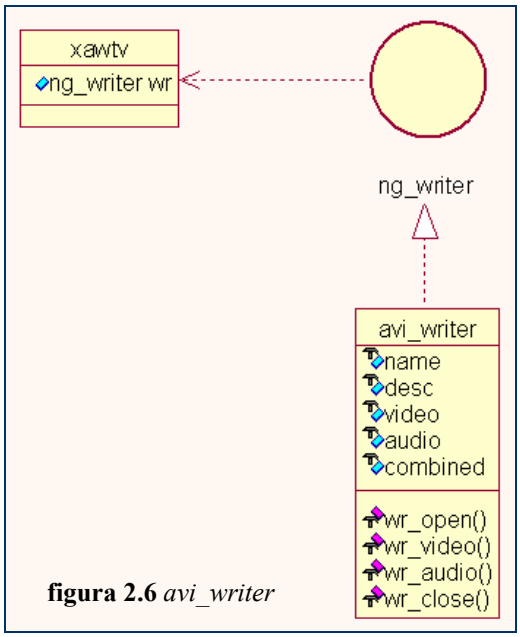


figura 2.6 *avi_writer*

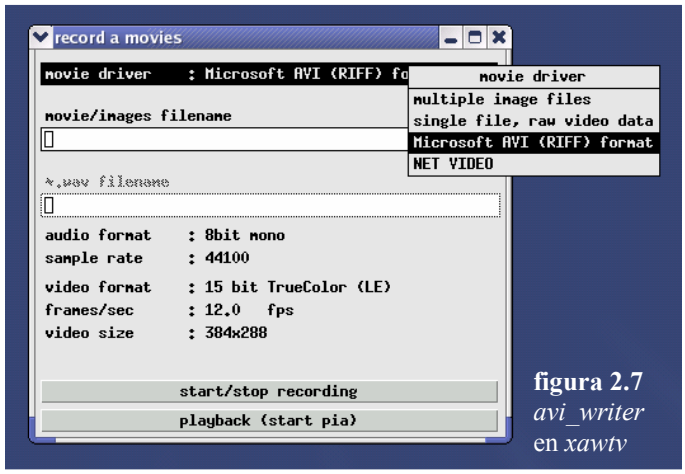


figura 2.7 *avi_writer* en *xawtv*

La asociación entre **ng_writer** y **avi_writer**:

```
struct ng_writer avi_writer = {
    name:       "avi",
    desc:       "Microsoft AVI (RIFF) format",
    combined:   1,
    video:      avi_vformats,
    audio:      avi_aformats,
    wr_open:    avi_open,
    wr_video:   avi_video,
    wr_audio:   avi_audio,
    wr_close:   avi_close,
};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
avi_open()	wr_open()	Abre fichero donde se guardará el video
avi_video()	wr_video()	Captura *frame* de video y lo guarda en fichero
avi_audio()	wr_audio()	Captura *frame* de audio y lo guarda en fichero
avi_close()	wr_close()	Cierra el fichero

Una vez que se ha definida la implementación, se debe registrar el *plugin* para poder ser reconocido y manejado por *xawtv*.

```
void ng_plugin_init(void){
      ng_writer_register(NG_PLUGIN_MAGIC,__FILE__,&avi_writer);
}
```

ng_writer_register(), que recordemos, está definida en *grab-ng.c*, se encargará de registrar **avi_writer** en una lista de **ng_writers**. La función **ng_plugin_init()** es llamada desde **ng_plugins()** (ver apartado 2.4.1.2).

2.4.2.2 read-avi.c

Al igual que en el caso anterior es un *plugin*. Implementa la estructura **ng_reader** (ver figura 2.8).

Se compila como librería dinámica y se carga desde el reproductor *pia* (figura 2.9) al inicio. Carga un fichero de video y lo reproduce.

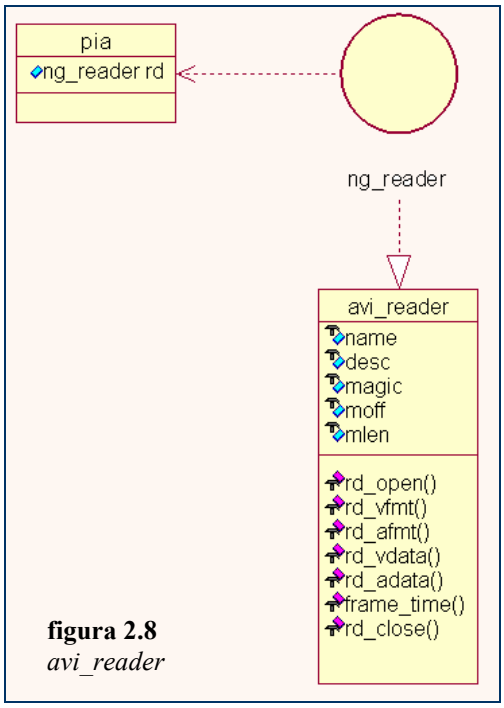


figura 2.8
avi_reader

figura 2.9 *pia*

```
struct ng_reader avi_reader = {
    name:       "avi",
    desc:       "Microsoft AVI (RIFF) format",

    magic:  { "RIFF" },
    moff:       {  0     },
    mlen:       {  4     },

    rd_open:    avi_open,
    rd_vfmt:    avi_vfmt,
    rd_afmt:    avi_afmt,
    rd_vdata:   avi_vdata,
    rd_adata:   avi_adata,
    frame_time: avi_frame_time,
    rd_close:   avi_close,
};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
avi_open()	rd_open()	Abre fichero donde se guarda el video
avi_vdata()	rd_video()	Lee un *frame* de video y lo devuelve a *pia* para ser reproducido
avi_adata()	rd_audio()	Lee un *frame* de audio y lo devuelve a *pia* para ser reproducido
avi_close()	rd_close()	Cierra el fichero

Una vez definida la implementación, se registra el *plugin* para poder ser reconocido y manejado por *pia*.

```
void ng_plugin_init(void){
      ng_reader_register(NG_PLUGIN_MAGIC,__FILE__,&avi_reader);
}
```

ng_reader_register() está definida en *grab-ng.c*, se encarga de registrar **avi_reader** en una lista de **ng_readers**. La función **ng_plugin_init()** se llama desde **ng_plugins()** (ver apartado 2.4.1.2).

2.4.3 x11

Contiene los programas compilados de la suite *xawtv*. Entre ellos los dos en los que nos centraremos a lo largo del estudio, *xawtv* y *pia*.

2.4.3.1 xawtv.c

xawtv[*] es la aplicación principal de la suite. Se encarga de obtener la imagen de una cámara a través de una tarjeta sintonizadora, webcam… además, permite capturar video usando diversos formatos de compresión.

Una de sus ventajas es que se puede ampliar su funcionalidad mediante *plugins*. Como ya se ha visto, *xawtv* es el encargado de manejar los flujos de escritura/salida, es decir, gestiona las estructuras **ng_writer** registradas.

Debido al inteligente diseño con que el cuenta la aplicación, no ha sido necesario estudiar con profundidad este fichero para poder añadir funcinalidad al programa. Únicamente debemos centrarnos en el uso de *plugins* y en el paquete *libng*.

Mención aparte merece el tema de la compilación, que se verá más adelante (apartado 2.6).

2.4.3.2 pia.c

pia[**] es el reproductor disponible. Su utilidad es más bien escasa, permite reproducir muy pocos formatos de video. Además, no es configurable, y carece de interfaz de usuario (es a modo consola). Sin embargo, es poco complejo y fácil de usar.

Como ventaja, puede adquirir mayor versatilidad con el uso de *plugins*. Al ser un programa de reproducción, es el encargado de manejar flujos de lectura/entrada. Será el encargado de manejar las estructuras **ng_reader** registradas.

Para salir de la aplicación *pia* hay que presionar la tecla ‘Q’.

[*] ver **figura 2.1**
[**] ver **figura 2.9**

2.5 Ampliación mediante *plugins*

Hasta ahora se ha visto la función de cada paquete de forma individual. También se ha explicado la carga de *plugins*, en concreto la carga de los *plugins* **`avi_writer y avi_reader`** contenidos en *write-avi.c* y *read-avi.c* respectivamente.

Los ficheros de los *plugins* carecen de fichero de cabecera *.h*, entonces, ¿cómo es posible que *xawtv* y *pia* los reconozcan?, y lo más importante, ¿puedo añadir *plugins* y que sean detectados automáticamente sin modificar ni una sola línea de código?. La respuesta a la primera pregunta es sencilla, los *plugins* son compilados como librerías dinámicas *.so* y son cargadas en tiempo de ejecución. Respecto a la segunda pregunta, podemos responder que sí, pero siempre que dichos *plugins* "implementen" las estructuras **`ng_writer`** (ver figura 2.10) y **`ng_reader`** (ver figura 2.11) definidas en *grab-ng.h* y sean registrados mediante las funciones **`ng_writer_register()`** y **`ng_reader_register()`**. Ver figura 2.13.

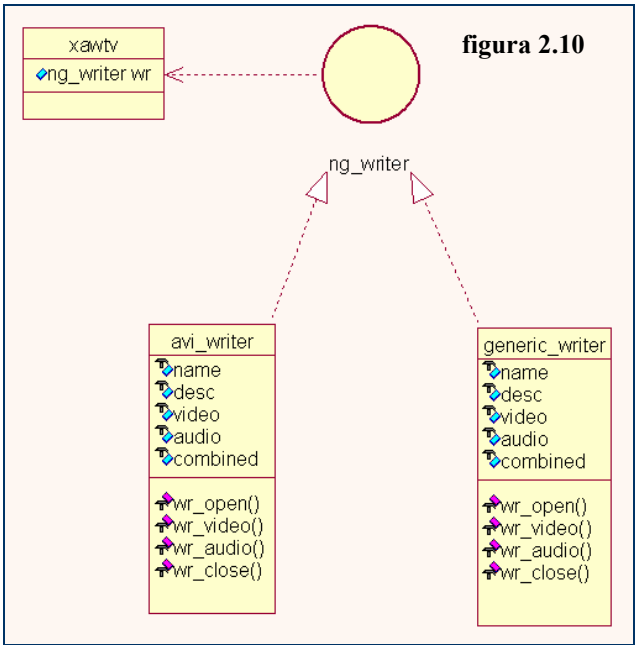


figura 2.10

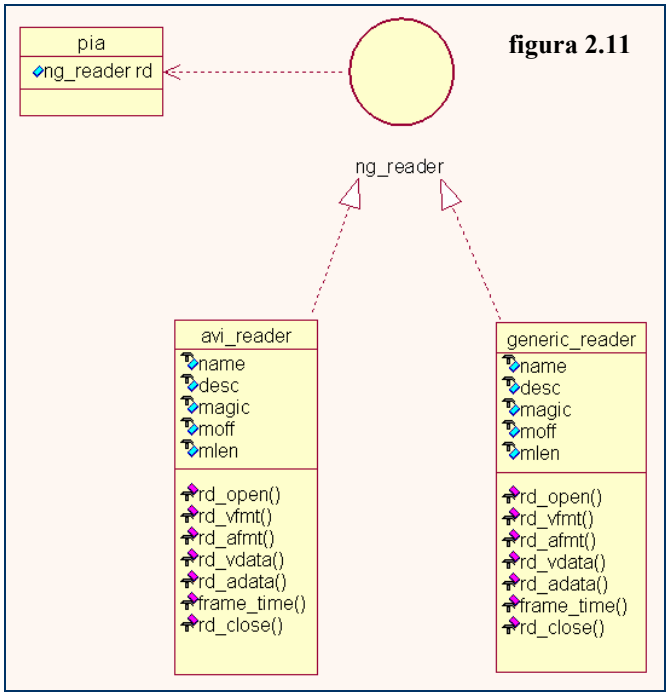


figura 2.11

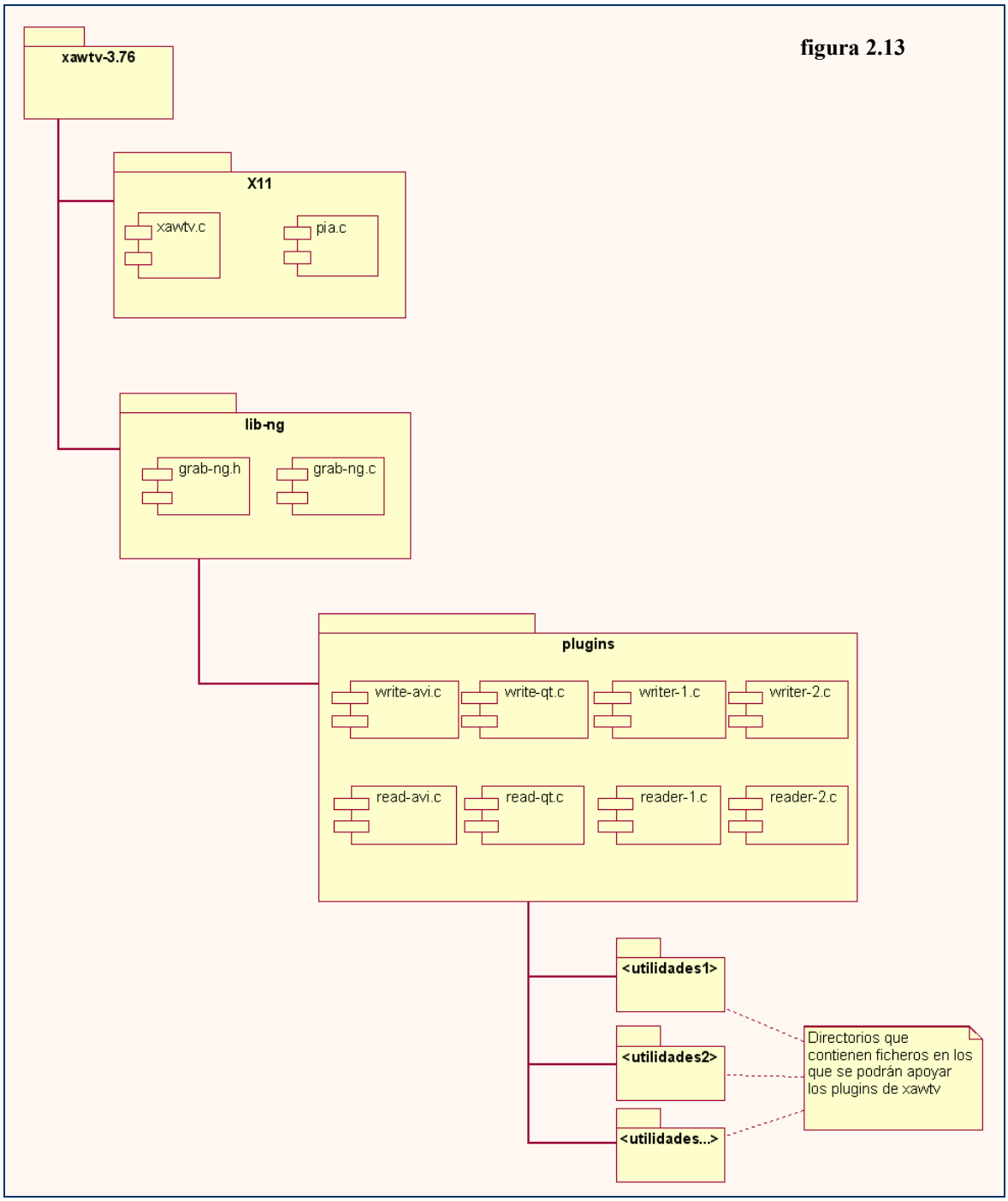

xawtv-3.76
X11
xawtv.c
pia.c
lib-ng
grab-ng.h
grab-ng.c
plugins
write-avi.c
write-qt.c
writer-1.c
writer-2.c
read-avi.c
read-qt.c
reader-1.c
reader-2.c
<utilidades1>
<utilidades2>
<utilidades...>
Directorios que contienen ficheros en los que se podrán apoyar los plugins de xawtv

figura 2.13

2.6 Compilación

La compilación **[9]** de *xawtv* es sencilla, hay que ejecutar en del directorio raiz

```
./configure
```

que prepara una configuración adecuada a nuestro equipo para la compilación (busca versión instalada de gcc…). En concreto, a partir de un fichero de esqueleto *Makefile.in* se genera el script **[10]** *configure*. Al ser ejecutado genera un *makefile*, un fichero de cabecera que define directivas que dirigen la compilación y diversos ficheros de log.

Una vez generado el *makefile*, debemos ejecutarlo por medio de la utilidad

```
make
```

xawtv utiliza en su compilación un método distribuido de ficheros con reglas de compilación en cada uno de los subdirectorios. Cada subdirectorio contiene un fichero llamado *Subdir.mk*. Una vez ejecutado el *makefile* principal, se recorren todos los subdirectorios para compilar los fuentes a partir de los ficheros *Subdir.mk*.

Por último, ejecutaremos (como *root*)

```
make install
```

que instalará los ficheros compilados en nuestro sistema. En concreto copiará todos los *plugins* en el directorio */usr/local/lib/xawtv* para poder acceder a ellos en tiempo de ejecución (indistintamente de dónde tengamos instalado *xawtv*). Ver figura 2.13. Más detalles sobre la compilación en el apartado 8.

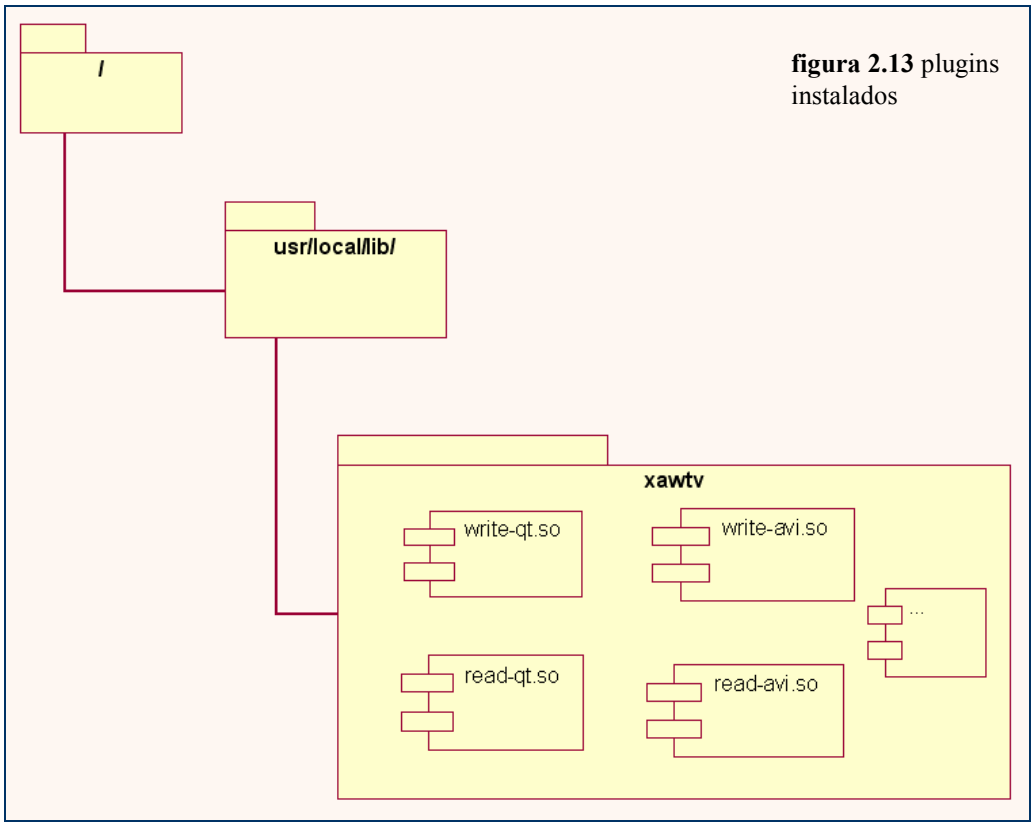


figura 2.13 plugins instalados

3 Creación de *plugins* para la transmisión de video

Partiendo de las aplicaciones *xawtv* y *pia*, se ha diseñado un *plugin* para *xawtv* que se encarga de enviar video, y otro que permite a la aplicación *pia* recibirlo. Usaremos las librerías *socket* disponibles en *Linux*. El *plugin* para *xawtv* se encuentra en el fichero *write-net-avi.c* y para *pia* en *read-net-avi.c* (ver figuras 3.1, 3.2, 3.3 y 3.4).

El servidor permite manejar varios clientes (admite varias conexiones). Las conexiones por parte del cliente se realizan a través del protocolo TCP. Además, una vez recibida la petición de conexión por el servidor, el cliente tiene la capacidad de negociar una serie de propiedades sobre la conexión y el video que va a recibir. Este proceso de negociar se realiza mediante una conexión TCP, que se cierra cuando termina la negociación. Surge entonces la necesidad de crear un mecanismo sencillo para poder añadir nuevas propiedades de negociación.

El cliente se queda a la espera de recibir video, y es el servidor (que ya tiene conocimiento de qué cliente desea recibir video, pues guardó su dirección IP en una tabla interna durante la conexión TCP) el que realiza una conexión con el cliente para enviarle el video. El protocolo de transmisión de video ha sido negociado previamente. Así, el servidor puede enviar a cada cliente usando un protocolo diferente. Para que la aplicación permita el uso de varios protocolos de transmisión de video simultáneamente, se ha diseñado la interfaz *controlador*. Esta interfaz puede ser implementada con la funcionalidad de distintos protocolos. Permite una gran facilidad a la hora de añadir nuevos protocolos de transmisión de video, como pueden ser UDP, RTP, AVStreams de Corba, etc.

Por último, ya que la aplicación *pia* no dispone de GUI, se ha programado una interfaz gráfica en lenguaje JAVA que se comunica con la aplicación *pia*. Esta GUI permite configurar propiedades sobre la conexión y ofrece al usuario estadísticas durante la recepción del video. Importante recordar, que para salirse de la aplicación *pia* debe presionarse la tecla 'Q', aunque se trabaje con la GUI.

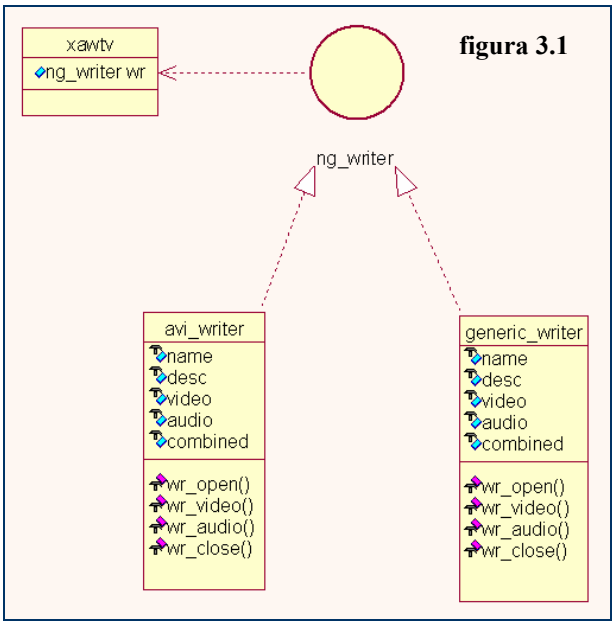


figura 3.1

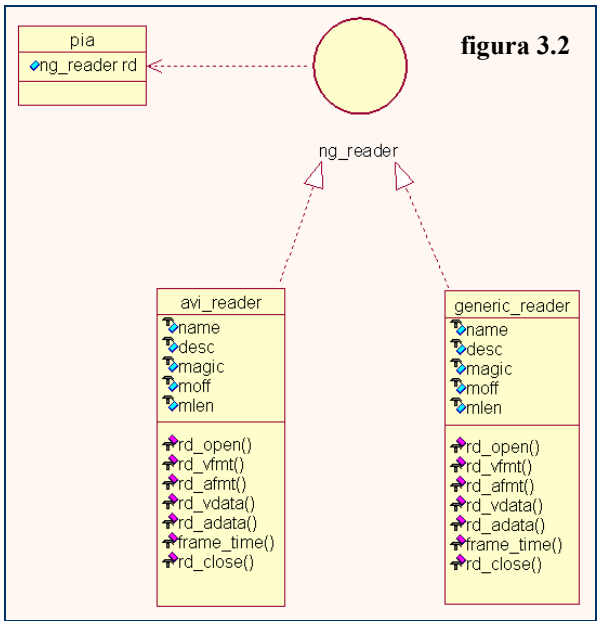


figura 3.2

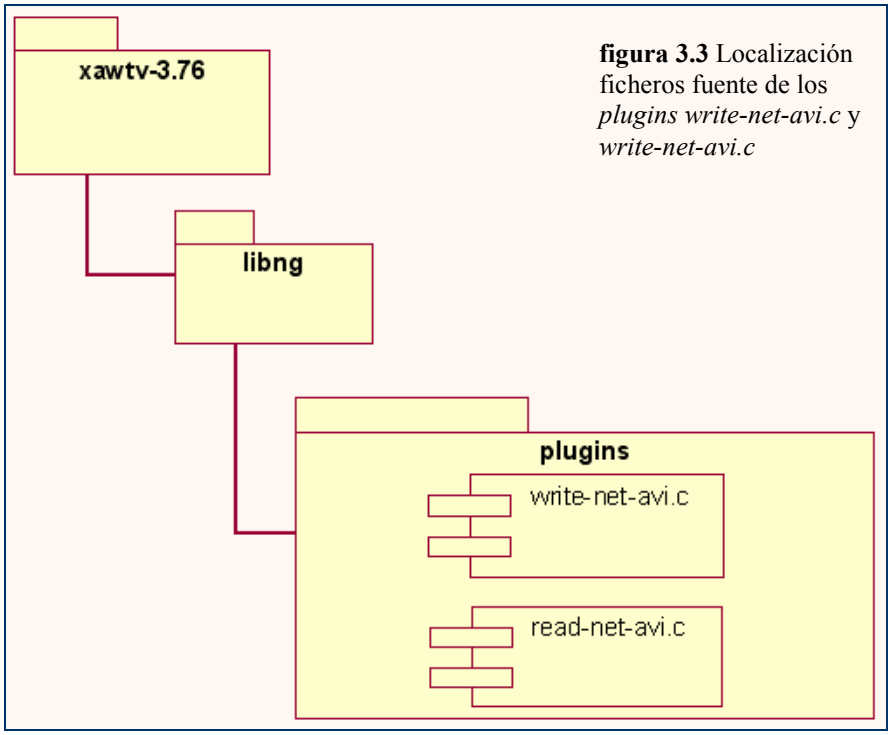


figura 3.3 Localización ficheros fuente de los *plugins write-net-avi.c* y *write-net-avi.c*

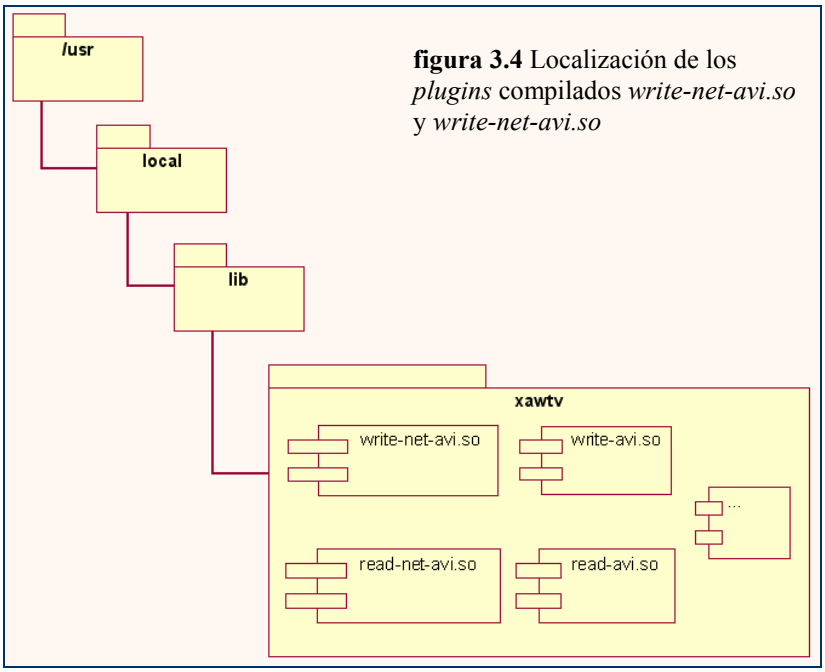


figura 3.4 Localización de los *plugins* compilados *write-net-avi.so* y *write-net-avi.so*

4 Protocolo de inicialización y negociación TCP

4.1 Introducción a TCP

TCP e IP **[11]** fueron desarrollados por el Departamento de Defensa norteamericano, para llevar a cabo un proyecto que pretendía conectar diferentes redes diseñadas por diferentes organismos en una red de redes.

El protocolo TCP se encuentra en la capa de transporte **[12]**, y está definido en *rfc793 – Transmission Control Protocol*, se describen a continuación las características principales.

4.1.1 Servicio

TCP funciona junto a IP. El protocolo IP por sí solo no asegura la entrega de datagramas, ya que es un protocolo:

- **sin conexión,** datagramas con mismo origen y destino son tratados de forma independiente por los routers y pueden seguir caminos distintos
- **no confiable,** no asegura la entrega de datagramas
- de entrega **best-effort,** hará lo posible para que los datagramas sean entregados

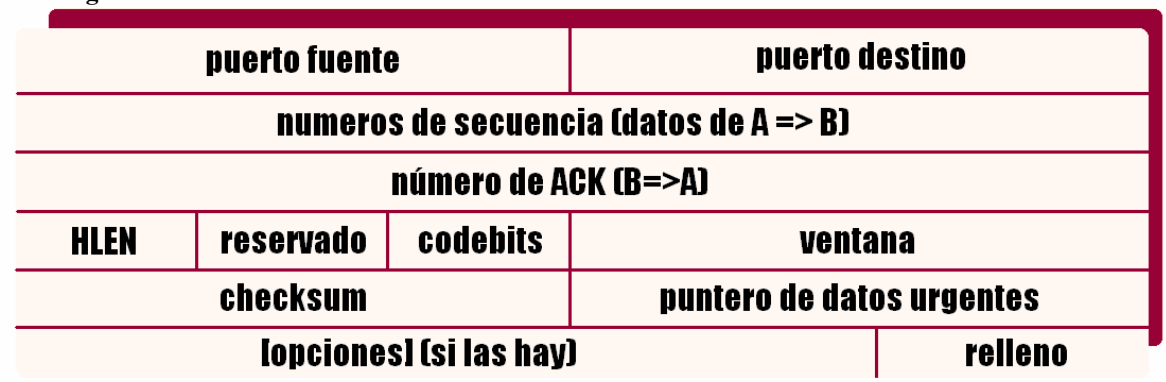
Pueden haber pérdidas de datagramas IP por diversos motivos: fallo en el hardware de red, *switchs* o *routers* congestionados que descartan datagramas… Existe así una necesidad de proporcionar a aplicaciones fiabilidad sobre el protocolo IP (no confiable).

TCP añade a IP la funcionalidad necesaria para conseguir unas comunicaciones fiables. Así, las características del servicio de TCP son:

- Orientado a **conexión**.
- Entrega **fiable** y ordenada: semántica “como mucho una vez”.
- Datos no estructurados. La estructura la determinan las aplicaciones.
- **Segmentación** para mayor eficiencia: por defecto el módulo TCP organiza los datos para no transmitir segmentos muy grandes o muy pequeños.
- **Conexión** full dúplex **simétrica**: tras establecer la conexión el extremo emisor y receptor son exactamente iguales. Cualquiera de ellos puede cerrar la conexión.

4.1.2 Vocabulario y formato

figura 4.1 formato trama TCP



- **Puerto fuente/destino**: Identifican la conexión TCP.
- **HLEN**: Longitud del encabezado del segmento (que puede variar por el campo *Opciones*).
- **Code**: Seis bits que caracterizan el segmento:
  - *URG*: Lleva datos fuera de banda, el campo "Puntero a urgentes" es válido.
  - *ACK*: Se confirman datos, el campo "Número de ACK" es válido.
  - *PSH*: Se solcita la operación de PUSH.
  - *RST*: Reset de la conexión.
  - *SYN*: Sincronizar números de secuencia.
  - *FIN*: Indica al receptor que el emisor no va a enviar más datos.
- **Checksum**: Para esta suma de verificación se debe conocer la dirección IP origen, lo que está en contradicción con la independencia entre capas.
- *Comunicación A⇐B*:
  - **Número de secuencia**: Identifica el primer octeto de los datos que lleva el segmento en el campo DATOS.
  - **Puntero de urgencia**: El segmento puede llevar datos que se entregan a la aplicación B en modo "fuera de banda".
- *Comunicación B⇐A*:
  - **Número de ACK**: Identifica el siguiente octeto que A espera en la comunicación B ⇑A. Esto indica cuantos datos confirmados tiene A.
  - **Ventana**: Este campo lo utiliza A para indicarle a B cuál es el tamaño de su ventana de recepción, lo que limita la ventana de transmisión de B. Es un mecanismo de control de flujo extremo a extremo.
- **Opciones**: La negociación de algunos parámetros del protocolo se hace utilizando este campo. Por ejemplo el tamaño máximo del segmento (MSS).

4.1.3 Procedimiento

Ya que sabemos la función del protocolo TCP, vamos a ver cómo aporta esa confiabilidad que lo caracteriza:

- Las técnicas para proporcionar confiabilidad se basan en el acuse de recibo (*ACK*) cuando un datagrama llega bien, y retransmisión por parte del origen en caso contrario. Ver figura 4.2.

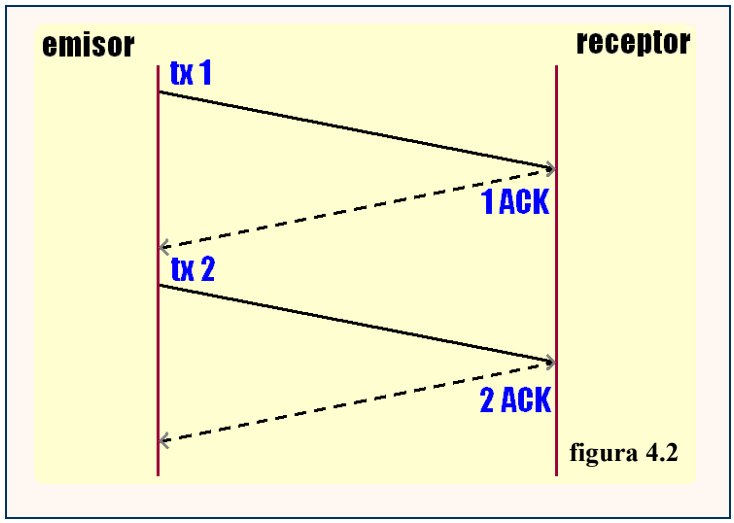


figura 4.2

- Se llama ventana de transmisión al tamaño de los datos que el extremo transmisor puede enviar sin recibir acuse de recibo. El transmisor debe almacenar en memoria esos datos hasta que haya sido confirmada su recepción. Figura 4.3

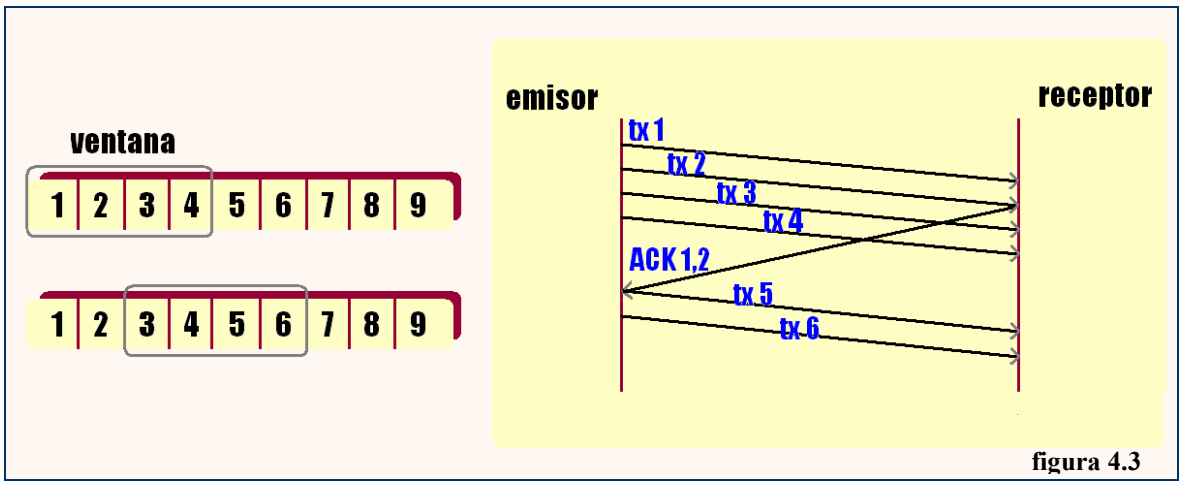


figura 4.3

- La ventana de transmisión se puede ver como una ventana que se desliza sobre la lista de datos que transporte debe enviar.

Hay que tener en cuenta que los datos a transmitir se dividen en segmentos con un tamaño tal que a ser posible no sea fragmentado en varios datagramas IP (alrededor de 500 *bytes*, valor negociado en el establecimiento de la conexión).

Para que se pueda establecer una conexión tcp, es necesario además de la dirección IP, un puerto, así una conexión TCP viene identificada por 4 valores (figura 4.4):

- Extremo origen IP / puerto
- Extremo destino IP / puerto

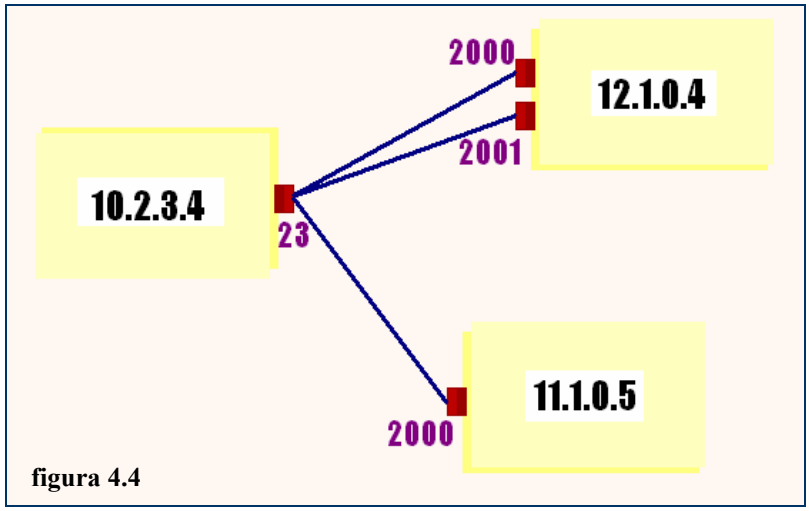


figura 4.4

Por ejemplo, un servidor Telnet escuchando en el puerto 23, puede recibir varias conexiones de un mismo ordenador, cada una de ellas independiente ya que el puerto origen es distinto.

La apertura de la conexión no es simétrica, existe:

- Extremo iniciador: Inicia la conexión, indicando dirección y puerto del otro extremo.

- Extremo receptor: Debe previamente registrarse en el S.O. indicándole que acepta peticiones de conexión en un determinado puerto.

4.2 Negociación de Propiedades

4.2.1 Introducción

Un cliente puede necesitar recibir video a través de la red por distintos motivos, o incluso puede que la recepción esté condicionada por factores ajenos a él.

Los usuarios que no dispongan de una red de alta velocidad, si desean obtener un video fluido, requerirán un buen formato de compresión, a costa de perder parte de la calidad original de la imagen, o incluso en sistemas de seguridad, tal vez no interese un video fluido, sino recibir una tasa menor de fotogramas. En cambio, en aplicaciones de visión artificial, es recomendable el uso de video sin comprimir para poder analizar las imágenes que se reciban a tiempo real. En definitiva, diferentes aplicaciones tienen diferentes necesidades.

Es ahí cuando surge la necesidad de una negociación que permita establecer una calidad de servicio.

Para ello, como parte del proyecto, se ha definido una interfaz *property* que deben respetar todas las propiedades que se deseen negociar. De esta forma, añadir una nueva propiedad es tarea sencilla, y no rompe con el esquema de diseño del programa.

4.2.2 Propiedades

4.2.2.1 La estructura *Property*

```
#define MAX_VALUES 10
#define MAX_PROPERTIES 20

struct Property{

      char   name[20];
      int    id;
      int    value[MAX_VALUES];
      int   (*setValue)(int val);
      int   (*setValues)(int val[MAX_PROPERTIES]);
      int   (*getValue)(int pos);
      int*  (*getValues)(void);
      void* (*serialize)(int source);
      void* (*deserialize)(int *package);
};
```

Incluida en el fichero *grab-ng.h*. Permite especificar un nombre de propiedad **`name`** y un identificador **`id`**. Cada propiedad permite almacenar hasta 10 valores en el array de enteros **`value`**. Valores que pueden modificarse y obtenerse a través de implementaciones de funciones como **`*setValue()`**, **`*setValues()`**, **`*getValue()`** y **`*getValues()`**. Las funciones **`*serialize()`** y **`*deserialize()`** están pensadas para preparar las propiedades de forma que puedan ser enviadas a través de la red. Ver figura 4.5

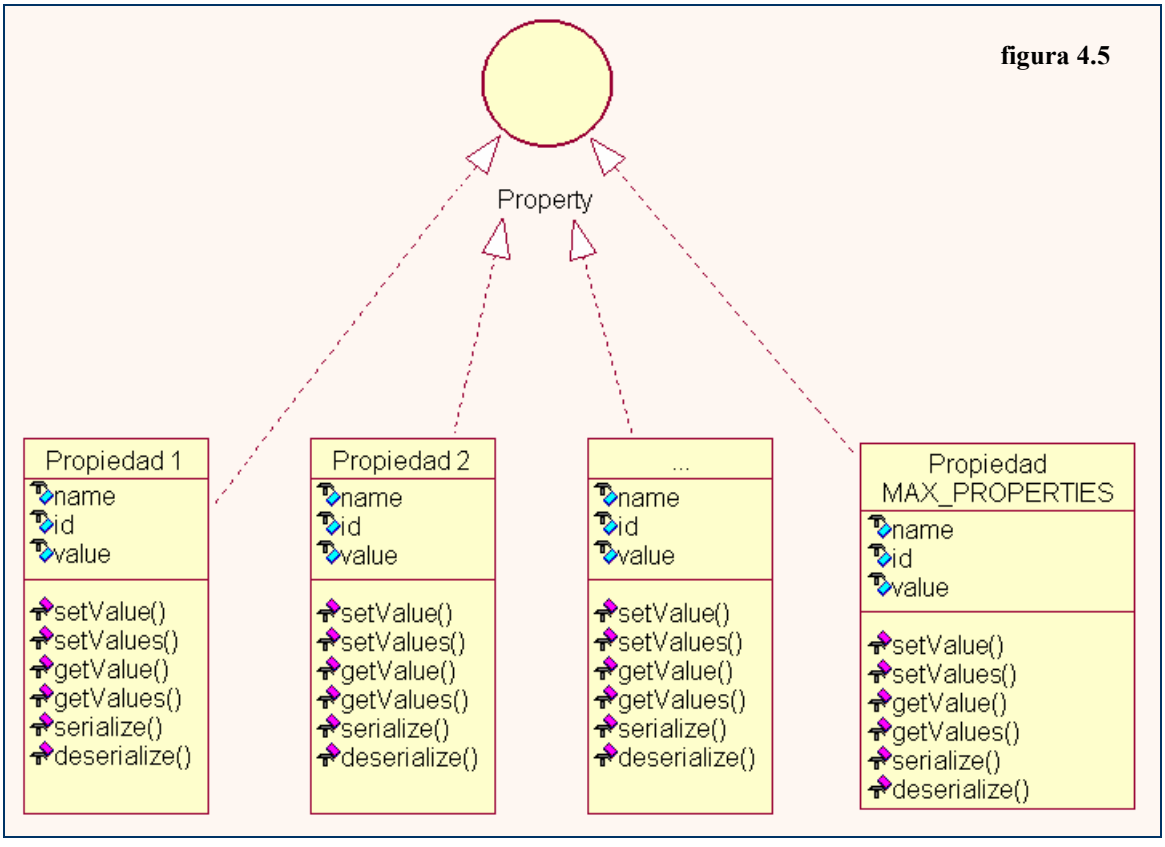


figura 4.5

4.2.2.2 La estructura *negotiation*

```
struct negotiation{
    int rate;
    int protocol;
    struct Video_Format vf;
};
```

Incluida en el fichero *grab-ng.h*. Guarda los valores de las propiedades* resultantes de la negociación. El servidor tiene una **`struct`** `negotiation` por cada cliente, mientras que el cliente sólo una. También contiene la estructura

```
struct Video_Format{
    int id;
    int width;
    int height;
    int bytesperline;
};
```

incluida en *grab-ng.h*, guarda valores relacionados con el formato de video, como un identificador **`id`** sobre formato de compresión, el ancho **`width,`** la altura **`height`** (en píxeles), y **`bytesperline`**, útil para formatos sin compresión.

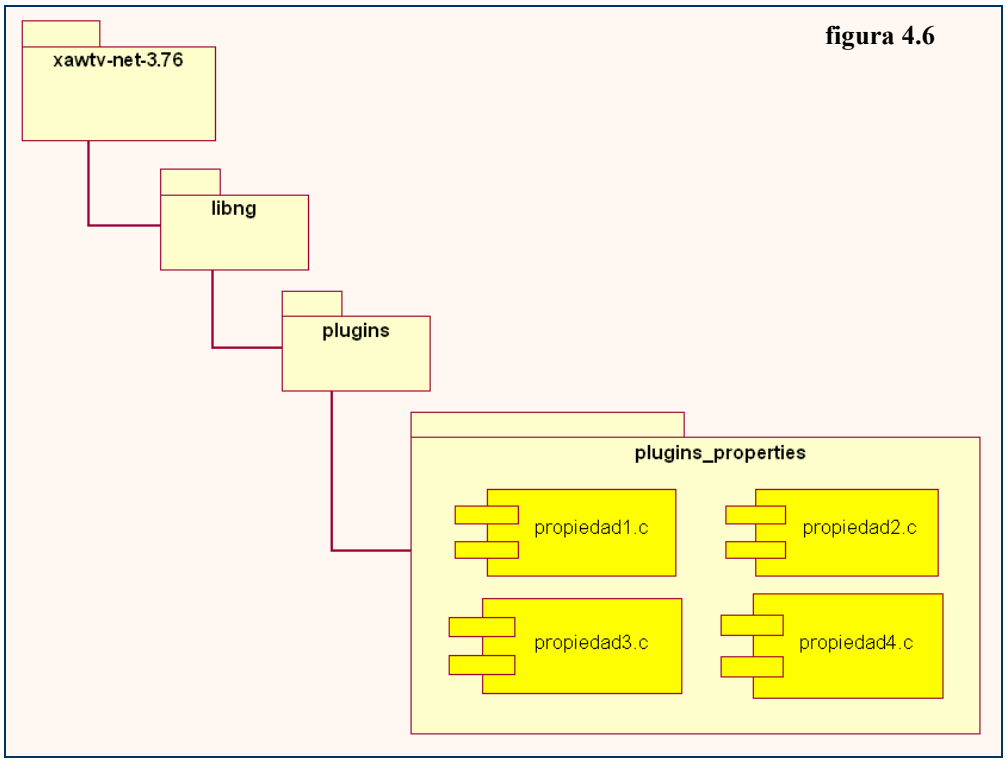
4.2.2.3 Integración de las propiedades en *xawtv*

Las propiedades se definen en ficheros independientes y son compilados como librerías dinámicas. *xawtv* y *pia* son los encargados de cargarlas y registrarlas. Sigue así la filosofía que se puede encontrar en el diseño de *xawtv* respecto a la carga de *plugins*.

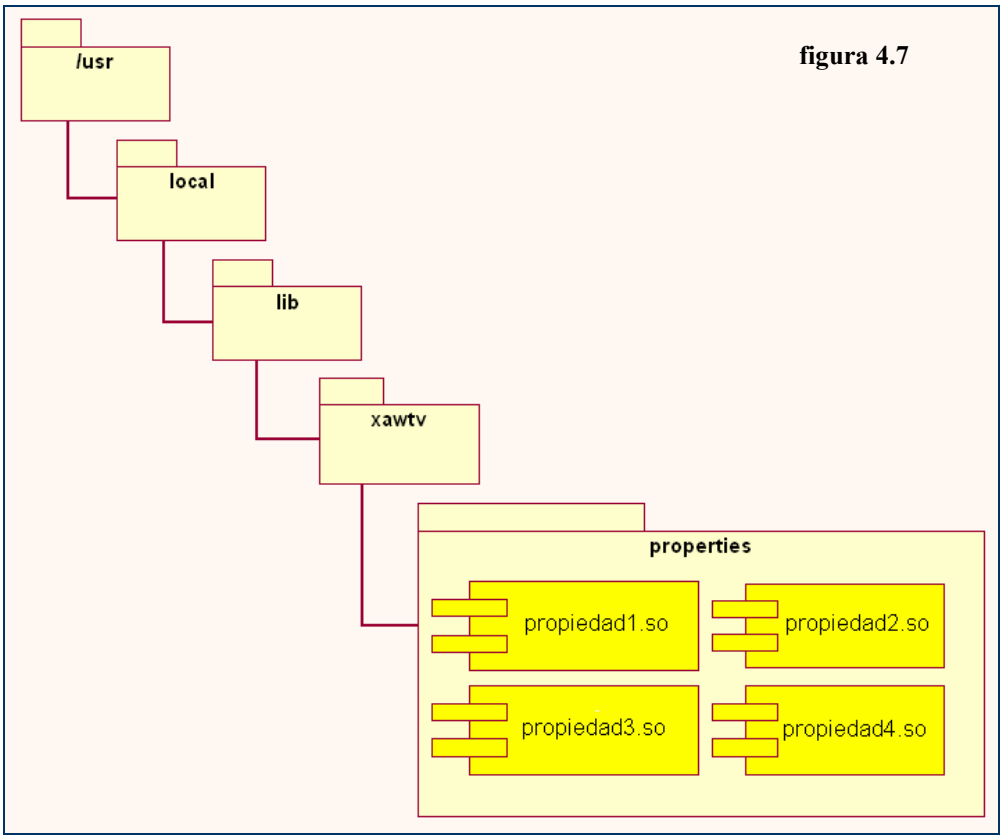
4.2.2.3.1 Ficheros de propiedades

Los ficheros fuente de las propiedades se encuentran en la carpeta *plugins_properties* dentro de *plugins* (ver figura 4.6).

* ver apartado 4.2.6, muestra las propiedades implementadas



Una vez compilados se instalan en el directorio donde se copian los *plugins* compilados de *xawtv*. Ver figura 4.7



4.2.2.3.2 Carga y registro de propiedades

Recordamos que la función

```
static int ng_plugins(char *dirname)
```

que se encuentra en *grab-ng.c* carga todos los *plugins* localizados en el directorio **`*dirname`** y los registra. Además, ahora cargará todas las propiedades y las registrará. Para registrar las propiedades llama a la función **`register_me()`** implementada en la librería dinámica correspondiente a la propiedad (ver figura 4.8, la carga de propiedades en el cliente es de forma análoga respetando la interfaz **`ng_reader`**)

```
void register_me(){
      property_register([propiedad]);
}
```

La función **`property_register()`** se encuentra en *grab-ng.c*, y se encarga de registrar una propiedad en los *arrays* de propiedades:

```
struct Property properties[MAX_PROPERTIES];
struct Property server_properties[MAX_PROPERTIES];
```

existe un *array* **`properties`** de propiedades para el cliente y otro **`server_properties`** para el servidor, ya que si el cliente y el servidor se ejecutan en la misma máquina, al no diferenciarlos los sobrescribiría. Están definidos en *grab-ng.c.* Así, cualquier *plugin* tiene acceso ellos, con lo que se puede acceder a las propiedades registradas.

Antes de registrar cualquier propiedad en estos *arrays*, han de ser inicializados. De ello se encarga la función

```
void properties_init(void)
```

que inicializa todas las posiciones de ambos *arrays*.

El acceso a las propiedades registradas se realiza mediante las funciones:

```
struct Property* get_properties(void);
struct Property* get_server_properties(void);
int properties_length(void);
int serv_properties_length(void);
```

`get_properties()` y **`get_server_properties()`** devuelven una referencia al *array* de propiedades registradas de cliente y servidor respectivamente, mientras que **`properties_length()`** y **`serv_properties_length()`** indican el número de propiedades registradas. Se llaman desde los *plugins*.

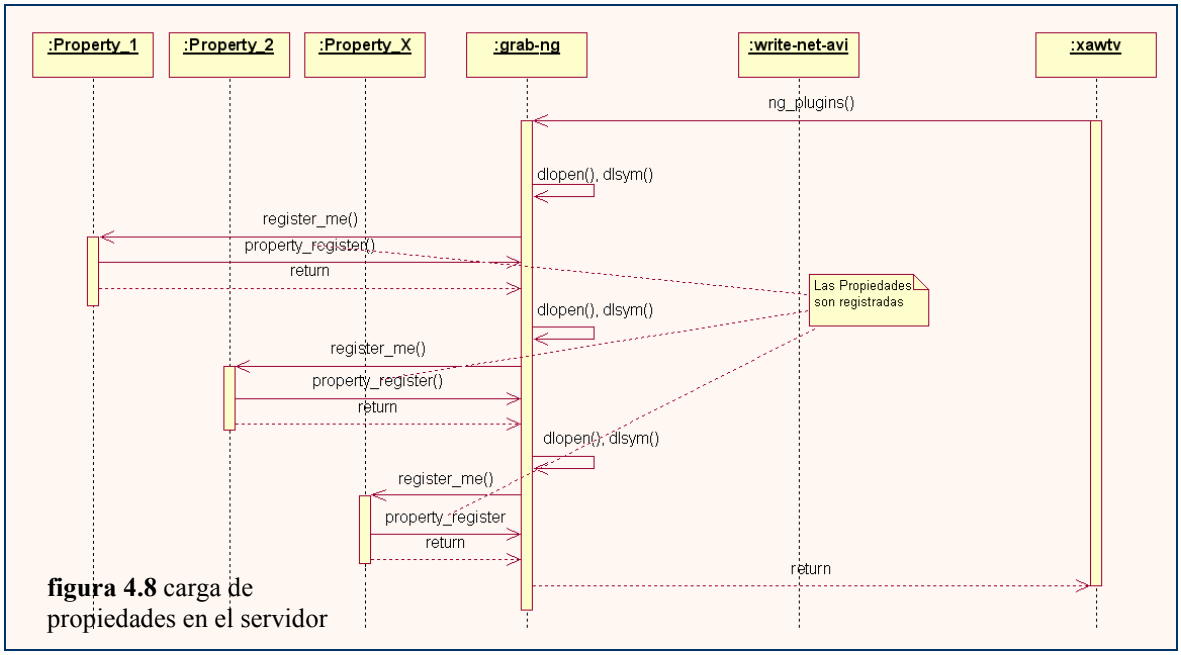


figura 4.8 carga de propiedades en el servidor

4.2.2.4 Propiedades implementadas

Se han implementado cuatro propiedades.

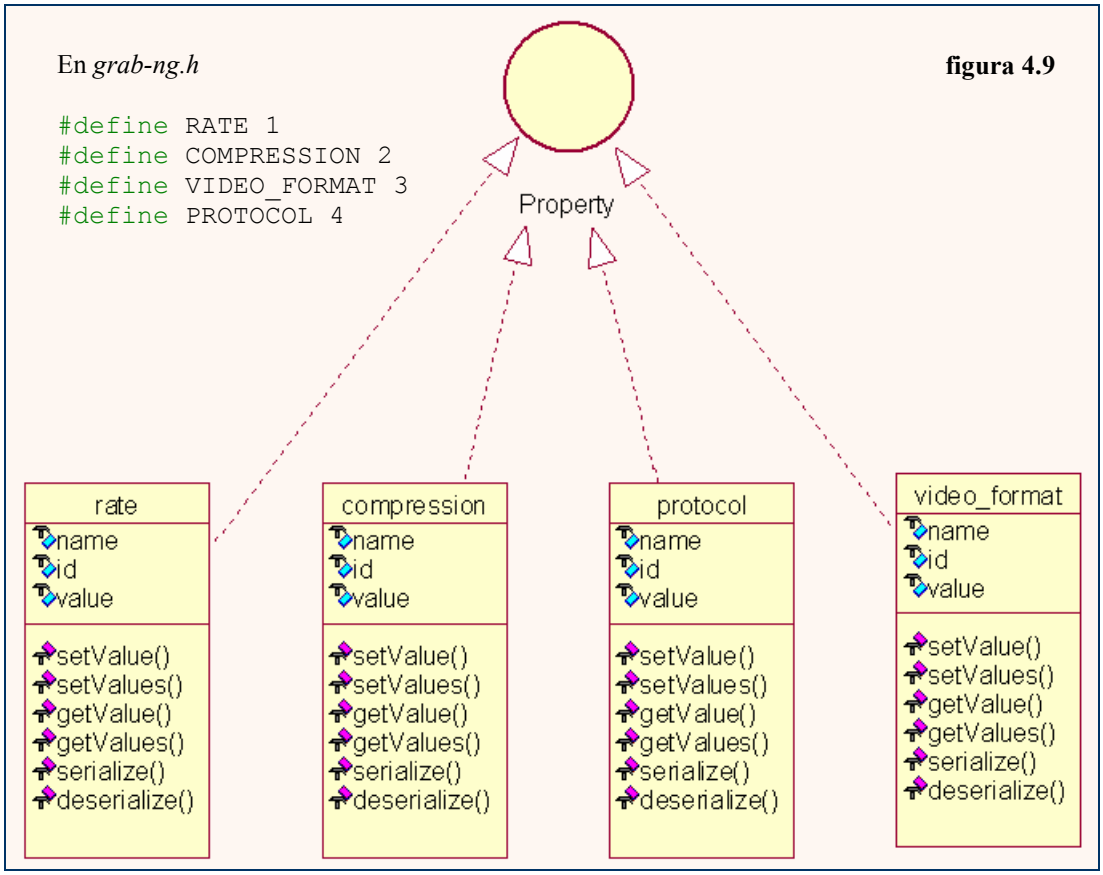


figura 4.9

4.2.2.4.1 Rate

rate define una tasa de envío de *frames* para el cliente. Un valor de *rate = 1*, indica que el cliente desea recibir todos los *frames* que envía el servidor, con el valor 2 recibiría uno de cada dos…

```
struct Property rate = {

      name:        "RATE",
      value:       {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
      id:          1,
      setValue:    setRate,
      setValues:   setRates,
      getValue:    getRate,
      getValues:   getRates,
      serialize:   serialize_rate,
      deserialize: deserialize_rate,

};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
setRate()	setValue()	Asigna valores
setRates()	setValues()	Sin implementar
getRate()	getValue()	Obtiene un valor en una posición dada
getRates()	getValues()	Sin implementar
serialize_rate()	serialize()	“Serializa” la propiedad
deserialize_rate()	deserialze()	“Deserializa” la propiedad

Las funciones de “serialización” y “deserialización” se explican en el apartado 4.2.3.2.

4.2.2.4.2 Compression

compression indica el formato de compresión de video. En *xawtv* un formato de compresión se identifica con un valor entero que se encuentra definido en *grab-ng.h.*

```
struct Property compression = {

      name:        "COMPRESSION",
      value:       {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
      id:          2,

      setValue:    setCompression,
      setValues:   setCompressions,
      getValue:    getCompression,
      getValues:   getCompressions,
      serialize:   serialize_compression,
      deserialize: deserialize_compression,

};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
setCompression()	setValue()	Asigna valores
setCompressions()	setValues()	Sin implementar
getCompression()	getValue()	Obtiene un valor en una posición dada
getCompressions()	getValues()	Sin implementar
serialize_compression()	serialize()	"Serializa" la propiedad
deserialize_compression()	deserialze()	"Deserializa" la propiedad

Las funciones de "serialización" y "deserialización" se explican en el apartado 4**.2.3.2**.

4.2.2.4.3 Video_Format

video_format informa sobre los parámetros *ancho*, *alto* y *bytesporlinea* de la imagen. Los valores se fijan de tres en tres y se obtienen también de tres en tres.

```
struct Property video_format = {

      name:          "VIDEO_FORMAT",
      value:         {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
      id:            3,

      setValue:      setVformat,
      setValues:     setVformats,
      getValue:      getVformat,
      getValues:     getVformats,
      serialize:     serialize_vformat,
      deserialize:   deserialize_vformat,

};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
setVformat()	setValue()	Sin implementar
setVformats()	setValues()	Fija los parámetros ancho, alto y bytesporlinea de tres en tres
getVformat()	getValue()	Sin implementar
getVformats()	getValues()	Devuelve una referencia a un *array* que contiene los tres parámetros
Serialize_Vformat()	serialize()	"Serializa" la propiedad
deserialize_Vformat()	deserialze()	"Deserializa" la propiedad

Las funciones de "serialización" y "deserialización" se explican en el apartado **4.2.3.2**.

4.2.2.4.4 Protocol

Identifica el protocolo deseado para la transmisión y recepción de video. Cada protocolo es identificado mediante un número entero.

En *grab-ng.h*

```
#define UDP 0
#define RTP 1
#define TAO 2
```

```
struct Property protocol = {

      name:          "PROTOCOL",
      value:         {-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
      id:            4,
      setValue:      setProtocol,
      setValues:     setProtocols,
      getValue:      getProtocol,
      getValues:     getProtocols,
      serialize:     serialize_protocol,
      deserialize:   deserialize_protocol,

};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
setProtocol()	setValue()	Asigna valores
setProtocols()	setValues()	Sin implementar
getProtocol()	getValue()	Obtiene un valor en una posición dada
getProtocols()	getValues()	Sin implementar
serialize_protocol()	serialize()	“Serializa” la propiedad
deserialize_protocol()	deserialze()	“Deserializa” la propiedad

Las funciones de “serialización” y “deserialización” se explican en el apartado **4.2.3.2**.

4.2.3 Negociación

El cliente es quien pregunta (propiedad a propiedad) al servidor si es capaz de ofrecerle los valores que ha definido el usuario. El servidor contesta con el valor que le puede ofrecer, que puede ser el que pidió el cliente u otro en el caso de que no pueda ofrecérselo.

Así, para cada propiedad se negocian sus valores, por ejemplo, el cliente puede solicitar para la propiedad *protocol* el valor 0 que pertenece a UDP. Si el servidor no puede ofrecer UDP, le ofrecerá otro protocolo distinto según el orden de preferencia de una tabla interna con valores que posee. Es más, el cliente no sólo es capaz de preguntar por un solo protocolo, sino que es capaz de enviarle al servidor por orden de preferencia los

distintos protocolos por los que está interesado. El servidor comprueba uno a uno si es capaz de ofrecérselos, si puede hacerlo lo hará e informará al cliente de ello, en caso contrario le informará del protocolo que va a usar.

Para llevar a cabo esta negociación, se ha definido un formato de datos y un procedimiento que permiten a servidor y cliente entenderse

4.2.3.1 Formato

figura 4.10

- **Fuente**: Identifica si la propiedad y sus valores pertenecen al cliente o al servidor
  - 0 Cliente
  - 1 Servidor
- **ID Propiedad**: Corresponde al identificador que tiene la propiedad
  - 1 *rate*
  - 2 *compression*
  - 3 *video_format*
  - 4 *protocol*
- **VAL 0…9**: Valores asignados a la propiedad
- **Fin:** Indica fin de la información. Este campo siempre contiene -1.

4.2.3.2 Las funciones `serialize()` y `deserialize()`

Recordemos que estas funciones son implementadas en las propiedades.

```
void* (*serialize)(int source)
```

Se encarga de crear el *array* de la figura 4.10. Rellena el campo **Fuente** con el valor de **`source`**.

Si la fuente **`source`** es el **cliente,** fija el **ID** a partir del identificador de la propiedad, y rellena los campos que contienen los **valores** con los que el usuario especificó. Son los valores que le interesan, y se guardan por orden de prioridad. El resto de valores que quedan sin especificar se rellenan con el valor -1. Para propiedades como *protocol*, *rate* y *compression* los valores se guardan de uno en uno, mientras que para *video_format*, se guardan de tres en tres, siempre dependiendo de la implementación.

Si **source** es el **servidor**, se fija el **ID** a partir del identificador de la propiedad, y rellena los campos **VAL 0…9** con el valor resultante de la negociación (la negociación siempre devuelve un valor único), los valores que quedan sin especificar, se rellenan con -1.

En todas las implementaciones realizadas de esta función se devuelve el *array* de la figura 4.10.

```
void* (*deserialize)(int *package)
```

Extrae toda la información a partir del *array* **`*package`** proporcionado por **`serialize()`**. Comprueba el origen, la propiedad y sus valores.

Si el origen es el **servidor** se devuelve el resultado de la negociación, es decir **VAL 0** en el caso de *protocol*, *rate* y *compression* y **VAL 0**, **VAL 1** y **VAL 2** en *video_format*. El valor devuelto depende de la implementación.

Si el campo fuente indica que el origen ha sido el **cliente**, para dicha propiedad se comprueban sus valores con los que son válidos según una tabla interna (definida previamente), en el caso de que no coincida ninguno, se devuelve un valor por defecto, en caso contrario se devuelve el primer valor que coincida. Al igual que antes, el valor devuelto depende de la implementación.

Ver figura 4.14.

4.2.3.3 Negociación cliente.

El cliente obtiene el *array* de las propiedades registradas

```
propert = get_properties();
```

las guarda en ***propert** y llama a la función

```
void set_client_properties(struct Property *propert)
```

definida en *utils.c* (fichero que se encuentra en el directorio */plugins* y una vez compilado se guarda en la librería *libng.a*, ver figura 4.11) ya que no es una función que tenga relación directa con *xawtv*. Se encarga de rellenar el *array* de propiedades con sus respectivos valores (a través de **`setValue()`**) a partir del fichero *propiedades.dat* que sigue la estructura:

```
#1 RATE
<valor1>
...
<valor10>
#2 VIDEO_FORMAT
<valor1>
<valor2>
<valor3>
#3 COMPRESSION
<valor1>
...
<valor10>
#4 PROTOCOL
<valor>
```

```
#1 RATE
2
5
9
#2 VIDEO_FORMAT
264 352 0
288 356 1052
#3 COMPRESSION
MJPEG
#4 PROTOCOL
RTP
```

Cada propiedad definida en el fichero va precedida del símbolo # y un identificador. Es el usuario quien se encarga de definir los valores de cada una. El programa acepta hasta diez posible valores que se introducen por orden de preferencia.

Por ejemplo, en el cuadro de la derecha se ve como el cliente desea un valor *rate* = *2*, pero si ese valor no puede ser ofrecido por el servidor, le interesaría el valor *5*, o en su defecto el *9*. Es probable que no se acepte ningún valor de los indicados, en ese caso el servidor establecería un valor por defecto. Ver figura 4.12.

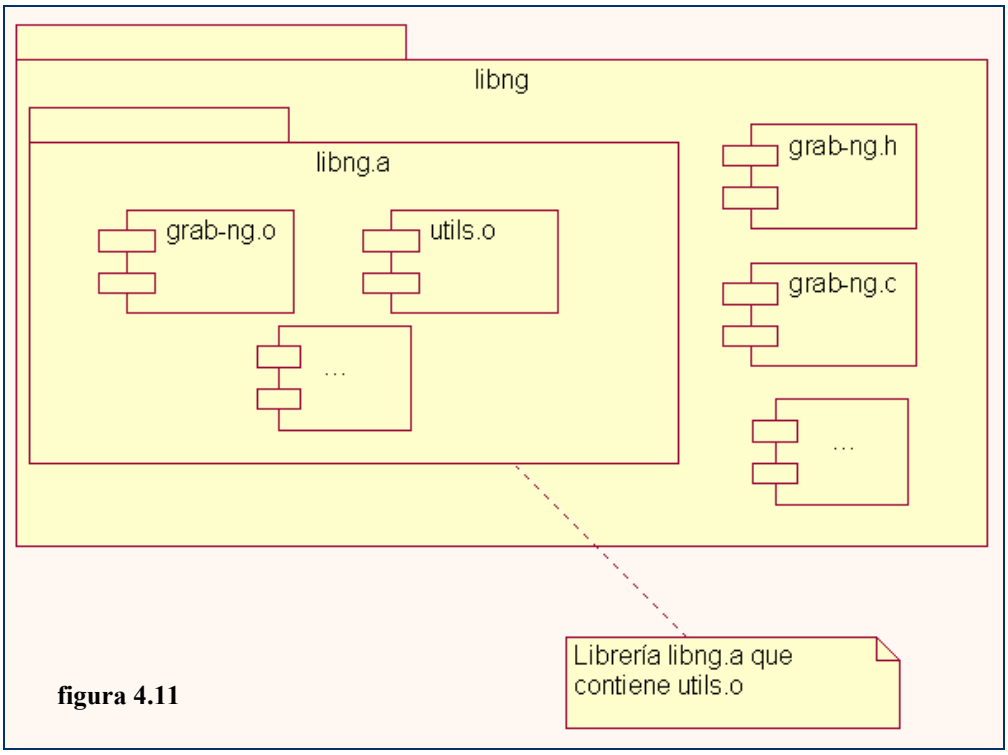


figura 4.11

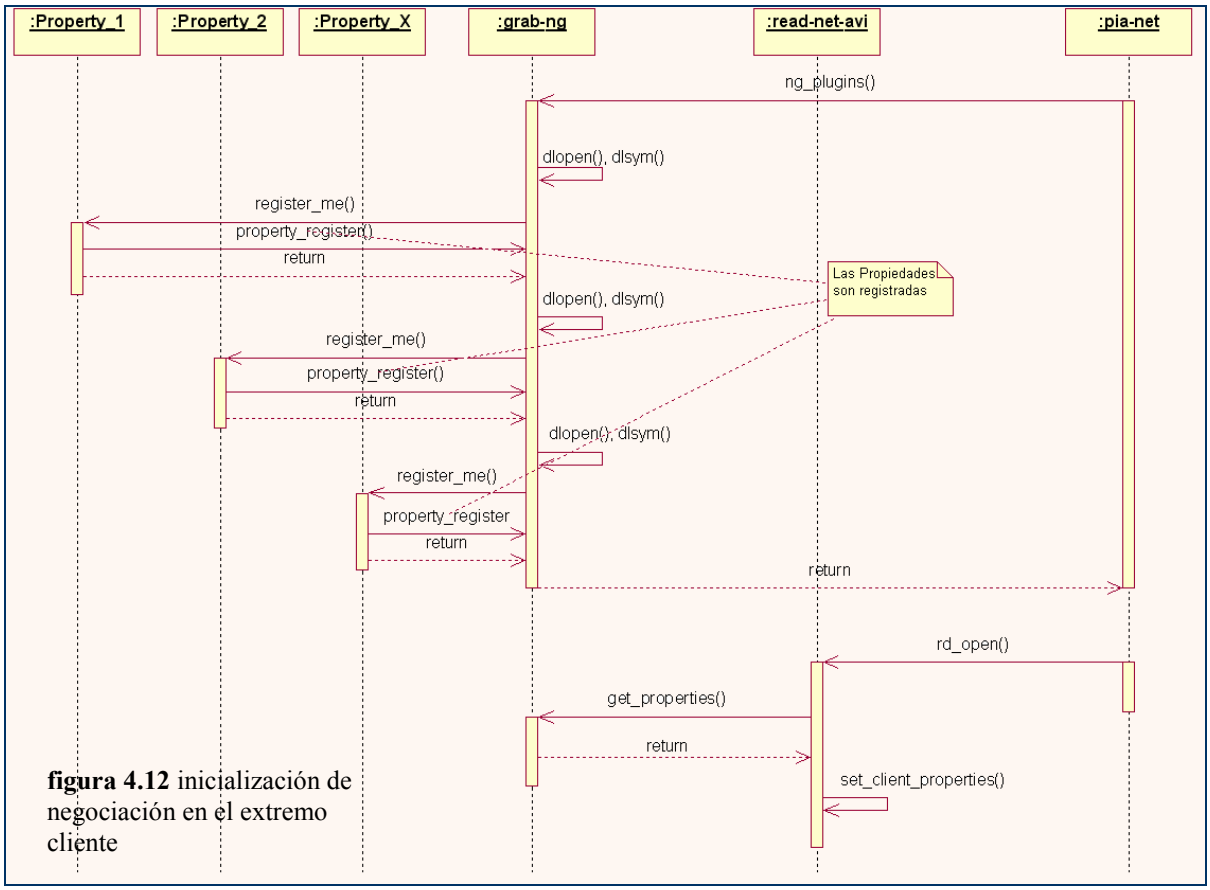


figura 4.12 inicialización de negociación en el extremo cliente

```
struct negotiation negotiation_client_start(char *IP,
                                            struct Property *propert)
```

Una vez que ya se dispone de las propiedades a negociar, el cliente inicia una negociación con el servidor, a través de la función **`negotiation_client_start()`**. Se contactará con el servidor a través de su dirección **`IP`** y se negociarán todas las propiedades del *array* ***`propert`**. Como resultado, se guardan los valores en una estructura de tipo **`struct`** `negotiation`. La conexión TCP con el servidor se cierra tras la negociación.

```
struct negotiation set_manual_negotiation (struct Property *propert)
```

Si no se desea seguir este protocolo de negociación, se llama a la función **`set_manual_negotiation()`** que puede ser implementada según la necesidad del usuario. La implementación por defecto no realiza conexión TCP, tan solo rellena la estructura **`negotiation`** con los valores establecidos por el usuario (sin negociar). Ver figura 4.13.

La forma en la que un usuario puede anular la negociación TCP, consiste en indicar en el fichero *propiedades.dat* la sentencia **NOTCP**.

```
#4 PROTOCOL
NOTCP
TAO
```

4.2.3.4 Negociación servidor. Varios clientes.

El servidor obtiene el *array* de las propiedades registradas,

```
propert = get_server_properties();
```

las guarda en ***propert** y llama a la función

```
void set_server_properties(struct Property *propert,
                           struct Video_Format video_format)
```

definida en *utils.c*. Esta función se encarga de rellenar el *array* de propiedades ***propert** con los valores que el servidor es capaz de servir (usando la función **setValue()**). Se fijan por orden de preferencia y en caso de que el cliente solicite un valor de una propiedad que el servidor no sea capaz de servir, el servidor devolverá el primer valor de su tabla (ese será el valor por defecto). Ver figura 4.13

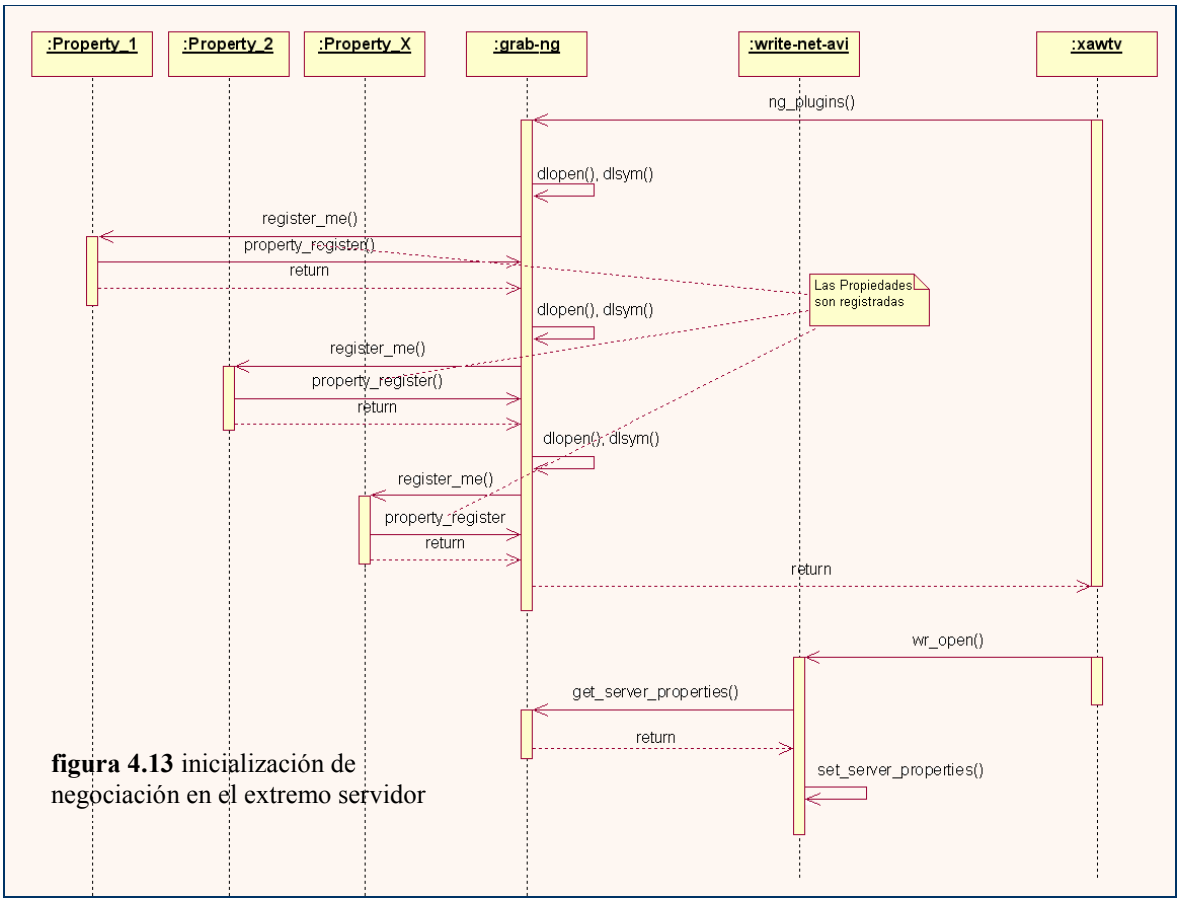


figura 4.13 inicialización de negociación en el extremo servidor

Una vez fijadas las propiedades, el servidor lanza el *thread* **connection_thread** en **avi_net_open()** que queda a la espera de conexiones,

```
pthread_create (&connection_thread, NULL, connection_listener, NULL)
```

lo crea con la función:

```
void *connection_listener()
{
      struct sockaddr_in cli_ad;
      int i,err;

      sfd = TCP_server_start();

      for(;;)    //Bucle infinito de escucha de conexiones TCP.
      {

            err=negotiation_server_start(sfd,propert,negotiation,&cli_ad);
            if (err == -1) printf("Negociaciación errónea\n");

            for (i=0;i<num_ctrls;i++)
            {
                  if (controller[i].id == negotiation->protocol)
                              controller[i].add_Client(cli_ad,*negotiation);
            }

      }
}
```

que se encarga de llamar la primera vez que se arranca el *thread* a

```
int TCP_server_start()
```

incluida en *utils.c*. Crea *socket* del servidor, llama a la función **bind()** y se encarga de escuchar peticiones a través de **listen()**. Devuelve el descriptor asociado al *socket*.

```
int negotiation_server_start(int sfd, struct Property *propert,
struct negotiation *negotiation, struct sockaddr_in *cli_ad);
```

El servidor entra ahora en un bucle infinito con la llamada a la función **negotiation_server_start()** que se ejecuta una vez para cada cliente y es la encargada de llevar a cabo la negociación. Devuelve *0* si la negociación ha sido satisfactoria, y *-1* en caso de que se haya producido algún error. Es la encargada de captar conexiones, ya que hasta que no llegue ningún cliente, el servidor se queda esperando en la llamada a **accept()**. Establece un retardo de 6 segundos entre dos conexiones consecutivas (antes del **accept()**) para que no intenten conectar dos clientes de forma simultánea y se produzcan problemas de inicialización, lo que daría lugar a errores en el envío de datos, imposible desconexión de clientes, etc. Ver figura 4.15.

`*propert` es el array de propiedades registradas con los valores asignados en la función **`set_server_properties()`**. El resultado de la negociación se guarda en la estructura **`*negotiation`**, y la dirección del cliente en **`*cli_ad`**. Esta dirección se usa para añadir un cliente en la tabla interna de clientes mediante la función **`add_Client()`** (ver apartado 5). Así, la conexión TCP además de negociar es la que capta clientes cuyo protocolo de conexión se base en TCP. El protocolo de transmisión de video puede ser cualquier otro (según la implementación), pues el cliente cierra la conexión TCP tras la negociación.

Cuando se cierra el servidor con la llamada a **`avi_net_close()`**,

```
static int
avi_net_close(void *handle)
{
      close(sfd);
      pthread_kill(connection_thread,SIGKILL);
      return 0;
}
```

se cierra el *socket* TCP, y "mata" al *thread* **`connection_thread`** con la función **`pthread_kill()`**

4.2.3.5 Negociación TCP. Diagramas y ejemplos.

En los dos apartados anteriores se ha visto la negociación desde el punto de vista cliente y servidor. Mediante diagramas y ejemplos se explicará cómo es la comunicación entre ambos durante la negociación.

Cuando un cliente no desea negociación, lo hace mediante el parámetro **NOTCP** (ver figura 4.14).

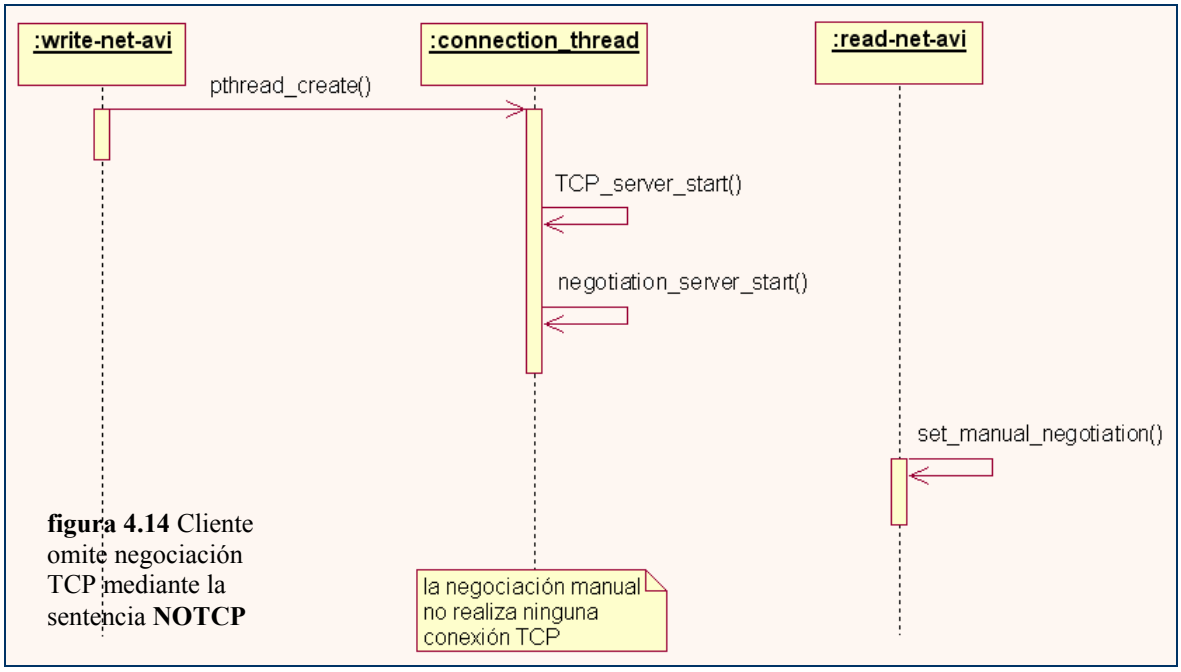


figura 4.14 Cliente omite negociación TCP mediante la sentencia **NOTCP**

En el diagrama de secuencia de la figura 4.15 se representa el proceso de negociación de una forma genérica. Ahora se puede observar el empleo de las funciones **`serialize()`** y **`deserialize()`** de una forma más clara. El cliente "serializa" una propiedad para poder enviarla por la red. Cuando la recibe el servidor llama a **`deserialize()`** que compara los valores que ha pedido el cliente de esa propiedad, con los que es capaz de servir. En el caso de que alguno coincida, la función devuelve ese valor, en caso contrario devuelve un valor por defecto. Ese valor se guarda, y la función **`serialize()`** lo intoduce en la trama de negociación y se lo envía al cliente. Este proceso se repite para cada propiedad. Cuando no se desean negociar más propiedades se envía la trama de finalización.

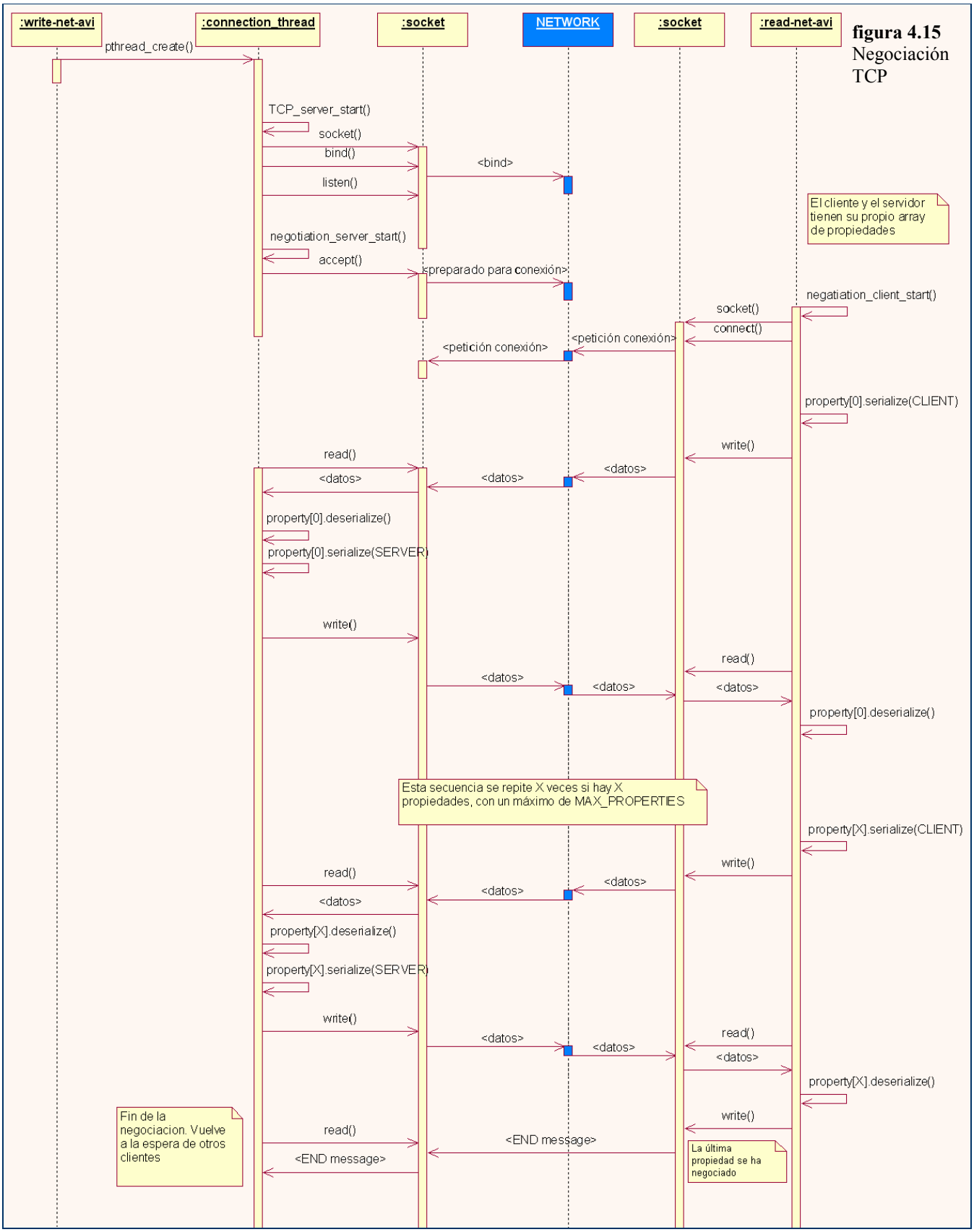
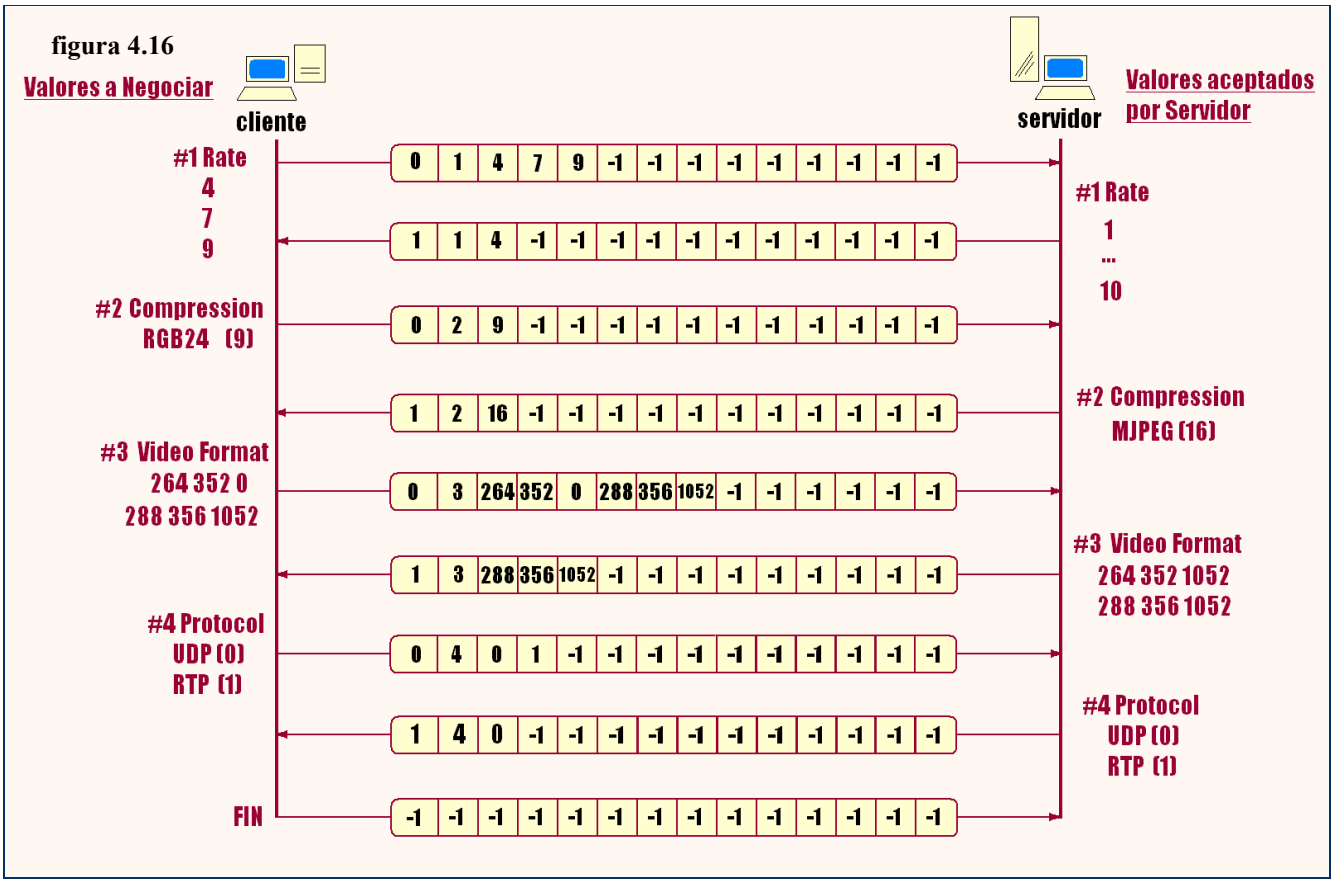


figura 4.15 Negociación TCP

De una forma más concreta, y centrándonos exclusivamente en el formato de la trama, observemos el diagrama de negociación de la figura 4.16.



El cliente tiene cuatro propiedades a negociar, con sus respectivos valores. En la parte derecha se observa la tabla de valores que el servidor es capaz de ofrecer para cada propiedad. En todas las tramas cuyo origen es el cliente, el campo **ID** tiene valor *0*, y *1* en el caso del servidor. El siguiente campo corresponde al identificador de la propiedad, y los siguientes excepto el último son sus valores asociados (el último campo siempre es *-1*).

Veamos el caso de *rate*, el cliente solicita los valores *4*, *7* y *9* por orden de preferencia. El servidor que es capaz de ofrecer valores desde *1* hasta *10*, le contesta indicando que le servirá *rate* con valor *4*.

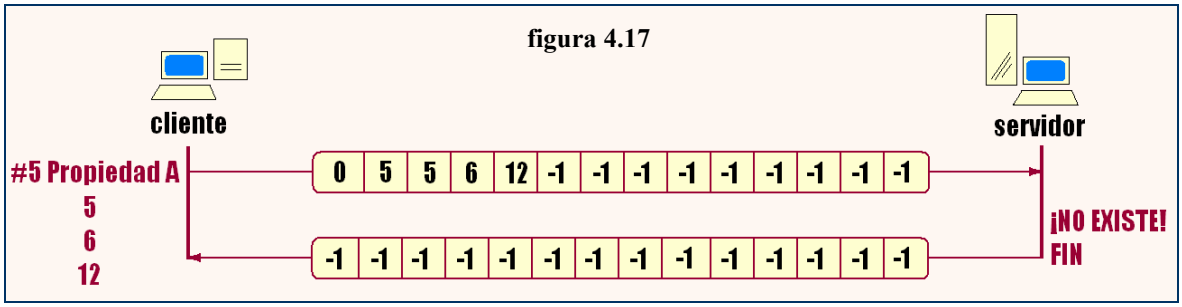
Ocurre lo contrario cuando se intenta negociar *compression*, el cliente solicita un formato de compresión (cuyo identificador es el *9*) que el servidor no es capaz de ofrecer (sólo acepta el formato con identificador *16*). El servidor informa que va a ofrecerle el formato *16*.

En la negociación de la propiedad *video_format* el servidor sólo puede ofrecerle el segundo formato de video que solicitó el cliente, y para *protocol* ocurre lo mismo que en el caso de *rate*.

Cuando el cliente no desea negociar más propiedades, envía la trama de finalización, con todos los campos a valor *-1*.

Puede darse el caso en el que se realice una petición (recordemos que es siempre el cliente el que pregunta al servidor sobre las propiedades) sobre una propiedad que no ha sido registrada en el servidor. Entonces el servidor envía al cliente la trama de fin de negociación. El cliente se desconectará. Figura 4.17

figura 4.17



5 Controladores

5.1 Introducción

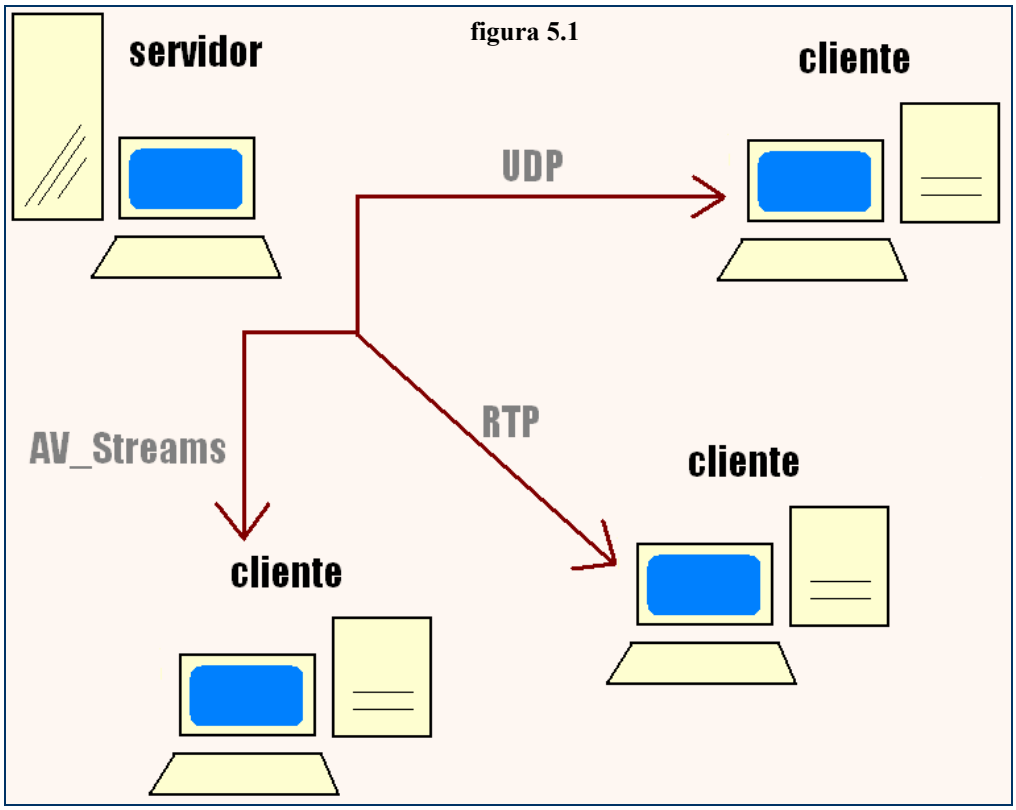
Existen distintos protocolos que pueden usarse para la transmisión de video, algunos específicos como son RTP, AVStreams de Corba… otros aunque no específicos sí que pueden ser adecuados por sus características, como por ejemplo UDP.

Es interesante la implementación de transmisión de video en *xawtv* empleando un protocolo de transmisión, pero lo es aún más un programa capaz de enviar video usando varios protocolos al mismo tiempo, y lo que es más, sin tener que modificar ni una sola línea de código.

Basándonos en la filosofía de *plugins* de *xawtv,* se ha ideado una interfaz llamada *controller* que puede ser implementada con el uso de diferentes protocolos. Cada controlador implementado permitirá a *xawtv* enviar vídeo y recibirlo. Tanto cliente como servidor deben cargar ese controlador (que es una librería dinámica al igual que los *plugins* y las propiedades) y hacer uso de sus interfaces para el establecimiento de la conexión, envío, recepción de vídeo y desconexión.

El servidor es capaz de enviar usando todos los protocolos implementados en los controladores al mismo tiempo. A mayor número de controladores implementados, más protocolos para la transmisión de video se soportan. Un cliente puede recibir video usando el protocolo UDP, y otro cliente puede recibirlo usando RTP, o incluso CORBA, y todo ello a partir de un mismo servidor, y sin necesidad de modificar código en el cliente ni en el servidor cada vez que se añada una nueva implementación. Ver figura 5.1

figura 5.1



5.2 La estructura *Controller*

```
#define MAX_CONTROLLERS 3
#define MAX_CONNECTIONS 5

struct Controller{

	char name[20];
	int id;
	int (*init)(void);
	int (*add_Client)(struct sockaddr_in cli_ad, struct negotiation ng);
	int (*remove_Client)(struct sockaddr_in cli_ad);
	int (*connect)(void *handle, struct negotiation ng);
	int (*send)(void *handle, struct ng_video_buf *buf);
	struct ng_video_buf* (*recv)(void *handle);
	int (*close)(void *handle, int sc);
};
```

Incluida en el fichero *grab-ng.h*. **`Controller`** Permite especificar un nombre de controlador **`name`** y un identificador **`id`**. La implementación de cada controlador debe permitir una inicialización del controlador con la función **`(*init)()`**. La interfaz **`Controller`** tiene funciones relacionadas con el servidor y otras con el cliente.

Un **servidor** debe añadir clientes con la función **`(*add_Client)()`**, borrarlos con **`(*remove_Client)()`**, enviarles video con **`(*send)()`** y cerrar la conexión a través de la función **`(*close)()`**.

Por otra parte, el **cliente** debe ser capaz de conectar al servidor con la función **`(*connect)()`**, recibir video con **`(*recv)()`** y cerrar la conexión llamando a **`(*close)()`**. Ver figura 5.2.

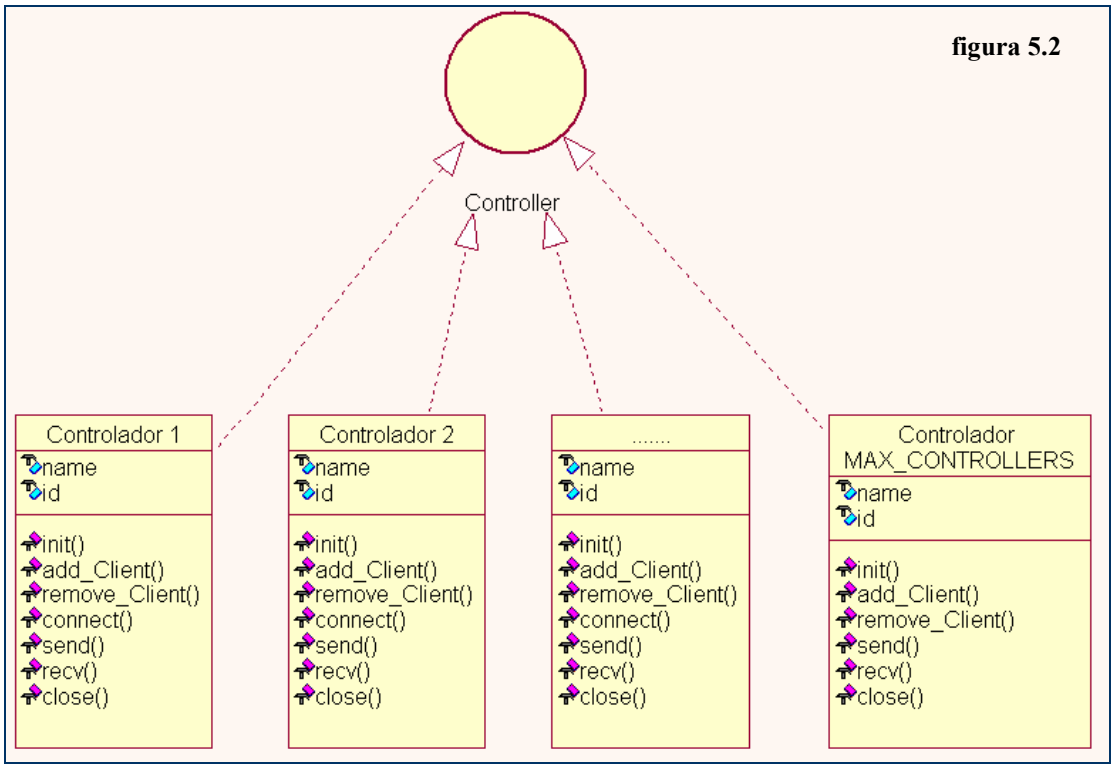


figura 5.2

5.3 Controladores y *xawtv*

5.3.1 Integración de los controladores

Los controladores se definen en ficheros independientes y son compilados como librerías dinámicas. *xawtv* y *pia* son los encargados de cargarlos y registrarlos. Se sigue así la filosofía de diseño de *xawtv* respecto a la carga de *plugins*.

5.3.2 Ficheros de controlador

Los ficheros fuente de los controladores se encuentran en la carpeta *controllers* dentro de *plugins* (ver figura 5.3).

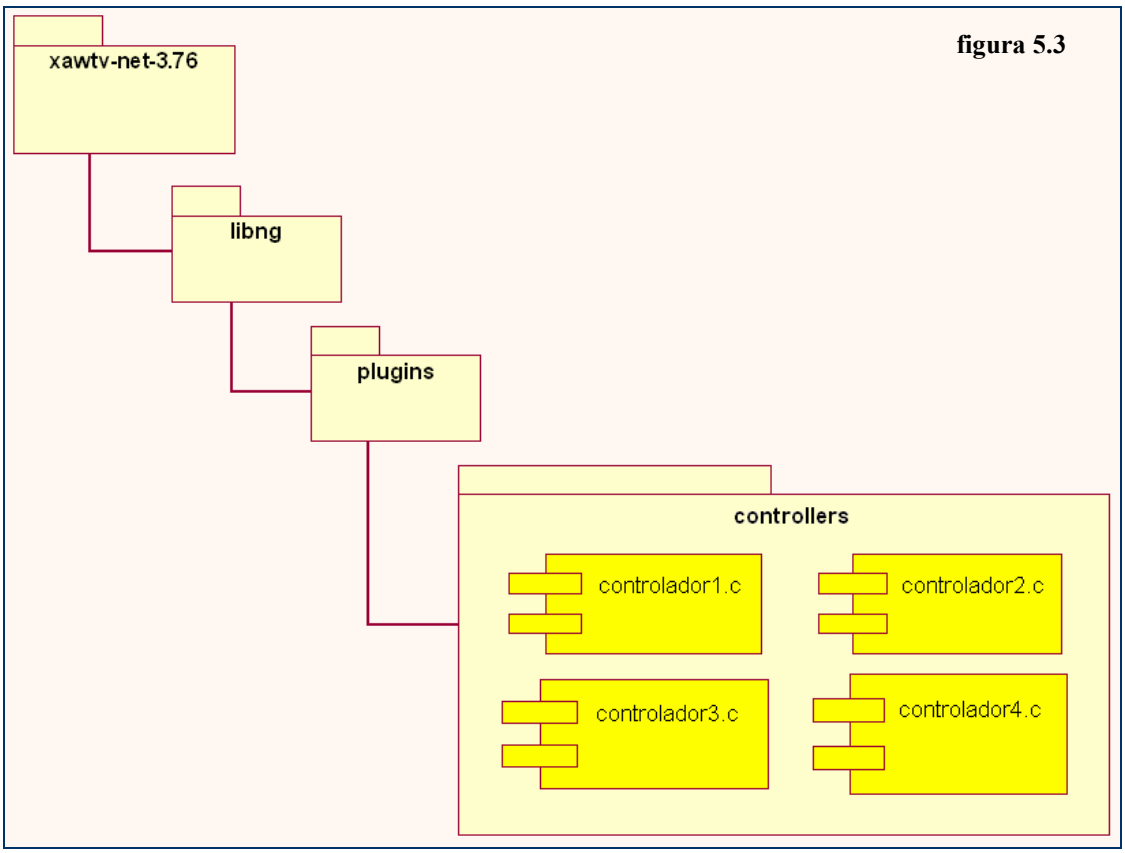


figura 5.3

Una vez compilados se instalan en el directorio donde se copian los *plugins* compilados de *xawtv*. Ver figura 5.4

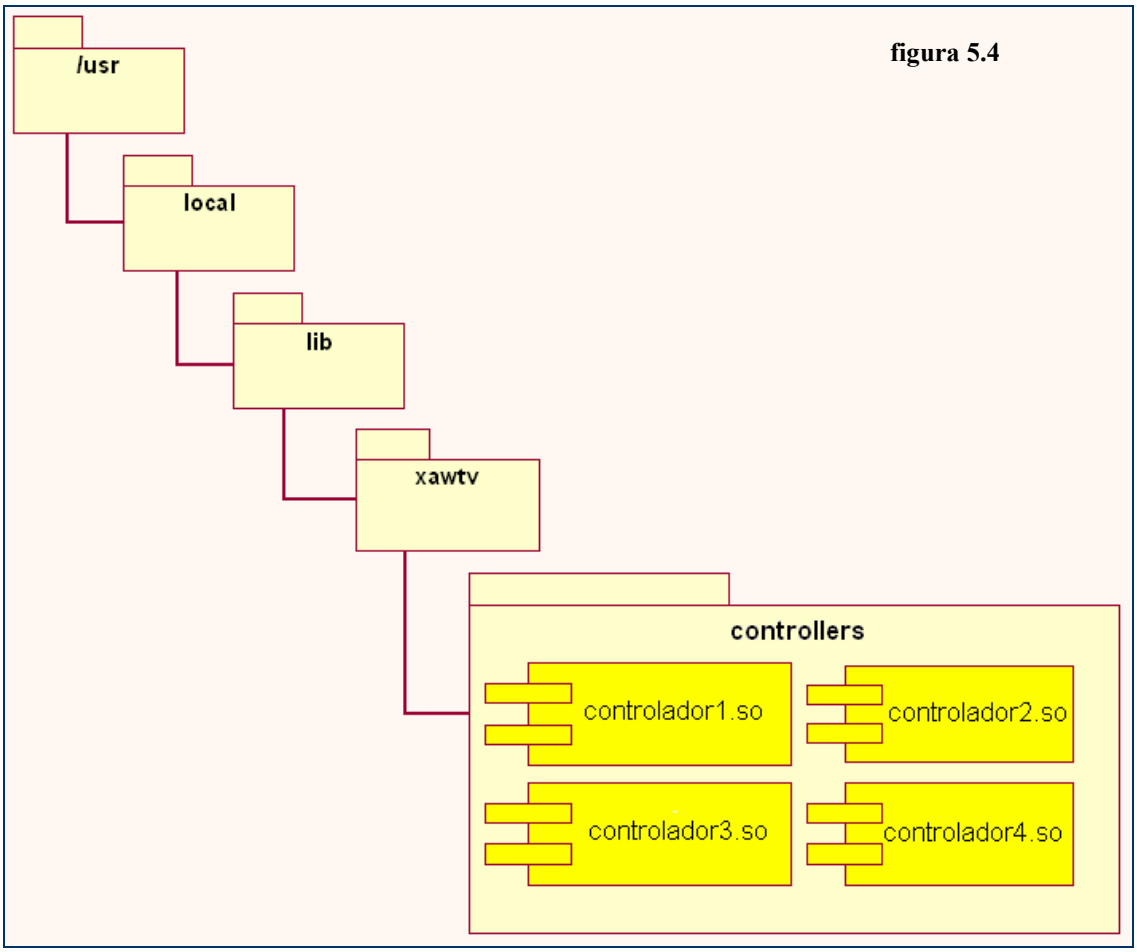


figura 5.4

5.3.3 Carga y registro de controladores

La carga es similar a la de los *plugins.* La función

```
static int ng_plugins(char *dirname)
```

que se encuentra en *grab-ng.c* carga todos los *plugins* localizados en el directorio **`*dirname`** y los registra. Ahora, además de cargar las propiedades y registrarlas, hará lo mismo con los controladores. Para registrar los controladores llama a la función **`register_me()`** implementada en la librería dinámica correspondiente al controlador (ver figura 5.5, la carga de propiedades en el servidor es de forma análoga respetando la interfaz **`ng_writer`**)

```
void register_me(){
      controller_register([controlador]);
}
```

La función **`controller_register()`** se encuentra en *grab-ng.c*, y se encarga de registrar un controlador en un *array* de controladores:

```
struct Controller controllers[MAX_CONTROLLERS];
```

existe un único *array* **`controllers`** de controladores. En el caso de las propiedades debía definirse uno para el cliente y otro para el servidor, aquí no es necesario, pues las funciones que se implementan en cada controlador no son compartidas. Está definido en *grab-ng.c.* Así, cualquier *plugin* tiene acceso al *array*, con lo que se puede acceder a los controladores registrados.

Antes de registrar cualquier controlador en el *array*, ha de ser inicializado, de ello se encarga la función

```
void controllers_init(void)
```

que inicializa todas las posiciones del *array*.

El acceso a los controladores registrados se realiza mediante funciones

```
struct Property* get_controllers(void);
int controllers_length(void);
```

`get_controllers()` devuelve una referencia al *array* de controladores registrados, mientras que **`controllers_length()`** indica el número de controladores registrados. Se llaman desde los *plugins*.

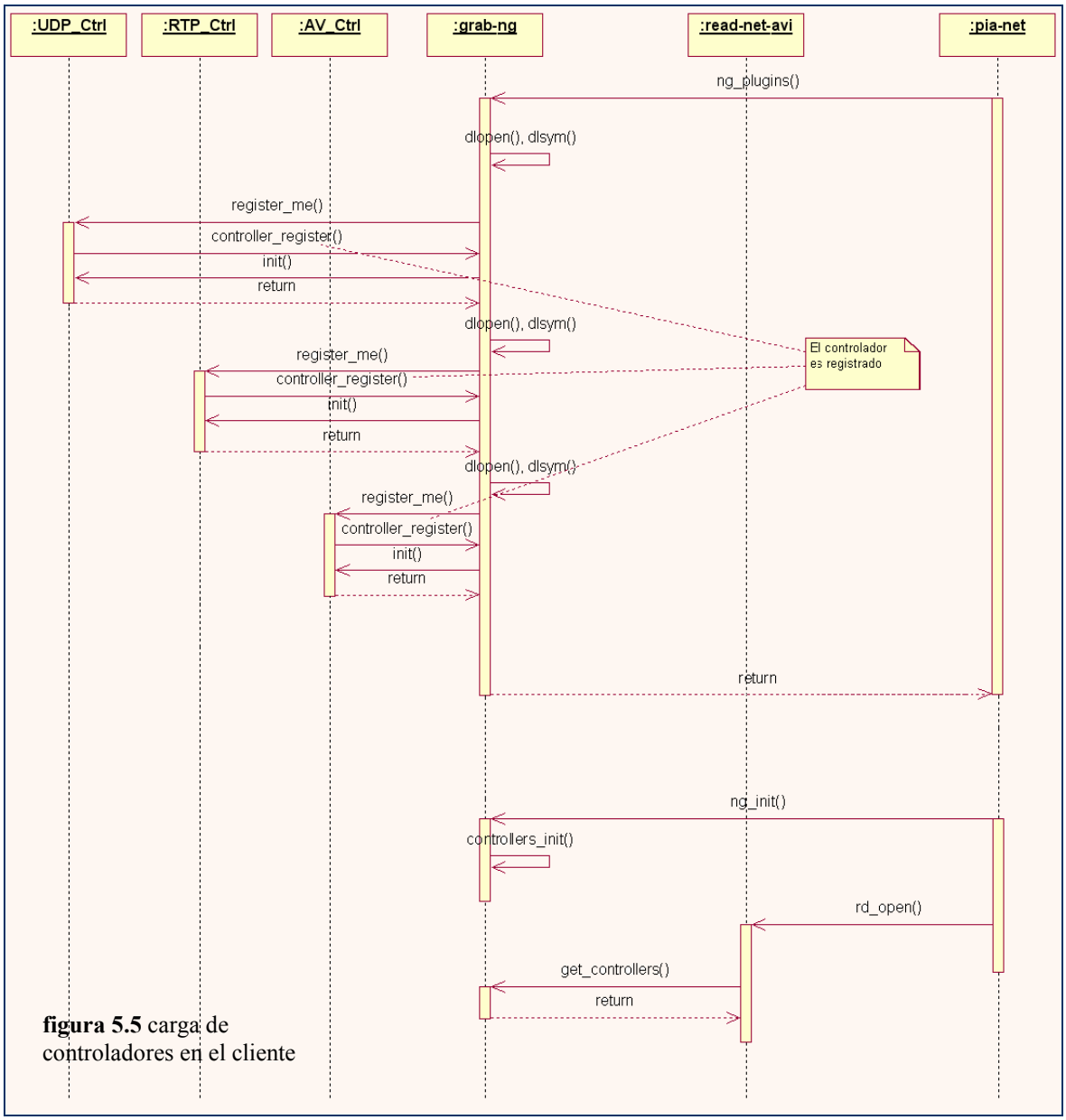


figura 5.5 carga de controladores en el cliente

5.3.4 Incorporación a *read-net-avi.c*

El *plugin* **`avi_net_reader`** (cliente) carga todos los controladores y se encarga de llamar a sus funciones. Omitiendo el código no relacionado con los controladores

```
struct Controller *controller;
```

```
static void* avi_net_open(char *arg, int *vfmt, int vn)
{
	struct avi_handle	*h;
	controller = (struct Controller*)
		malloc(MAX_CONTROLLERS*(sizeof(struct Controller)));
	controller = get_controllers();
	num_ctrls = controllers_length();

	for (i=0;i<num_ctrls;i++)
	{
		if (controller[i].id == negotiation.protocol){
			controller[i].connect(h,negotiation);
		}
	}
	return h;
}
```

en la función **avi_net_open()** se obtienen todos los controladores registrados mediante la función **get_controllers()** y el número de éstos con **controllers_length()**. Se recorre el *array* que guarda los controladores, buscando el relacionado con el protocolo que se negoció. Cuando se haya localizado, se llama a su función **connect()**. Se devuelve a *xawtv* una estructura de tipo **avi_handle**, que posteriormente *xawtv* se la pasará a **avi_net_vdata()** y **avi_net_close()** como parámetro.

La estructura **avi_handle** se define tanto en el *plugin* como en la implementación de cada controlador.

```
struct avi_handle {
    int                 fd;
    char                IP[15];
    struct iovec        *vec;
    long long ts;
    struct ng_video_fmt vfmt;
    struct ng_audio_fmt afmt;
    int                 frames;
};
```

Contiene un descriptor **fd**, permite guardar una dirección **IP**, ofrece información sobre la estampa de tiempo de un *frame*, así como de fomato de video **vfmt** y audio **afmt**. Además, guarda el número de *frame* en la variable **frames**.

```
static struct ng_video_buf* avi_net_vdata(void *handle, int drop)
{

    struct avi_handle *h = handle;
    int i;
      for (i=0;i<num_ctrls;i++)
      {
            if (controller[i].id == negotiation.protocol)
            {
                  return controller[i].recv(h);
            }
      }
}
```

avi_net_vdata() llama a la función **recv()** del controlador cuyo protocolo se negoció, y devuelve el video recibido.

```
static int avi_net_close(void *handle)
{
    struct avi_handle *h = handle;
    int i;

    for (i=0;i<num_ctrls;i++)
    {
            if (controller[i].id == negotiation.protocol)
            {
                  controller[i].close(h,CLIENT);
                  return 0;
            }
    }
    free(h);
    return 0;
}
```

En **avi_net_close()** se llama a la función **close()** del controlador negociado.

5.3.5 Incorporación a *write-net-avi.c*

El *plugin* **avi_net_writer** (servidor) carga todos los controladores y llama a sus funciones. Omitiendo el código no relacionado con los controladores

```
struct Controller *controller;
```

```
static void*
avi_net_open(char *filename, char *dummy,
       struct ng_video_fmt *video, const void *priv_video, int fps,
       struct ng_audio_fmt *audio, const void *priv_audio)
{
      controller = (struct Controller*)
      malloc(MAX_CONTROLLERS*(sizeof(struct Controller)));
      controller = get_controllers();
      num_ctrls = controllers_length();
      return h;
}
```

en la función **avi_net_open()** se obtienen todos los controladores registrados a través la función **get_controllers()** (al igual que **avi_net_reader**) y el número de éstos con **controllers_length()** se guarda en la variable **num_ctrls**. Se devuelve a *xawtv* una estructura de tipo **avi_handle** (definida en el apartado 5.3.4), que posteriormente *xawtv* se la pasará a **avi_net_video()** y **avi_net_close()** como parámetro.

```
static int
avi_net_video(void *handle, struct ng_video_buf *buf)
{
      int i;
      for (i=0;i<num_ctrls;i++){
            controller[i].send(handle,buf);
      }
            return 0;
}
```

avi_net_video() llama a la función **send()** de todos los controladores registrados, le pasa como parámetros el manejador ***handle** y **buf** que contiene el video capturado.

```
static int
avi_net_close(void *handle)
{
      int i;
      struct avi_handle *h = handle;

      for (i=0;i<num_ctrls;i++) controller[i].close(h,SERVER);
      free(h);
      return 0;
}
```

En **avi_net_close()** (obviando líneas de código no relacionadas con los controladores) se llama a la función **close()** de todos los controladores registrados, indicando que quien llama la función es el servidor.

Como se vio en el apartado 4.2.3.4, el *thread* **connection_thread** es el encargado de captar clientes. Los añade al controlador mediante la llamada a función **add_Client()**

```
void *connection_listener()
{
	struct sockaddr_in cli_ad;
	int i,err;

	sfd = TCP_server_start();

	for(;;)	//Bucle infinito de escucha de conexiones TCP.
	{

		err=negotiation_server_start(sfd,propert,negotiation,&cli_ad);
		if (err == -1) printf("Negociaciación errónea\n");

		for (i=0;i<num_ctrls;i++)
		{
			if (controller[i].id == negotiation->protocol)
					controller[i].add_Client(cli_ad,*negotiation);
		}

	}
}
```

donde **cli_ad** es la dirección IP del cliente, y ***negotiation** contiene los valores de las propiedades resultantes de la negociación con ese cliente.

6. Controlador para transmisión UDP

6.1 Introducción a UDP

El *User Datagram Protocol* (UDP) es un protocolo de la capa de transporte definido por el departamento de defensa de los Estados Unidos, para usarlo junto al protocolo IP de la capa de red.

El protocolo UDP **[13]** se encuentra definido en *rfc768 - User Datagram Protocol*, se describen a continuación las características principales de este protocolo.

6.1.1 Servicio

Las características de UDP:

- Dirección UDP = (dirección IP, puerto UDP). Puerto UDP es un número entre 1 y 65535.
- Permite enviar **mensajes de tamaño limitado** (por ejemplo 8Kbytes en la implementación de UDP en los sistemas operativos de Sun, o 64Kbytes en *linux*).
- **No orientado a conexión**: Los datos y la cabecera UDP a enviar se introducen en un datagrama IP. Cada mensaje es tratado independientemente.
- Entrega **no confiable**: El receptor UDP descarta los datagramas que llegan con errores, pero no pide retransmisión.
- Utilidad: Aquellas aplicaciones que intercambien mensajes de tamaño acotado y admitido por UDP, y en la que la fiabilidad no sea prioritaria, o no sea asumible el tiempo de establecimiento de conexión TCP.

6.1.2 Vocabulario y formato

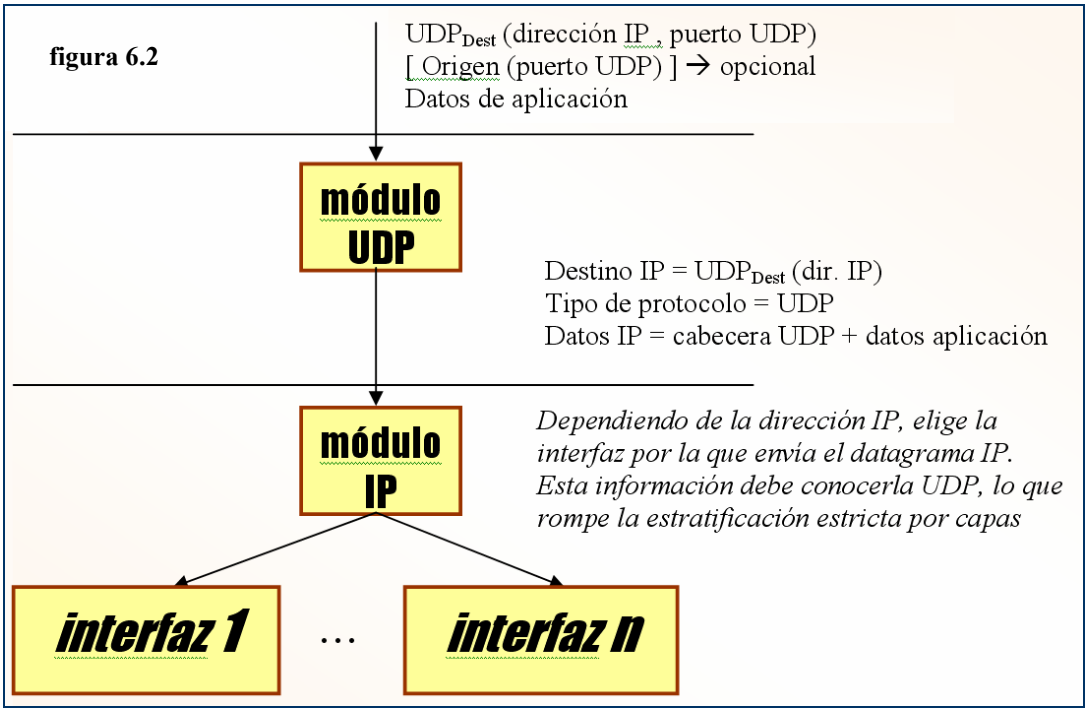
figura 6.1

Un mensaje UDP tiene una cabecera de 8 bytes, y un área de datos de tamaño limitado (en general 8 o 16 Kb).

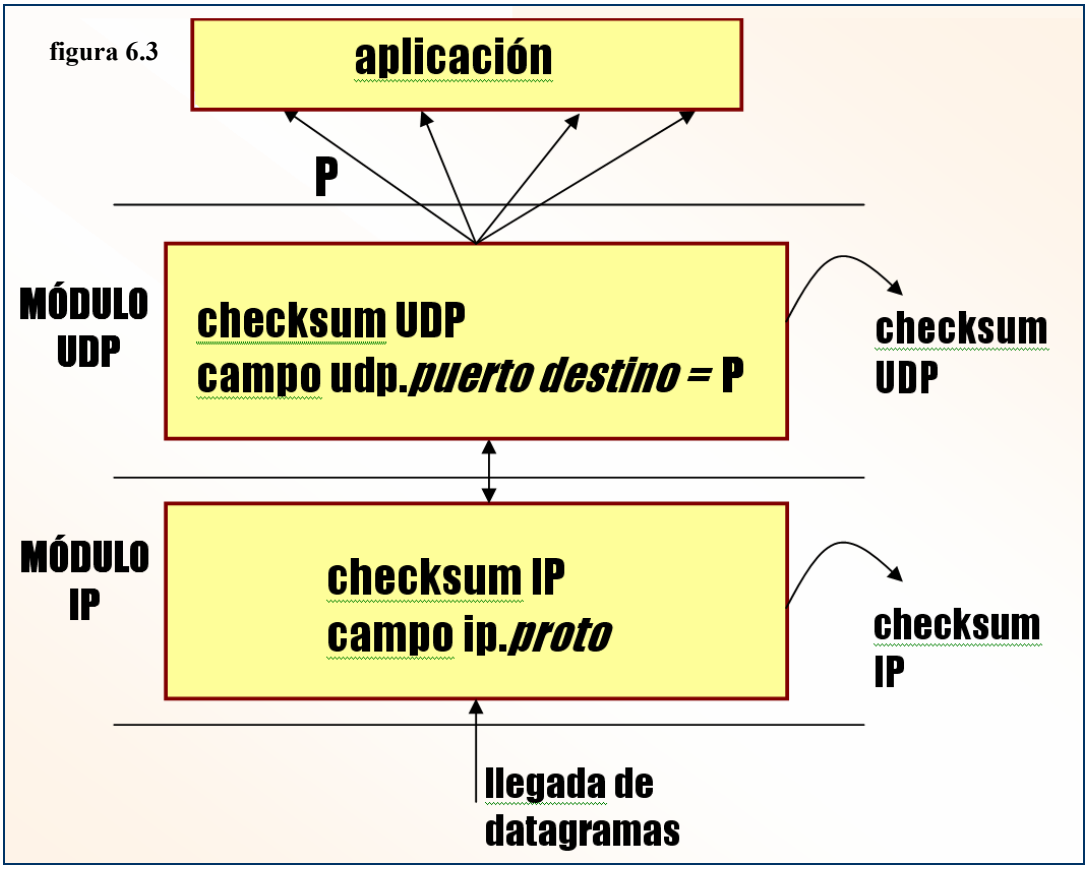
- **Puerto fuente**: En caso de que se pretenda que el destino responda a este datagrama UDP, se le puede indicar un puerto UDP de mi máquina a donde puede dirigirse. En caso contrario, lleva el valor 0.
- **Puerto destino**
- **Longitud**
- **Suma verificación**: Chequea datos UDP, cabecera UDP, y parte de la cabecera IP (rompiendo la estratificación por capas).

6.1.3. Procedimiento

Transmisión UDP:



Recepción UDP:



Para poder recibir datagramas UDP de un puerto, la aplicación ha de registrarse previamente (avisando al Sistema Operativo).

En sistemas *Unix*, para poder registrarse en la escucha de puertos UDP < 1024, es necesario tener permisos de 'root'. Ver figura 6.3

6.2 UDP y la transmisión de video.

Una vez vistas las características principales de UDP podemos entender el por qué es un protocolo idóneo para la transmisión de video. Es más, otros protocolos específicos para manejar flujos de video como pueden ser RTP o AVStreams de CORBA se basan en UDP.

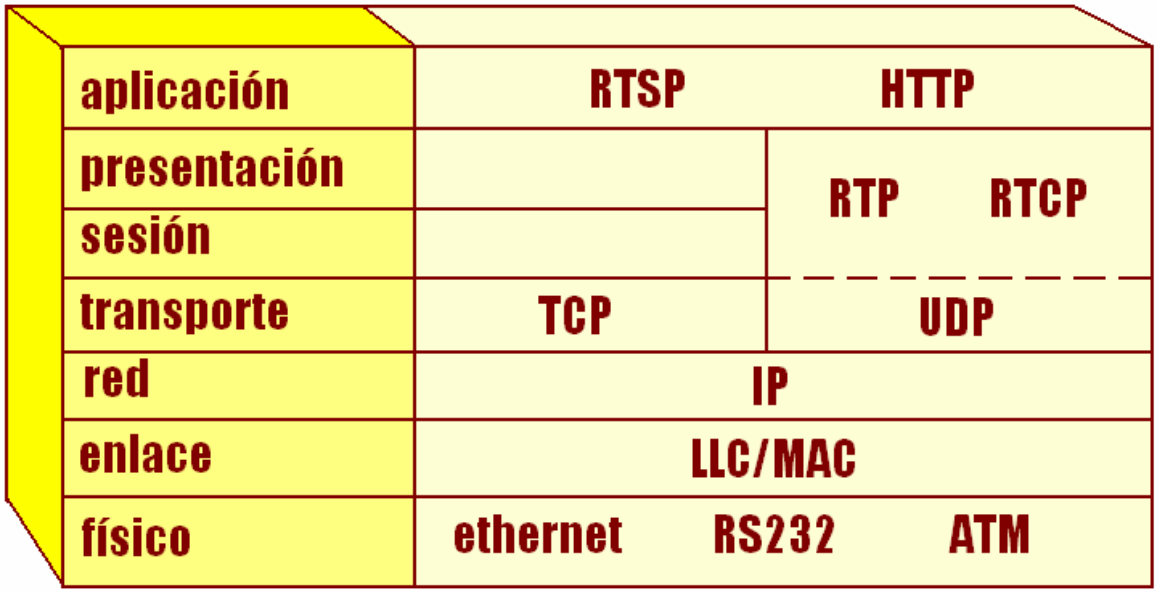
La alternativa a UDP es TCP, pero por su naturaleza no es el protocolo más indicado. Recordemos que TCP:

- Orientado a **conexión.**
- Proporciona **fiabilidad** y **entrega ordenada**. Durante la transmisión de video a tiempo real, si una imagen se pierde no es necesaria la retransmisión. O bien se pierde y no pasa nada o se pueden añadir mecanismos de control de errores para poder recuperar imagen, pero nunca una retransmisión. TCP sí que realiza retransmisiones.

En UDP, no se garantiza la entrega ordenada, aunque tiene solución usando un número de secuencia. Puede implementarse un *buffer* que internamente reordene tramas consecutivas, o simplemente que se descarten las tramas con número de secuencia anterior a la última recibida. Si se recibe vídeo a una tasa de 10 *fps*, la pérdida de uno o dos *frames* es un mal menor.

Como vemos, el uso de UDP es adecuado, pero puede ser necesario manejar una información adicional que UDP no ofrece. De ahí que protocolos específicos para la transmisión de video como RTP o AVStreams de CORBA se apoyen sobre UDP (ver figura 6.4).

figura 6.4 Modelo de referencia OSI



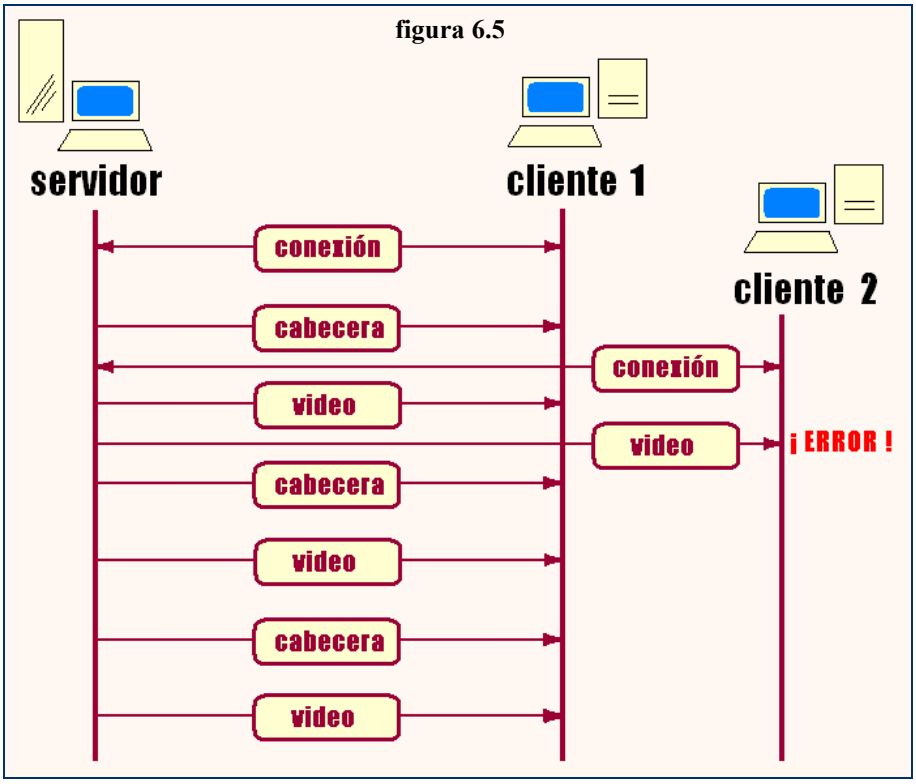
6.2.1 Sincronización. Fragmentación de imágenes.

Supongamos un servidor de video UDP que envía constantemente *frames* a un cliente. Sería de gran utilidad por parte del cliente obtener información relacionada con los *frames* que recibe (como se ha visto en el apartado anterior). Parámetros como:

- **número de secuencia**, para poder descartar *frames* desordenados, o bien ordenarlos por medio de un buffer, conocer cuantos *frames* se han perdido…
- **hora** a la que se envió el *frame*, para controlar retardos y *jitter* (ver apartado 6.2.2)
- **tamaño** del *frame* a recibir. En formatos que comprimen, el tamaño de cada *frame* varía, no así en formatos que no hacen. Además, es imprescindible cuando se usan protocolos que no garantizan la integridad del *frame,* como UDP.
- **otra información** que pueda interesar

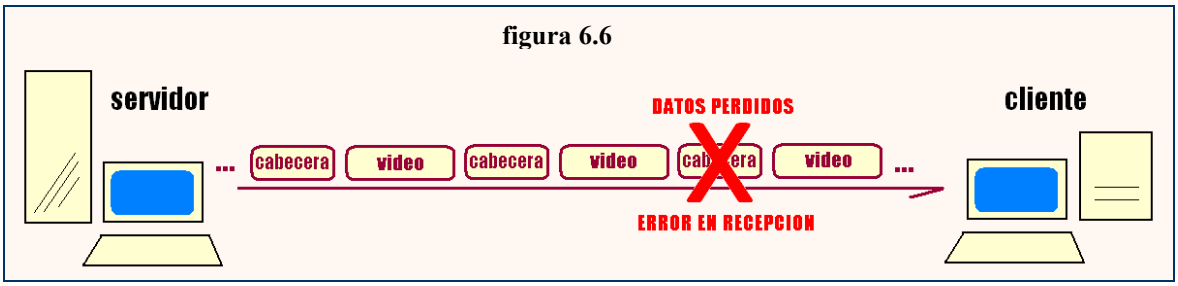
Una vez que conocemos qué tipo de información es interesante, debemos buscar la forma de enviarla. Parece claro que debe recibirse justo antes de cada *frame*, de forma que el servidor alternara el envío de información con el de imágenes.

figura 6.5



Pero surge un problema, el cliente no puede diferenciar cuándo recibe información, y cuándo imagen (sobre todo cuando recibe el primer paquete, ver figura 6.5). Es más, ¿y si se perdieran paquetes en la red o llegaran con información errónea?, se obtendrían paquetes con información que no sería útil (figura 6.6).

figura 6.6



Existen varias soluciones posibles. La más sencilla consiste en emitir cada cierto tiempo señales de referencia que permitan sincronización. El emisor emite siempre la misma secuencia: … 1 *sincronización* 2 *cabecera* 3 *video* 1 *sincronización* 2 *cabecera* 3 *video*… El cliente siempre espera recibir esa misma secuencia de paquetes. La primera vez que reciba, desechará la información hasta que encuentre la señal de sincronización. Una vez recibida sabe en qué momento le llegará información y en qué momento el video (figura 6.7). Tras la recepción de cada *frame* esperará a recibir información de sincronización, de manera que si se perdieran datos, el cliente sería capaz de volver a sincronizarse con el servidor (figura 6.8).

figura 6.7

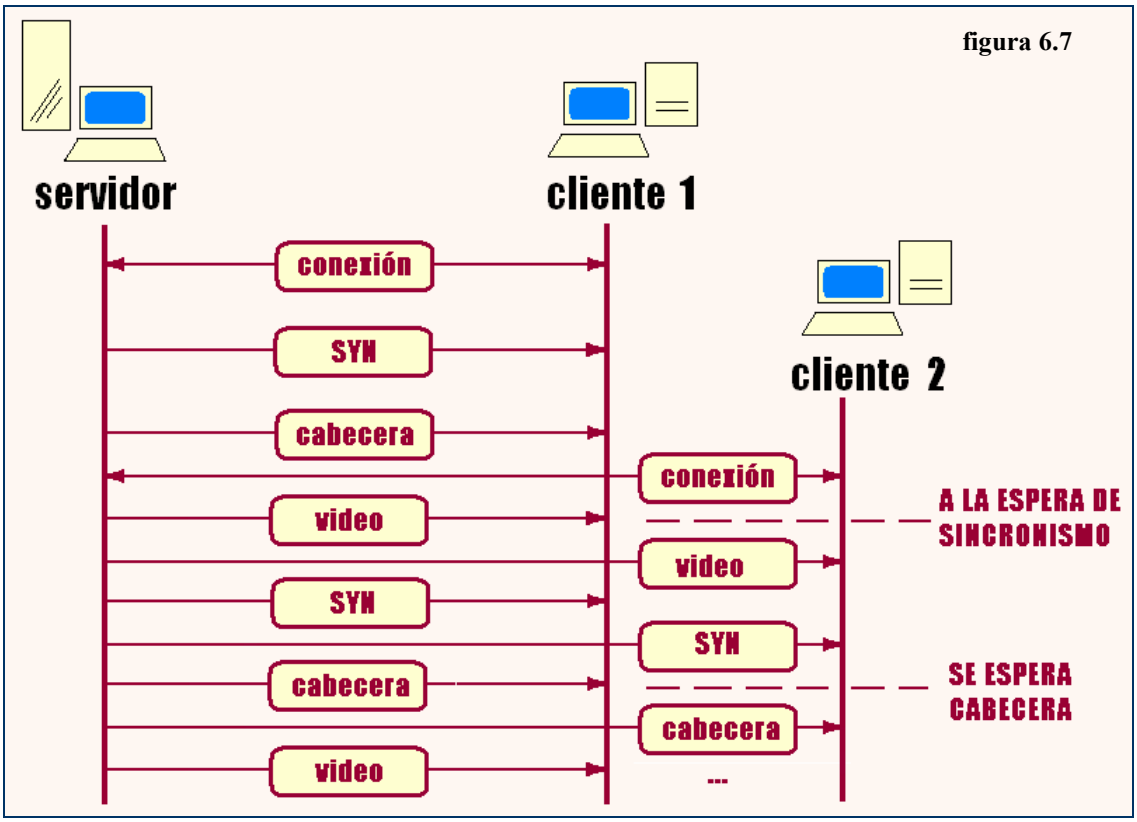
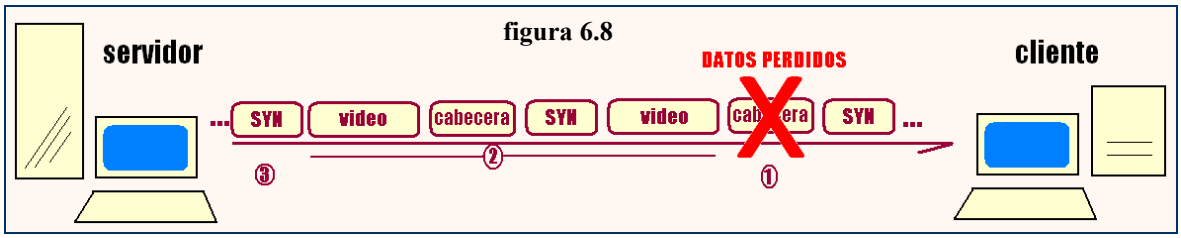


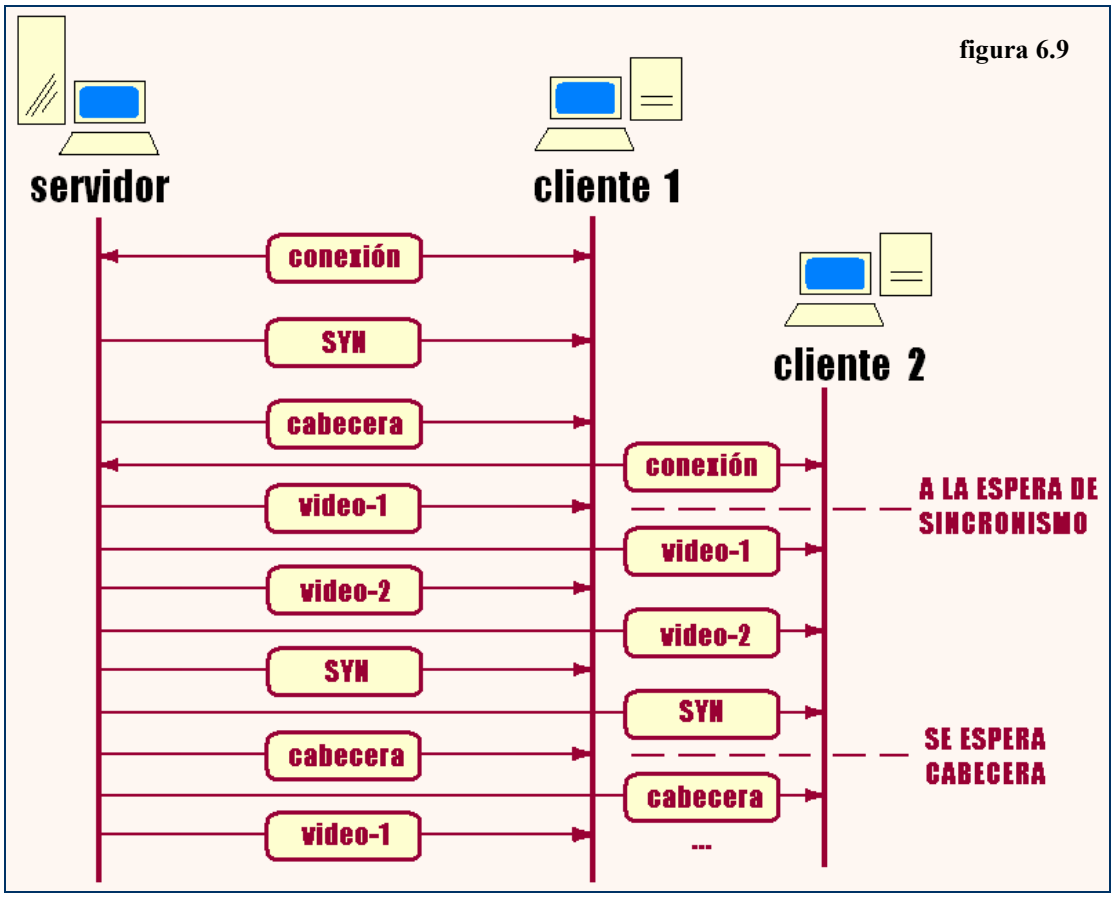
figura 6.8



(**1**) Se pierde o modifica una cabecera durante la transmisión
(**2**) El cliente no puede detectar las pérdidas, y recibirá parte de información de video como si fuera de cabecera, y parte de la trama de sincronización como video (la imagen se reproduciría con errores). Será cuando se pretenda recibir sincronización (**3**), cuando se reestablezca la recepción correcta de imágenes. En este ejemplo se pierden dos imágenes antes de recuperar la sincronización con el servidor

Los protocolos de transmisión, no permiten enviar paquetes de tamaño superior a 64 *kbytes*, incluso el tamaño máximo permitido suele ser mucho más pequeño. De este modo, si la imagen excede de ese tamaño, debemos fragmentar la imagen en varios paquetes (ver figura 6.9). Esto da lugar a que se añada un parámetro más a la cabecera de información: **número de fragmentos** en los que se fragmenta la imagen. Así, el cliente podrá saber cuántos paquetes de imagen debe recibir y unirlos en destino.

figura 6.9



También cabe la posibilidad de encapsular la información de video junto a la cabecera. Deberíamos crear entonces paquetes de longitud fija, y en el caso de que sobrara espacio, rellenar con bits de *padding*. Esto no significa que debamos prescindir de los paquetes de sincronización.

6.2.2 Jitter. *Buffer* de recepción.

El *Jitter* puede definirse como la variación de retardo en el tiempo entre *frames* consecutivos. Existen multitud de definiciones de *jitter*, pero tal vez en nuestro caso ésta sea la más acertada. Se puede medir usando diferentes técnicas, como media, desviación típica, máximo o mínimo tiempo entre *frames*… Es habitual observar el concepto *jitter* en el mundo de las telecomunicaciones, hablando de paquetes en lugar de *frames*. Es cierto que en transmisión de vídeo, el *jitter* se debe a ese retardo entre paquetes, pero también puede darse *jitter* a la hora de capturar video, procesarlo… *Jitter* es la variación de la media. Ver figuras 6.10 y 6.11.

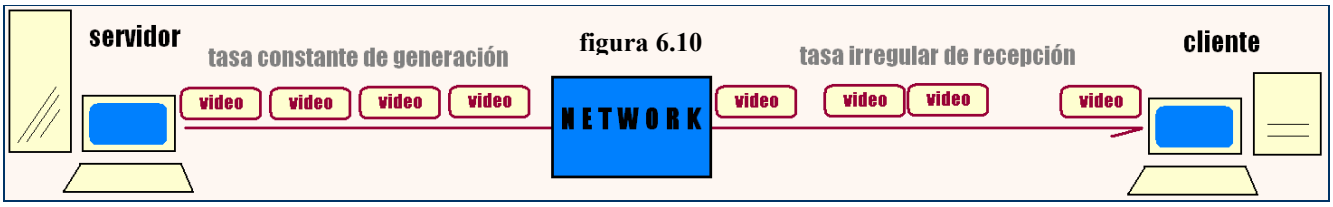


figura 6.10

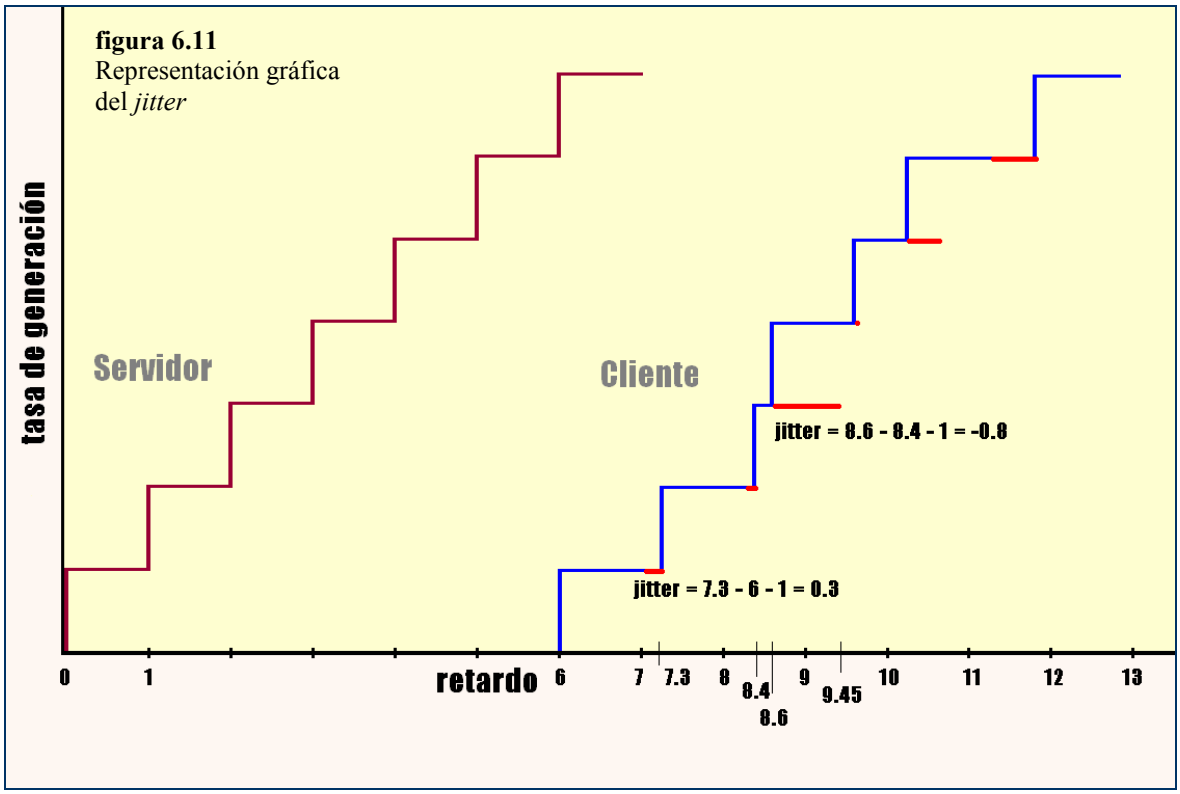
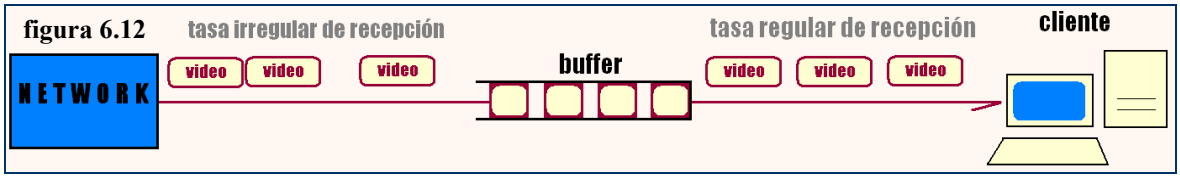


figura 6.11
Representación gráfica del *jitter*

El receptor puede corregir el *jitter* utilizando un *buffer* de recepción. A partir de este *buffer*, y conociendo la tasa de generación de *frames* del emisor, se puede reestablecer la tasa original de envío. Incluso el buffer sería capaz de ordenar internamente los *frames* que recibe según su número de secuencia.



El uso de un *buffer* también tiene inconvenientes, si el buffer es grande, ampliaremos el retardo de transmisión, y si es pequeño, es muy probable que no consigamos corregir el *jitter* si éste es muy grande.

6.3 Introducción a la creación del controlador UDP

El propósito de la creación de un controlador UDP, es comprender el funcionamiento de un protocolo de transmisión de video en tiempo real partiendo de UDP, así como el de otros protocolos diseñados específicamente para su uso en transmisiones en tiempo real, como son RTP o AVStreams de CORBA que parten de UDP. La idea consiste en crear un "nuevo protocolo" que sea capaz de enviar y recibir video, pero sin llegar a la complejidad de estos dos últimos.

Dentro del marco de este PFC, en primer lugar se realizó una conexión punto a punto entre cliente y servidor. Una vez establecida esta conexión, se utilizó un formato de compresión que no consumía gran ancho de banda y no necesitaba fragmentación de paquetes UDP. Al añadir la posibilidad de emplear formatos de video sin compresión, que consumen bastante ancho de banda, se tuvo que implementar la fragmentación de paquetes y el uso de cabeceras.

Más tarde se consiguió que un servidor atendiera peticiones de varios clientes, y que éstos consiguieran desconectarse con éxito y sin que el servidor perdiera estabilidad. Para que un cliente pueda diferenciar cuándo recibe cabecera y cuándo video (todos los clientes reciben lo mismo al mismo tiempo, además, pueden haber pérdidas en la red), debe recibir paquetes de sincronización, y esta función también se implementó.

Finalmente se agrupó el código de forma que pudiera implementar la interfaz controlador.

6.3.1 Compresión

Los formatos de compresión soportados por el controlador UDP son MJPEG, RGB24 y RGB15.

El formato MJPEG ocupa muy poco ancho de banda, y no necesita que los paquetes sean fragmentados para su envío. El tamaño de cada *frame* al sufrir compresión es variable y oscila entre 8 y 15 *Kbytes*. Esto obliga a conocer en recepción el tamaño de cada imagen que se va a recibir (para ello es conveniente el uso de cabeceras, ver apartado 6.3.2).

RGB24 captura imágenes con 24 bits de color sin comprimir. El tamaño de cada *frame* es de unos 280 *Kbytes* (en este caso el tamaño es fijo, pues no hay compresión), frente a los 10 *Kbytes* de media de las imágenes en MJPEG.

UDP bajo *linux*, no permite redireccionar a través de la tarjeta de red paquetes mayores a 64 *Kbytes*. Surge de ahí la necesidad de una fragmentación de paquetes.

Si las imágenes se fragmentan, se debe informar al cliente del número de paquetes en que se ha fragmentado cada imagen para que pueda recibirlos por orden. La solución consiste en añadir una cabecera previa al envío de los paquetes de video que indique a qué imagen pertenecen. Además puede aprovecharse para enviar otros parámetros de interés.

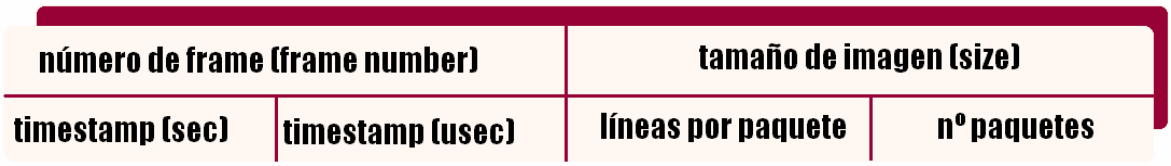
Para compresión MJPEG o JPEG**,** no es necesaria la fragmentación. En las pruebas realizadas en el laboratorio el tamaño de cada imagen estaba (como se ha dicho) entre 8 y 15 *Kbytes*. Es probable que con una cámara de mayor calidad, las imágenes capturadas sean de más tamaño y se necesite fragmentar. No se ha implementado una fragmentación de paquetes para MJPEG, aún así, la fragmentación sería sencilla (apoyándonos en las cabeceras), y acarrearía el problema de que si se perdiera un fragmento o se deteriorara por la red, se perdería la imagen completa (en RGB24 sólo se pierde la zona de la imagen que contiene el fragmento deteriorado).

El caso de RGB15 es idéntico al de RGB24.

6.3.2 Cabeceras y fragmentación

Independientemente del formato de compresión, una cabecera se envía previamente al video. El formato de la cabecera que se ha definido es el que se observa en la figura 6.13.

figura 6.13 Formato de cabecera de imagen



- **número de *frame***: el servidor numera los frames de forma consecutiva. Es un número de secuencia.
- **tamaño de la imagen**: contiene el tamaño total de la imagen en *bytes.*
- **timestamp (sec)**: estampa de tiempo del servidor en segundos
- **timestamp (usec)**: estampa de tiempo del servidor en milisegundos
- **líneas por paquete**: cuando una imagen sin comprimir se fragmenta en paquetes, ésta se divide en líneas, y cada paquete contiene varias. Aquí se indica cuantas líneas están contenidas en cada paquete
- **nº paquetes**: en cuantos paquetes se fragmenta una misma imagen

Los campos **número de *frame***, **timestamp** y **size** son útiles para todos los formatos de compresión. **size** será variable si los frames están comprimidos, y fijo si no lo están. **Líneas por paquete** y **número de paquetes** se emplean para formatos de video sin compresión (RGB24). Otro parámetro imprescindible para el formato RGB es **bytes por línea**, que, sin embargo, no se envía en la cabecera, pues tanto cliente como servidor ya lo conocen (su valor se negoció durante la conexión TCP).

El caso de MJPEG es sencillo, pues no hay fragmentación de imágenes. El servidor envía una cabecera informando del tamaño de imagen que va a enviar. El cliente la recibe, la analiza y espera una imagen del tamaño que el servidor le ha anunciado.

Con la fragmentación, el envío de imágenes RGB24 se complica. Veamos el ejemplo de la figura 6.14.

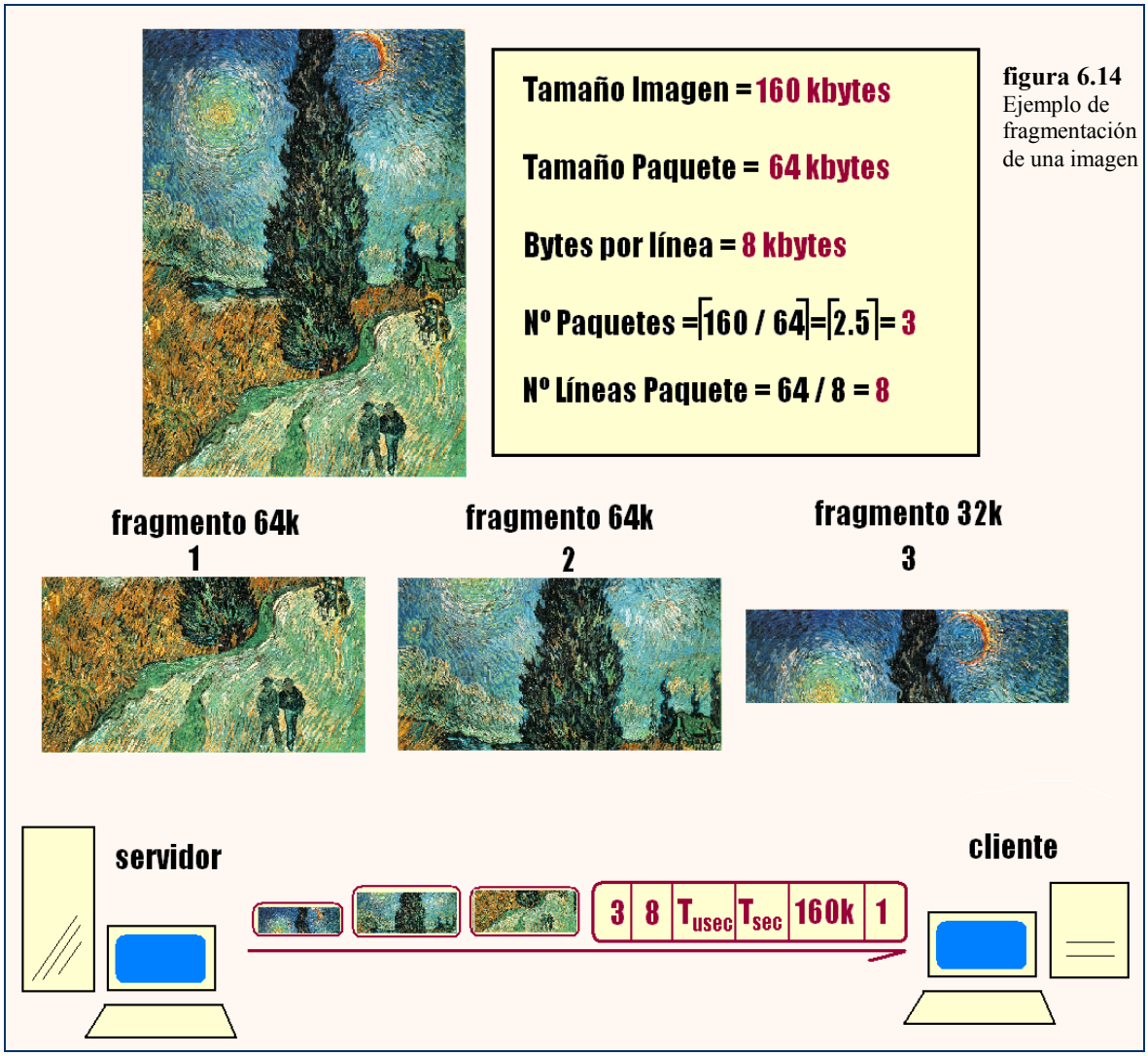


figura 6.14 Ejemplo de fragmentación de una imagen

Una imagen de 160 *Kbytes* va a ser enviada por el servidor. Se sabe que el tamaño de un paquete es de 64 *Kbytes* como máximo (además, interesa aprovechar el tamaño del paquete al máximo). Así que para calcular el número de paquetes que se van a necesitar, tan sólo hay que dividir. Como no es posible enviar dos paquetes “y medio”, se enviarán 3 paquetes.

Una vez que se conoce el número de paquetes (igual al número de fragmentos) en los que se envía la imagen, debemos calcular cuantas líneas caben en cada paquete. 64 *Kbytes* de un paquete entre 8 *Kbytes* por línea (valor que se negoció), se obtienen 8 líneas por paquete.

Si el cliente sabe a través de una cabecera que le van a llegar 3 paquetes con 8 líneas cada paquete, y que cada línea corresponde a 8 *Kbytes* de imagen (en total 192 *Kbytes*), ya sabe cuantos *Kbytes* debe recibir, y cuantos debe descartar, pues el tamaño de la imagen según la cabecera es de 160 *Kbytes* (los últimos 32 no contienen imagen).

El motivo de dividir una imagen en líneas tiene explicación en los apartados **6.4.3.1.4.2 *send_UDP_Video()*** y **6.4.3.2.2.2 *recv_UDP_Video().***

Además, como se observa en el dibujo, la imagen se fragmenta de una forma un tanto ilógica, pero el cliente al almacenarla como una pila la reconstruye, el motivo de este tipo de fragmentación se explica también en esos apartados. Ver figura 6.15.

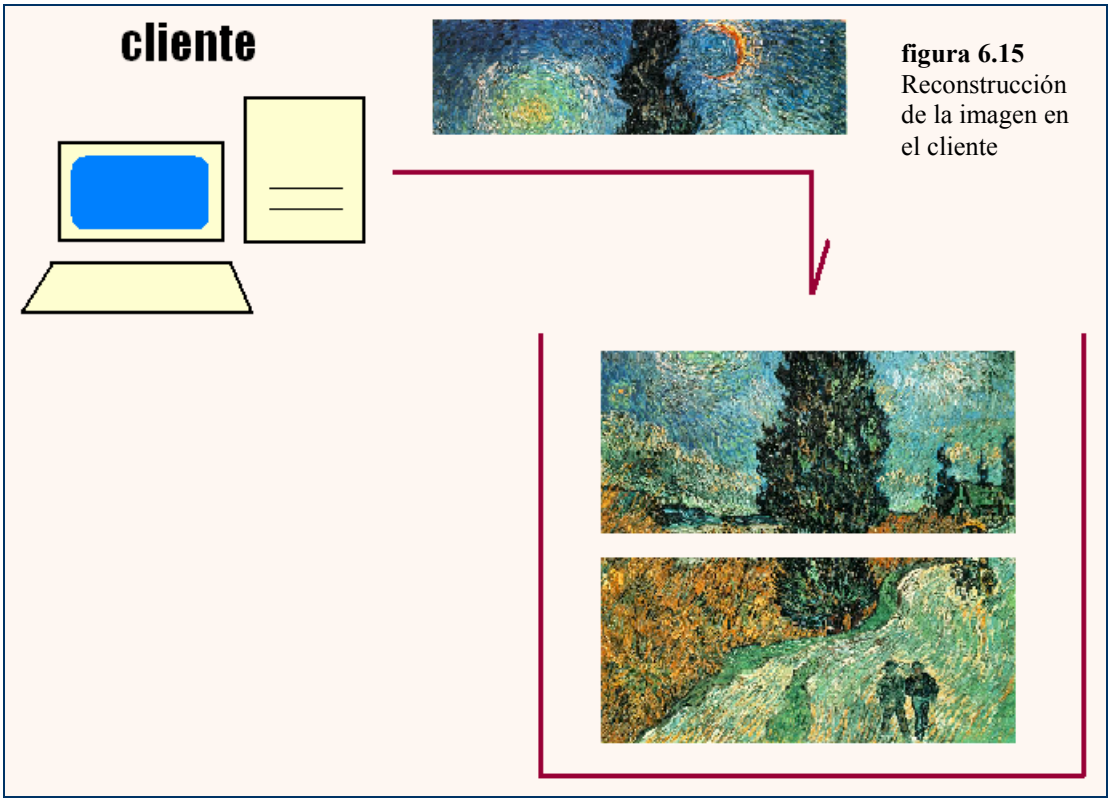


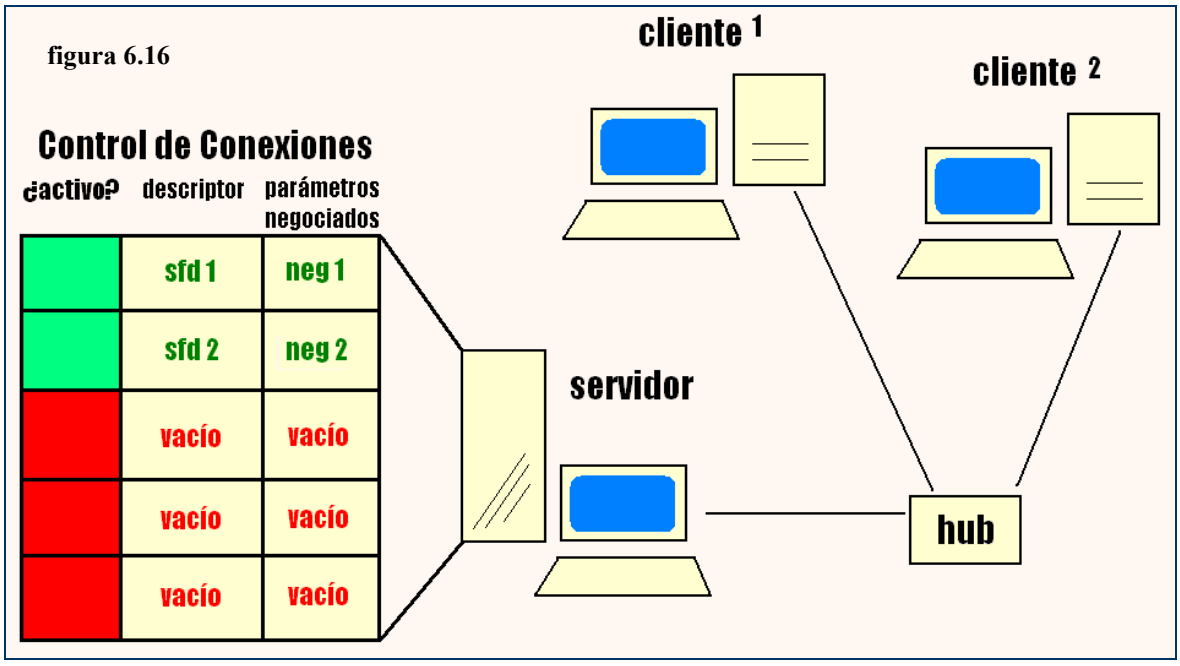
figura 6.15 Reconstrucción de la imagen en el cliente

6.3.3 Un servidor, varios clientes. Sincronización

Un servidor que se encarga de capturar video y enviarlo en tiempo real a todos los clientes conectados, no debe quedarse a la espera de captar nuevos clientes, pues bloquearía el proceso de captura y envío de vídeo. Por esta razón, se ha creado otro proceso que se encarga de las peticiones de conexión por parte de clientes (conexión TCP, ver apartado 4.2.3.4). Además, este proceso es quien se encarga de actualizar la lista de clientes del servidor que captura y envía video. Ver diagrama de secuencia de la figura 6.26

El servidor de video UDP mantiene una tabla de clientes en la que se indica el descriptor asociado a cada uno, cuales están activos, parámetros asociados a la conexión (formato de video, compresión…) (Ver figura 6.16). Además, el servidor comprueba constantemente si tiene clientes conectados. Como el ancho de banda y los recursos del servidor son limitados, se limita el número máximo de conexiones, rechazando las que hagan exceder ese límite. Además, el servidor es capaz de detectar desconexiones o "caídas" de los clientes a los que les está ofreciendo video (ver apartado 6.3.4).

figura 6.16



Los dos procesos, que trabajan de forma independiente, acceden a la tabla de conexiones cliente. Uno añade conexiones a la tabla, y otro la consulta cuando envía el video (para saber a quién hay que enviar).

Sabemos que el servidor envía video precedido de un paquete de cabecera. Si el proceso que capta clientes, añadiera un nuevo cliente en la tabla en el momento previo al envío de un *frame* (y en el instante posterior al envío de la cabecera), el servidor de video le enviaría a ese cliente la imagen, pero no la información asociada a ésta. Por otra parte, el cliente una vez que se conecta, lo primero que espera es una cabecera, y esto provocaría un error en recepción.

El error se puede solucionar mediante el uso de semáforos. No se añade un cliente a la tabla hasta el instante previo al envío de una cabecera. Aún así, se ha preferido implementar un mecanismo de sincronización. El motivo es, que además de poder

solucionar este problema, permite al cliente “recuperarse” ante la pérdida de paquetes en la red, pues espera paquetes de sincronización cada cierto tiempo. Ver figura 6.17.

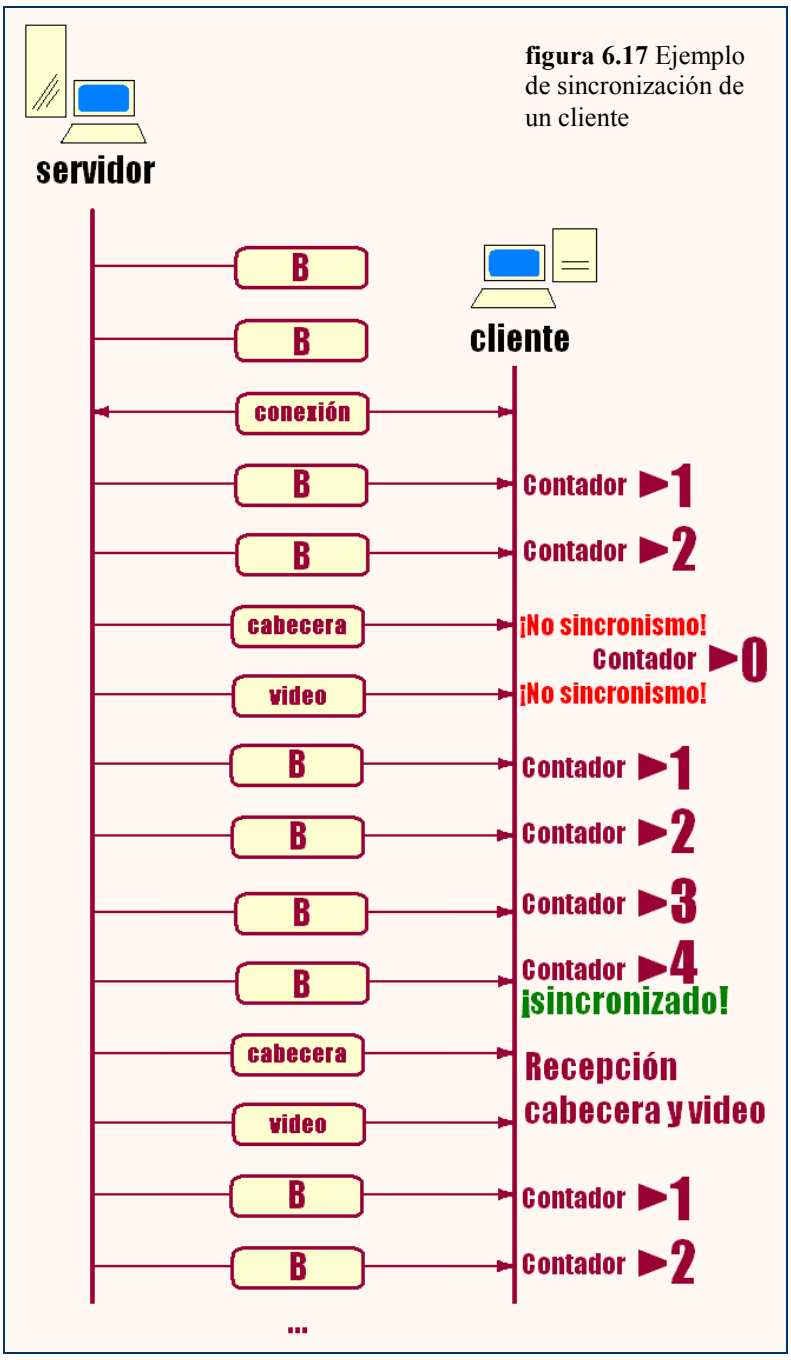


figura 6.17 Ejemplo de sincronización de un cliente

La implementación UDP utiliza como mecanismo de sincronismo el envío/recepción de cuatro caracteres “B” de forma consecutiva. El servidor envía:

1 - “B” “B” “B” “B”
2 - cabecera
3 - video
4 - “B” “B” “B” “B”
5 - cabecera
6 - ……

El cliente sabe que la recepción de la cabecera se hará tras recibir las cuatro “B” consecutivas (hace una comprobación *carácter* a *carácter*). Así, la primera vez que se conecta, hasta que no las recibe, sabe que debe descartar los paquetes que le llegan (pues no sabe qué contienen). Si algún paquete se perdiera de forma total o parcial, podría recuperar el sincronismo, porque después de recibir cada *frame* vuelve a esperar las cuatro “B” consecutivas.

Es posible que el propio video o alguna cabecera contengan el carácter “B”, pero en un principio parece poco probable que contengan cuatro consecutivas (tal vez no sea así), y aunque se diera el caso, el cliente recuperaría el sincronismo en la próxima recepción de caracteres de sincronización. Este mecanismo de sincronismo es muy simple, sería recomendable un estudio en profundidad en temas de sincronización. El carácter “B” se ha elegido aleatoriamente.

6.3.4 Desconexión

Una de las cualidades más importantes de un servidor es que sea estable frente al comportamiento de los clientes. Los clientes pueden provocar la inestabilidad del servidor.

Por ello un servidor debe controlar, al menos, las posibles desconexiones de sus clientes, debe saber cuáles tiene conectados, se han desconectado o simplemente no responden (se cortó la comunicación por algún motivo ajeno al cliente). El servidor UDP implementado, por defecto permite 5 clientes, si los que se desconectan no fueran borrados de la tabla interna del servidor, no permitiría más conexiones y llegaría un momento en que ningún cliente podría conectarse.

En el caso de la implementación UDP que se ha realizado, el servidor detecta la desconexión o “caída” de algún cliente cuando intenta enviarle video y no se lleva a cabo con éxito. Cuando esto se produce, busca el cliente en la tabla (a partir de su descriptor) y lo marca como inactivo, además de cerrar la conexión UDP con ese cliente.

En otras implementaciones usando protocolos específicos para la transmisión de video (RTP, AVStreams…) la desconexión de clientes se detecta mediante el envío de mensajes de control por parte de los clientes.

6.4. Implementación del controlador UDP

6.4.1 La estructura *UDP_Ctrl*

Implementa a la estructura **`controller`**.

```
struct Controller UDP_Ctrl = {

      name:           "UDP",
      id:             0,
      init:           init_UDP,                 //SERVIDOR
      add_Client:     add_UDP_Client,           //SERVIDOR
      remove_Client:  remove_UDP_Client,        //SERVIDOR
      connect:        connect_UDP,              //CLIENTE
      send:           send_UDP,                 //SERVIDOR
      recv:           recv_UDP,                 //CLIENTE
      close:          close_UDP,                //SERVIDOR Y CLIENTE
};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
`init_UDP()`	`init()`	Inicializa controlador
`add_UDP_Client()`	`add_Client()`	Añade cliente al servidor
`remove_UDP_Client()`	`remove_Client()`	Borra cliente del servidor
`connect_UDP()`	`connect()`	Conecta cliente con servidor
`send_UDP()`	`send()`	Envía video
`recv_UDP()`	`recv()`	Recibe video
`close_UDP()`	`close()`	Cierra conexión

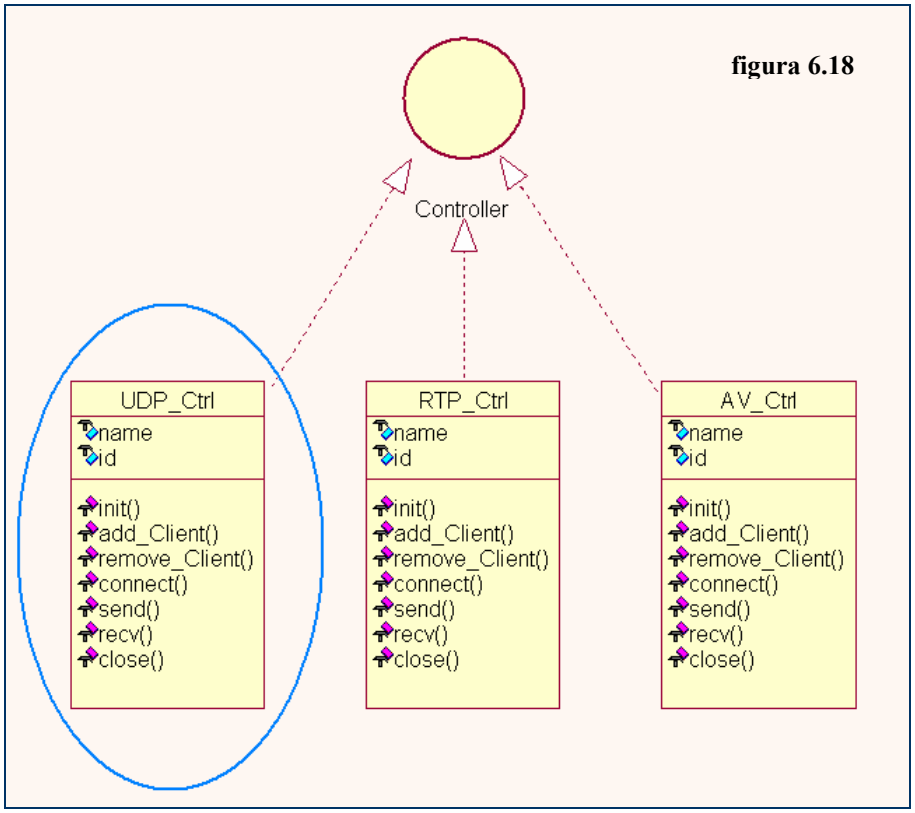


figura 6.18

6.4.2 Otras estructuras

6.4.2.1 La estructura *connection*

```
struct connection
{
      int fd;
      int active;
      struct negotiation n;

};
```

Contiene la información relacionada con la conexión de un cliente. El descriptor de *socket* UDP se guarda en **`fd`**, si el cliente está conectado el campo **`active`** se pone a 1 (en caso contrario a 0). Los parámetros que se negociaron con el cliente se guardan en la estructura `negotiation` **`n`**.

```
struct connection connections[MAX_CONNECTIONS];
```

Se define un array de estructuras **`connection`** de tamaño MAX_CONNECTIONS (ver *grab-ng.h*) donde se guarda la información de las conexiones de todos los clientes.

6.4.2.2 La estructura *Frame_Seq_HDR*

```
struct Frame_Seq_HDR
{
    long          frame_number;
    int           size;
    long long     sec;
    long long     usec;
    int           linesPerPackage;
    int           numberOfPackages;
};
```

Es la estructura que corresponde a la cabecera que se envía antes del video. Ver apartado 6.3.2 y la figura 6.13.

6.4.2.3 La estructura *avi_handle*

```
struct avi_handle {
    int                   fd;
    char                  IP[15];
    struct iovec          *vec;

    long long ts;
    struct ng_video_fmt   vfmt;
    struct ng_audio_fmt   afmt;
    int                   frames;
};
```

Ver apartado 5.3.4. Se define tanto en los *plugins* **avi_net_reader** y **avi_net_writer** como en los controladores. No se define en *grab-ng.h* porque cada *plugin* tiene una implementación de **avi_handle** diferente. ***vec** está relacionado con la fragmentación de video cuando se fragmenta un *frame* en varios paquetes (ver el uso de la variable **cluster** en el apartado 6.4.3.1.4.2 *send_UDP_Video()*)

6.4.3 UDP_Ctrl

6.4.3.1 Implementación de funciones servidor

Un controlador tiene definidas interfaces que usa el servidor e interfaces que usa el cliente. A continuación se describen las funciones relacionadas con el servidor.

6.4.3.1.1 *init_UDP()*

```
int init_UDP(){
	int i;
	for(i=0; i<MAX_CONNECTIONS - 1 ; i++)
	{
		connections[i].active = 0;
	}
return 0;
}
```

Fija el campo **active** de todas las conexiones a cero.

6.4.3.1.2 *add_UDP_Client()*

```
int add_UDP_Client(struct sockaddr_in cli_ad, struct negotiation ng)
```

Añade un cliente UDP. En primer lugar se comprueba si queda sitio para añadir la nueva conexión en el *array* de conexiones **connections**. Si queda sitio, se rellena la primera estructura **connection** vacía:

- Se iguala **active** = 1
- Se abre *socket* UDP y se guarda su descriptor en el campo **fd**
- Se rellena la estructura **negotiation** asociada a la conexión a partir de valores que se negociaron (estos valores se pasan a la función a través de **ng)**.
  - Se usa un *array* llamado **delay[]** (indexado de la misma forma que connections), que contiene los *rate* de cada cliente. Las variables que contiene el *array* son auxiliares. Sirven de contador que se decrementa hasta valer 0, en ese momento se envía la imagen (ver función **send_UDP_Video()**).

Una vez se ha rellenado la estructura, se llama a la función **connect()** de la librería *socket* a partir del **fd** anterior (asociado a la conexión cliente que se añade) y la dirección del cliente **cli_ad**.

Es el servidor el que, una vez que conoce la dirección IP del cliente **`cli_ad`** se conecta a él. Aunque desde el punto de vista de aplicación, es el cliente el que conecta con el servidor para recibir video, en realidad, el cliente sólo conecta con el *thread* encargado de captar clientes y cierra la conexión (quedándose a la espera de una conexión remota). El *thread* "avisa" al servidor de video, y es éste el que conecta con el cliente a través de **`add_UDP_Client()`**.Ver diagrama de secuencia de la figura 6.26

El servidor hace uso de multiplexación E/S síncrona (ver anexo) para poder servir a varios clientes a través del mismo puerto, por ello se añade el **`fd`** al grupo de escritura **`wr`** de los *sockets* activos, y se actualiza el descriptor máximo (si es necesario).

6.4.3.1.3 *remove_UDP_Client()*

No implementado, los clientes son eliminados de forma automática por el servidor cuando se desconectan (ver apartado 6.4.3.1.4).

6.4.3.1.4 *send_UDP()*

```
int send_UDP(void *handle, struct ng_video_buf *buf){

      if (send_UDP_headers((buf->size + 3) & ~3) == -1) return -1;
      if (send_UDP_Video(handle,buf) == -1) return -1;
      return 0;
}
```

Llama a las funciones **`send_UDP_headers()`** y **`send_UDP_Video()`**. **`*buf`** contiene el video que se va a enviar. Devuelve -1 en caso de error, y 0 si todo es correcto.

6.4.3.1.4.1 *send_UDP_headers()*

```
int send_UDP_headers(int size)
```

Envía señal de sincronismo y cabecera previa a una imagen. La señal de sincronismo consiste en la secuencia "B" "B" "B" "B" (ver apartado 6.3.3), y la cabecera **`Frame_Seq_HDR`** (apartado 6.4.2.2).

Se reserva memoria para la cabecera y se rellenan sus campos (entre ellos **`size`**, excepto los relacionados con la estampa de tiempo), también se define el separador (señal de sincronismo) "B".

Si no hay ningún cliente conectado, no se hace nada y se liberan recursos (memoria reservada para la cabecera)

Si hay uno o más clientes conectados, para cada conexión del *array* **`connections`**:

- se hace una comprobación del estado de cada *socket* del grupo de escritura **`wr`** (**`select()`** de la librería <*unistd.h*>)

- Se comprueba si está activa y si su **fd** pertenece al grupo de escritura **wr**.
- Se comprueba si el retardo impuesto a esta conexión (posición **i**) ha espirado (**delay[i] = 0**). Si ha expirado:
  - Se envía "B" "B" "B" "B" carácter a carácter (sincronismo).
    - Si hay error, se elimina cliente con la función **err_send()** (pone **active = 0** para la conexión asociada a ese cliente y cierra *socket* UDP).
  - Se obtiene la hora, se guarda en la cabecera (la hora se obtiene justo antes de enviar la imagen)
  - Se envía la cabecera.
    - Si hay error, se elimina cliente con la función **err_send()**.
- Libera memoria

6.4.3.1.4.2 *send_UDP_Video()*

```
int send_UDP_Video(void *handle, struct ng_video_buf *buf)
```

Envía un *frame*. La información de video está contenida en ***buf**. ***handle** es de tipo **avi_handle** (apartado 6.4.2.3), de ella se obtiene el formato de video y la compresión.

Se comprueba la compresión de video. Dependiendo del formato de compresión, habrá fragmentación de paquetes o no.

Para compresión **RGB24** o **RGB15,** se fragmenta la imagen (omitiendo el código no relacionado con la fragmentación)

```
struct iovec *cluster;
bpl = h->vfmt.width * ng_vfmt_to_depth[h->vfmt.fmtid] / 8;
linesPerPackage= (MAX_UDP_PACKET_SIZE-8512)/bpl;

packageSize = linesPerPackage*bpl;

numberOfPackages = size/(packageSize);
if (size % (packageSize) != 0) numberOfPackages++;
for (cluster = h -> vec, j = numberOfPackages - 1; j >= 0; j--)
{
      cluster->iov_base = ((unsigned char*)buf->data) +   (packageSize * j);
      cluster->iov_len = packageSize;
}
```

Un paquete se ha dividido en líneas. **bpl** son los *bytes* que contiene cada línea. Así, conociendo el tamaño máximo de paquete UDP soportado por *linux*, y los *bytes* por línea, podemos calcular cuántas líneas **linesPerPackage** como máximo puede contener un paquete. Se han restado 8512 *bytes* al tamaño máximo de paquete porque *red hat linux 8.0* (sobre el que se ha programado la aplicación) no redireccionaba paquetes de tamaño mayor (en versiones anteriores de *red hat*, el tamaño de paquete debe ser incluso más pequeño).

El tamaño de paquete idóneo **packageSize**, una vez que conocemos las líneas que caben en cada paquete y de cuántos *bytes* se compone cada línea, se calcula multiplicando ambos valores.

El número de paquetes **numberOfPackages** en los que se fragmenta una imagen se calcula a partir del tamaño **size** de la imagen y del tamaño de paquete **packageSize**. El número de paquetes debe ser un valor entero (no se pueden enviar "medio paquete"), por ello se coge el entero superior resultante de la división entre **size** y **packageSize**.

Resumiendo, ya se ha calculado el tamaño de cada paquete **packageSize**, el número de paquetes **numberOfPackages** en los que se fragmentará la imagen, las líneas **linesPerPackage** que contiene cada paquete y los *bytes* **bpl** que contiene cada línea.

Se ha usado el vector **struct** iovec ***cluster** para llevar a cabo la fragmentación de la imagen.

```
struct iovec{
      ptr_t iov_base;  //dirección de inicio
      size_t iov_len;  //número de bytes
}
```

El campo **iov_base** apunta a una dirección (de memoria) de inicio, **iov_len** indica cuantos *bytes* recorrer desde la dirección de inicio.

El bucle **for** del fragmento de código anterior se repite tantas veces como número de fragmentos (o de paquetes) en los que se va a separar la imagen. En cada bucle se genera un fragmento. Por ejemplo, si una imagen se va a fragmentar en 3 paquetes, en el primer bucle la variable **j = 2**. Así, la dirección de inicio **cluster->iov_base** apunta a la posición de memoria donde se guarda la imagen más un desplazamiento en *bytes*, y **cluster->iov_len** indica cuantos *bytes* deben obtenerse para guardar en el paquete (se leen tantos como caben en un paquete).

La primera vez, el desplazamiento será:

packageSize * (j = 2) = desplazamiento equivalente al tamaño de dos paquetes

y se obtienen los **packageSize = cluster->iov_len** *bytes* siguientes. Figura 6.19. Como se observa en la figura, este paquete puede contener *bytes* "basura" (los paquetes son de tamaño fijo), pero el cliente los puede descartar en recepción, pues conoce el tamaño **size** de la imagen.

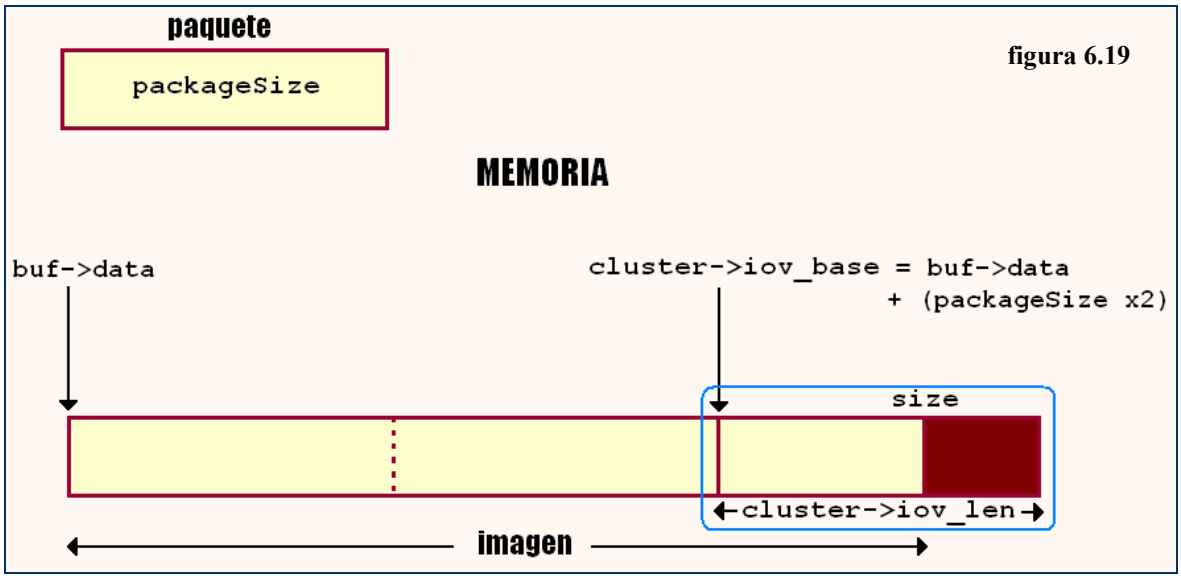


figura 6.19

En el siguiente bucle, j=1:

`packageSize` * (j = 1) = desplazamiento equivalente al tamaño de un paquete

y se obtienen los **`packageSize = cluster->iov_len`** *bytes* siguientes. Figura 6.20.

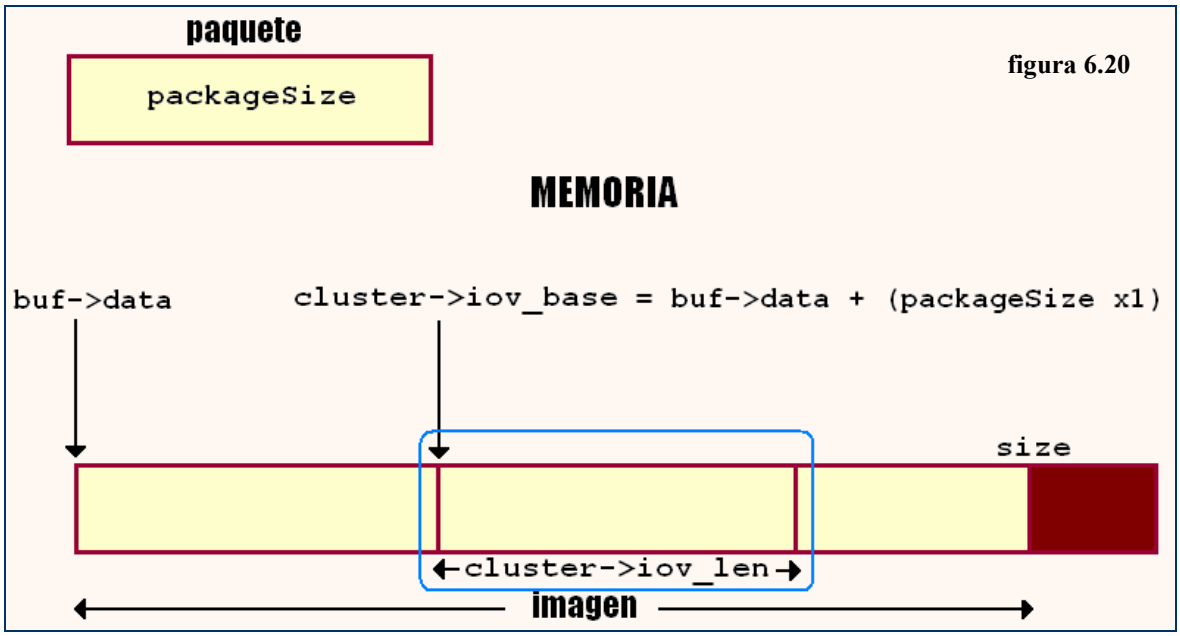


figura 6.20

En el siguiente bucle, j=0:

`packageSize` * (j = 0) = sin desplazamiento

y se obtienen los **`packageSize = cluster->iov_len`** *bytes* siguientes. Figura 6.21.

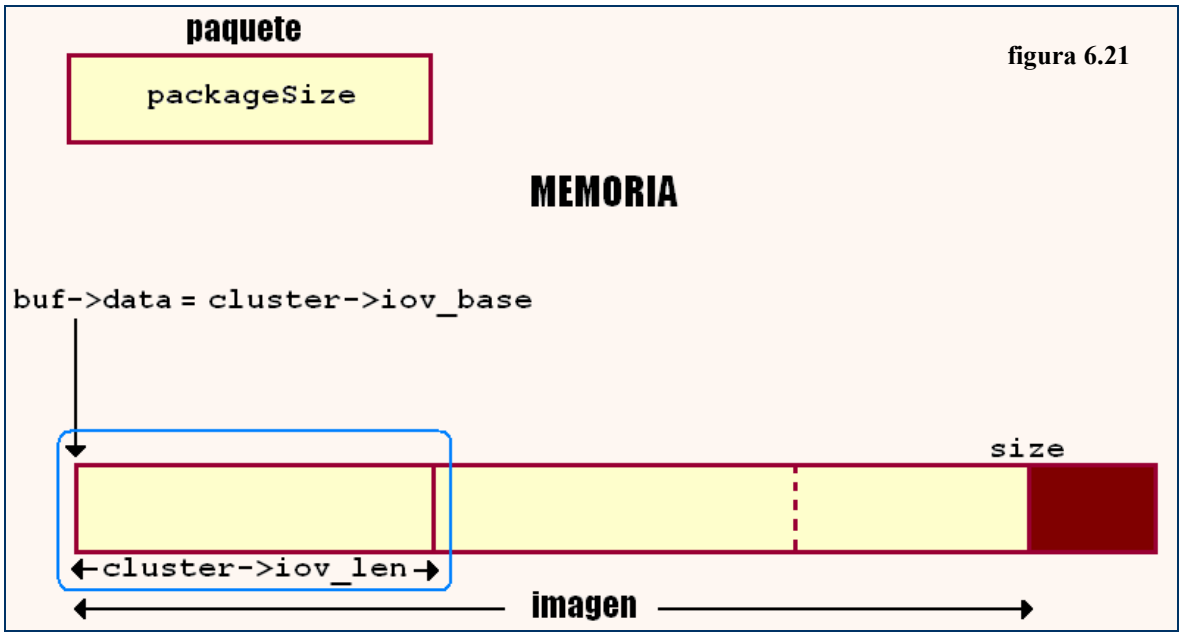


figura 6.21

Como se observa, el orden de creación de los fragmentos se realiza agrupando los *bytes* empezando por el **final** de la imagen (ver figura 6.14). Esta es la forma en la que *xawtv* lee una imagen de un fichero de video, y ha sido adaptada para la transmisión.

Si se ejecutara el código de arriba, cada vez que se entrara en el bucle se sobrescribirían los valores. Sólo se ha impreso el código necesario para entender el funcionamiento de la fragmentación, en realidad, el programa envía cada fragmento mediante la función **`writev()`**.

Si hay uno o más clientes conectados, para cada conexión del *array* **`connections`** (y para cada paquete o fragmento que se crea):

- se hace una comprobación del estado de cada *socket* del grupo de escritura **`wr`** (**`select()`** de la librería <*unistd.h*>)
- Se comprueba si está activa y si su **`fd`** pertenece al grupo de escritura **`wr`**.
- Se comprueba si el retardo impuesto a esta conexión (posición **`i`**) ha espirado (**`delay[i] = 0`**). Si ha expirado:
  - Se envía el **`struct`** `iovec` ***`cluster`** que apunta a un fragmento de la imagen mediante la función **`writev()`**.
    - Si hay error, se elimina cliente con la función **`err_send()`** (pone **`active = 0`** para la conexión asociada a ese cliente y cierra *socket* UDP).
  - Al enviar la imagen en fragmentos, el contador de retardo sólo se reinicia con el valor que se negocio para esa conexión (**`delay[i] = connections[i].n.rate`**) cuando se envía la imagen completa.
- Si el retardo impuesto a esta conexión no ha expirado todavía, se decrementa en uno por cada imagen (no por cada fragmento), y no se envía la imagen.

Además, debido a que la red *fast ethernet* sobre la que se trabaja soporta velocidades como máximo de 100 mbps, y el *switch* provoca colisiones, se ha tenido que implementar un retardo artificial por cada paquete. Esto sólo se da con el formato RGB24, que no comprime el vídeo y consume mucho ancho de banda. En una red más rápida no habría problemas, aún así sería recomendable que este retardo fuese negociado al iniciar la

conexión. También puede hacerse uso de la propiedad *rate* para no saturar la red, y recibir sólo parte de los *frames* que se capturan.

Para compresión **MJPEG o JPEG,** no es necesaria fragmentación. El proceso de envío de *frames* es el siguiente.

Si hay uno o más clientes conectados, para cada conexión del *array* **`connections`**:

- se hace una comprobación del estado de cada *socket* del grupo de escritura **`wr`** (**`select()`** de la librería <*unistd.h*>)
- Se comprueba si está activa y si su **`fd`** pertenece al grupo de escritura **`wr`**.
- Se comprueba si el retardo impuesto a esta conexión (posición **`i`**) ha espirado (**`delay[i] = 0`**). Si ha expirado:
  - Se envía la imagen **`buf->data`** mediante la función **`write()`** de las librerías de *sockets*.
    - Si hay error, se elimina cliente con la función **`err_send()`** (pone **`active = 0`** para la conexión asociada a ese cliente y cierra *socket* UDP).
  - Se reinicia el contador de retardo, con el valor que se negocio para esa conexión. (**`delay[i] = connections[i].n.rate`**)
- Si el retardo impuesto a esta conexión no ha expirado todavía, se decrementa en uno, y no se envía imagen.

El ancho de banda que consume MJPEG es mucho menor que en el caso de RGB24, por ello no es necesario un retardo artificial en el envío de imágenes.

6.4.3.1.5 *close_UDP()*

```
int close_UDP(void *handle, int cs)
```

La función **`close_UDP()`** puede ser llamada tanto por el cliente como por el servidor. Si la llama el servidor, **`cs = SERVER`**. **`*handle`** sólo lo usa el cliente.

```
if (cs == SERVER)
      {
            for( i=0; i<MAX_CONNECTIONS - 1; i++)
            {
                  if (connections[i].active == 1)
                  close(connections[i].fd);
            }
      }
```

Se recorre el array **`connections`** y se cierran los *sockets* UDP de las conexiones activas.

6.4.3.2 Implementación de funciones cliente

Las funciones del controlador UDP relacionadas con el cliente son las siguientes.

6.4.3.2.1 *connect_UDP()*

```
int connect_UDP(void *handle, struct negotiation ng)
```

Con la llamada a la función **`connect_UDP()`** el cliente permite la conexión procedente del servidor.

Los valores resultantes de la negociación están guardados en **`ng`**, y **`*handle`** se guarda en **`*h`**, de tipo `struct` **`avi_handle`**.

Se obtiene el formato de video y compresión negociada de la estructura **`ng`**, para guardarlos en el manejador **`*h`**. Después se crea el **`socket()`** UDP cliente, guardando el descriptor **`fd`** en **`h->fd`**.

Como ya se explicó en el apartado 6.4.3.1.2, el cliente es quien se queda a la espera de recibir la conexión del servidor para recibir video (el servidor obtenía la dirección IP del cliente en la negociación TCP). El servidor conecta con el cliente a través de su función **`add_UDP_Client()`**. Para que pueda conectar, el cliente hace una llamada a la función **`bind()`** de *sockets*. Ver diagrama de secuencia de la figura 6.26

6.4.3.2.2 *recv_UDP()*

```
struct ng_video_buf* recv_UDP(void *handle){
      int size;
      size = recv_UDP_headers(handle);
      return  recv_UDP_Video(handle,size);
}
```

Llama a las funciones **`recv_UDP_headers()`** y **`recv_UDP_Video()`**. Devuelve una estructura de tipo **`struct`** `ng_video_buf*` que contiene el video que se ha recibido.

6.4.3.2.2.1 *recv_UDP_headers()*

```
int recv_UDP_headers(void *handle)
```

Se encarga de recibir señales de sincronismo y cabecera. Devuelve el tamaño del *frame* que se va a recibir.

En primer lugar intenta recibir la secuencia de sincronización “B” “B” “B” “B”. Posee un contador interno, por cada carácter “B” que recibe, lo incrementa en una unidad hasta que llega al valor cuatro. Si no se llega a ese valor y se recibe un carácter diferente a “B”, el contador se reinicia. Ver figura 6.17.

Una vez que el cliente se ha sincronizado, se procede a la recepción de cabecera de tipo **`Frame_Seq_HDR`** . Al igual que se obtuvo la hora local del servidor justo en el momento anterior de mandar la cabecera. El cliente obtiene su hora local en el instante que la recibe para poder calcular valores como retardo y *jitter* que se imprimen a través de la GUI por medio de la función **`message()`**.

Los relojes del SO cliente y servidor deben estar sincronizados. Puede usarse el protocolo **NTP** para sincronizarlos a través de un servidor de hora, por ejemplo *time.nist.gov*. Se diseñó un sistema de sincronización de relojes cliente/servidor. El cliente calculaba el retardo medio de transmisión entre cliente y servidor, mandándole un número determinado de mensajes. Cuando conocía ese retardo, pedía la hora al servidor, y se lo sumaba. El problema era que la hora cliente tenía un desfase de medio a un segundo respecto al servidor, y al trabajar con video en red local (y ser el retardo de transmisión muy pequeño) ese sistema de sincronización era inútil. Sincronizar relojes de dos ordenadores es realmente complicado.

Finalmente se devuelve el tamaño **`size`** que se ha recibido en la cabecera y que será el tamaño de la imagen que se va a recibir.

6.4.3.2.2.2 *recv_UDP_Video()*

```
struct ng_video_buf* recv_UDP_Video(void *handle, int size)
```

Recibe un *frame*. El tamaño del *frame* que se va a recibir es de tamaño **`size`**. **`*handle`** es de tipo **`avi_handle`** (apartado 6.4.2.3), de ella se obtiene el formato de video y la compresión, además de el descriptor de fichero asociado al *socket* UDP. El *frame* recibido se devuelve mediante una estructura de tipo **`struct`** `ng_video_buf`.

Se comprueba la compresión de video. Dependiendo del formato de compresión habrá que tener en cuenta si los paquetes van a llegar fragmentados o no.

Para compresión **RGB24** o **RGB15,** la imagen llega en varios paquetes.

```
struct iovec *cluster;
int packageSize = header->linesPerPackage * h->vfmt.bytesperline;

for (cluster = h -> vec, j = header->numberOfPackages-1; j >=0; j --){

      cluster->iov_base = ((unsigned char*)buf->data) + (packageSize * j);
      cluster->iov_len = packageSize;
            if (-1 == readv(h->fd,cluster, 1)){
                  printf("<MSG>Error: en la recepcion de video\n");
            }
}
```

El tamaño de paquete **packageSize** se calcula a partir del valor **linesPerPackage** que se ha recibido en la cabecera **header**, y los **bytesperline** del formato de video. El cliente conoce así el tamaño de cada paquete que va a recibir.

El proceso de recepción de los paquetes y el "ensamblado" de la imagen es casi idéntico al del servidor. La diferencia está en que mientras el servidor obtiene los fragmentos de video desde su memoria para después enviarlos con la función **writev()**, el cliente los recibe mediante la función **readv()** y los guarda en memoria. Los guarda siguiendo la misma secuencia que el servidor usa para leerlos (ver figura 6.15).

El proceso es análogo al del servidor (ver el fragmento de código anterior), el bucle **for** se repite tantas veces como número de fragmentos (o de paquetes) se reciben. Para cada bucle se recibe un fragmento. Por ejemplo, si una imagen está fragmentada en 3 paquetes (el número de paquetes se indican en la cabecera **header->numberOfPackages**), en el primer bucle la variable **j = 2**. Así, la dirección de inicio **cluster->iov_base** apunta a la posición de memoria donde se va a guardar la imagen más un desplazamiento en *bytes*, y **cluster->iov_len** indica cuantos *bytes* deben obtenerse para ser escritos en memoria (se obtiene todo el paquete).

La primera vez, el desplazamiento será:

packageSize * (j = 2) = desplazamiento equivalente al tamaño de dos paquetes

y se guardan los **packageSize = cluster->iov_len** *bytes* siguientes. Figura 6.22. Como se observa en la figura, este paquete puede contener *bytes* "basura" (los paquetes son de tamaño fijo), pero más tarde *xawtv* los descarta, pues conoce el tamaño **size** de la imagen.

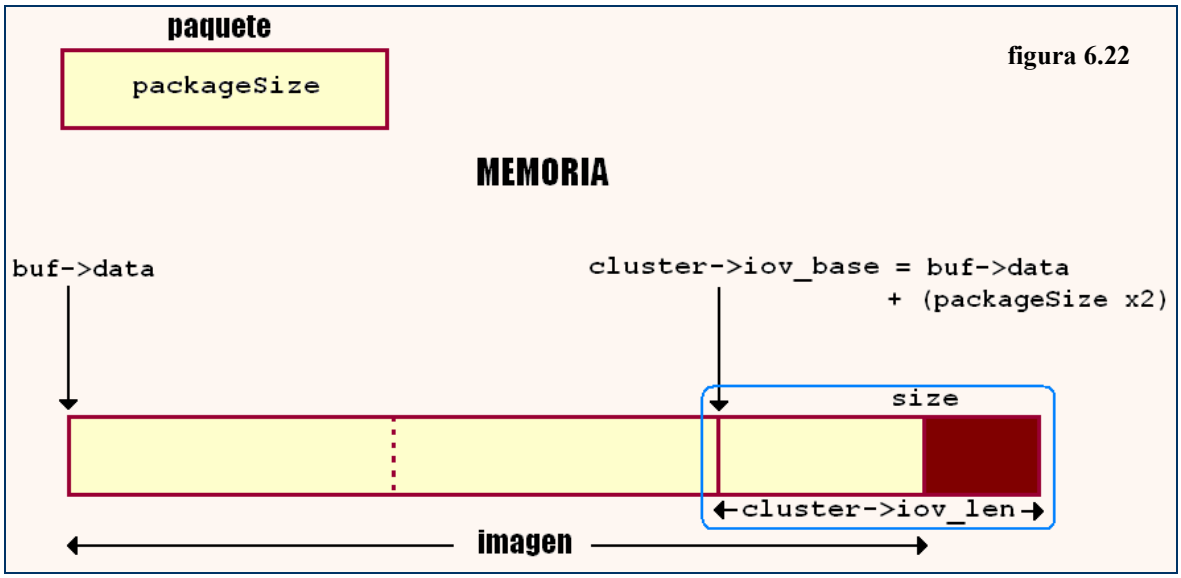


figura 6.22

En el siguiente bucle, j=1:

`packageSize` * (j = 1) = desplazamiento equivalente al tamaño de un paquete

y se guardan los **`packageSize = cluster->iov_len`** *bytes* siguientes. Figura 6.23.

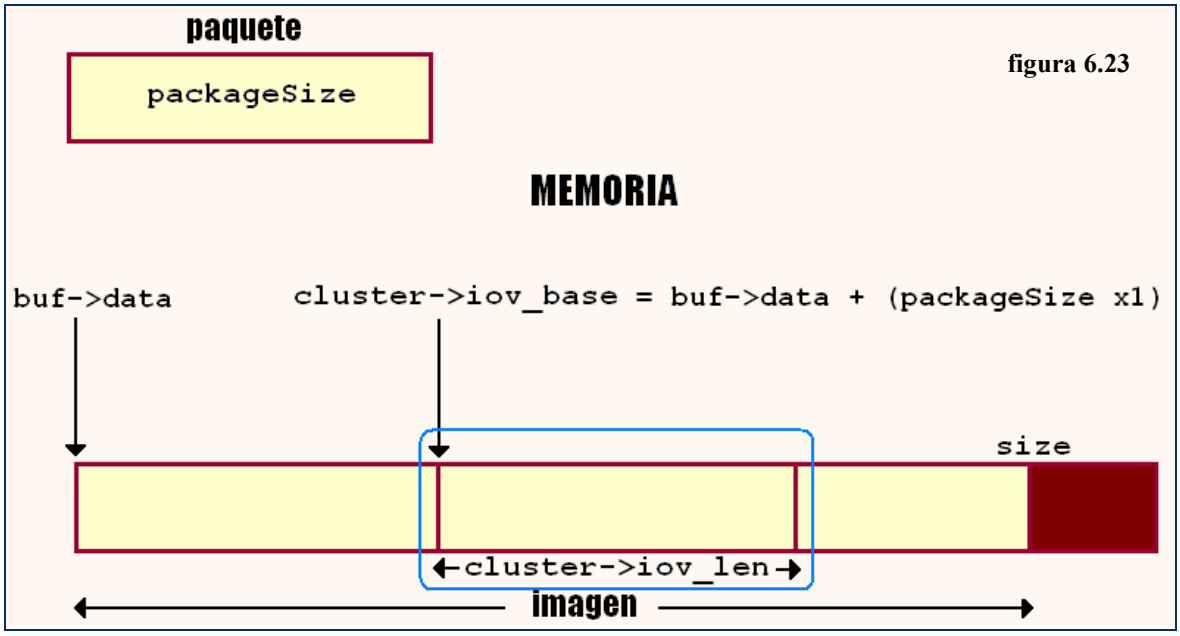


figura 6.23

En el siguiente bucle, j=0:

`packageSize` * (j = 0) = sin desplazamiento

y se guardan los **`packageSize = cluster->iov_len`** *bytes* siguientes. Figura 6.24.

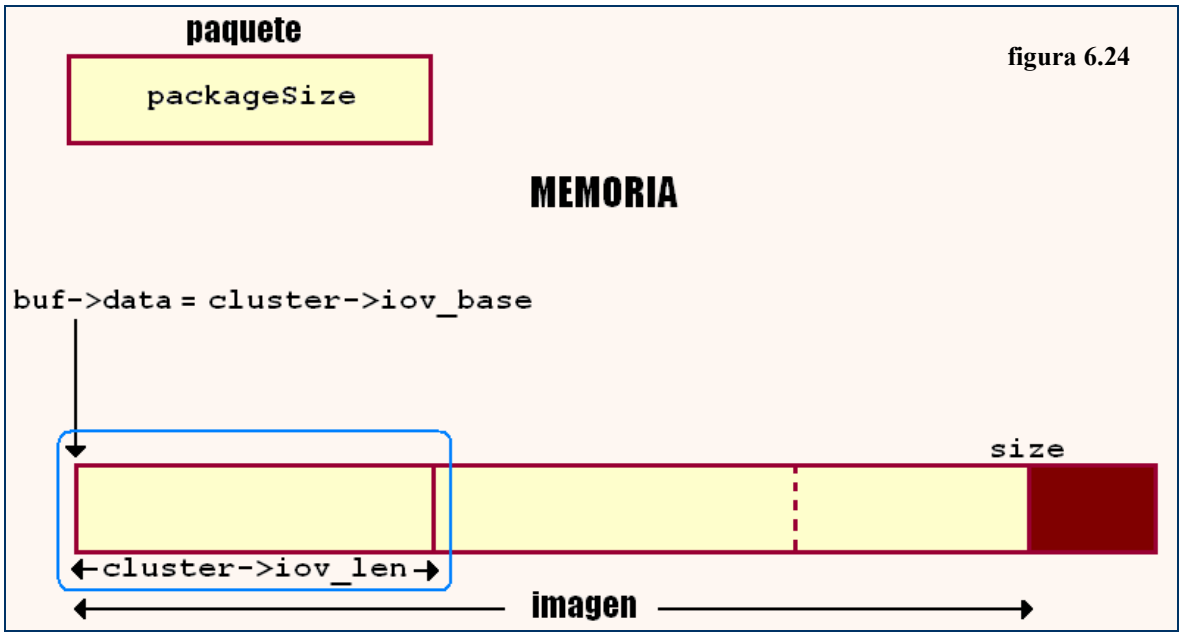


figura 6.24

El orden en el que se guardan los fragmentos se realiza agrupando los *bytes* empezando por el **final** de la imagen (porque así es como se envían). Esta es la forma en la que *xawtv* guarda una imagen en un fichero de video, y ha sido adaptada para la recepción. Los fragmentos se reciben y se guardan de uno en uno mediante la función **`readv()`**.

Para compresión **MJPEG o JPEG,** los paquetes no llegan fragmentados.

```
read(h->fd, buf->data, size)
```

Se reciben mediante la función **read()** de las librerías *socket*. **`h->fd`** contiene el descriptor de fichero asociado a la conexión con el servidor, en **`buf->data`** se guarda el video recibido, y **`size`** es el tamaño de la imagen que se va a recibir.

Para ambos casos (RGB24 y MJPEG) se devuelve la estructura **`struct`** `ng_video_buf` **`*buf`** que contiene el video.

6.4.3.2.3 c*lose_UDP()*

```
int close_UDP(void *handle, int cs)
```

La función **`close_UDP()`** cuando es llamada por el cliente, **`cs = CLIENT`**. **`*handle`** contiene el descriptor de fichero *socket* asociado a la conexión UDP.

```
if (cs == CLIENT) {
      close(h->fd);
      free(buffer);
}
```

Se cierra el *socket* **`h->fd`**, y libera memoria.

6.4.4 Diagramas de despliegue y de secuencia

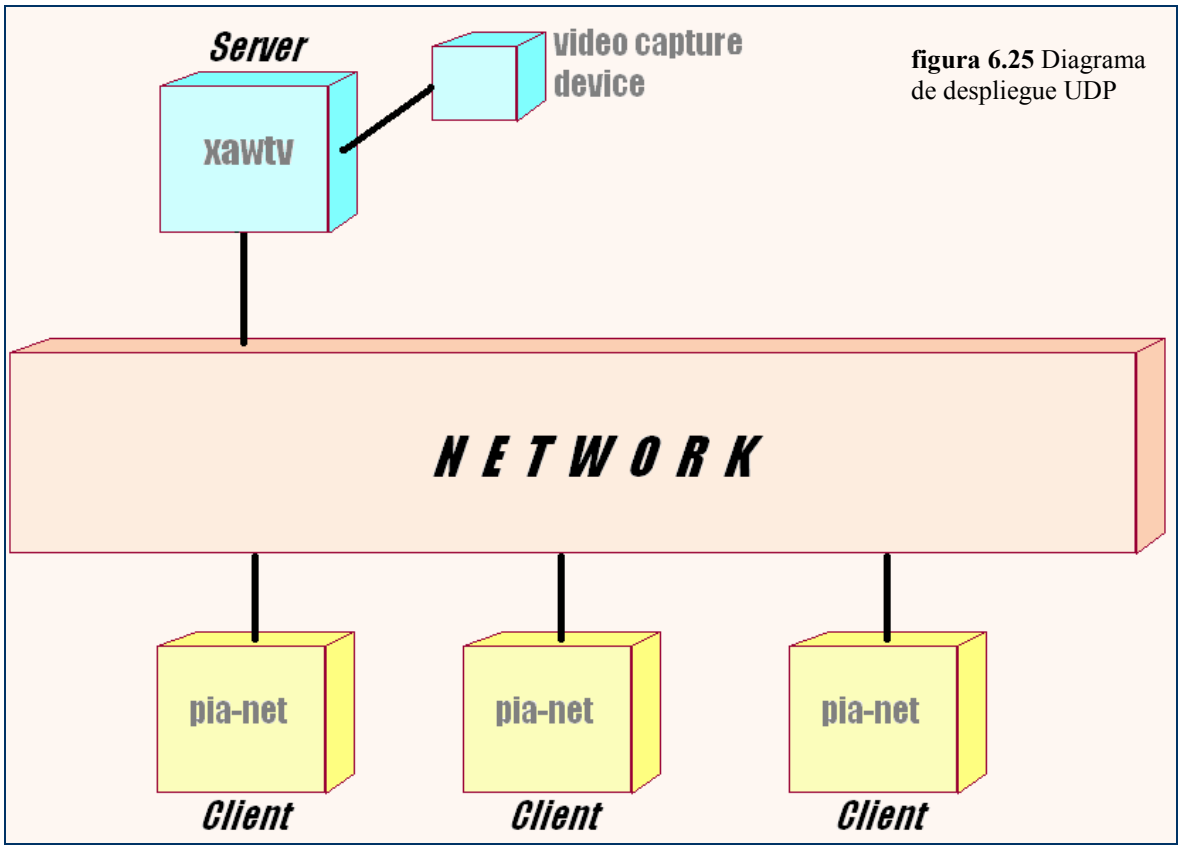


figura 6.25 Diagrama de despliegue UDP

NOTA: En el diagrama de secuencia la figura 6.26, se observa el proceso de conexión y desconexión de un único cliente para ilustrarlo de una forma más clara (además se omite el proceso de negociación que se encuentra en la figura 4.15). El servidor UDP es capaz de manejar varios clientes al mismo tiempo.

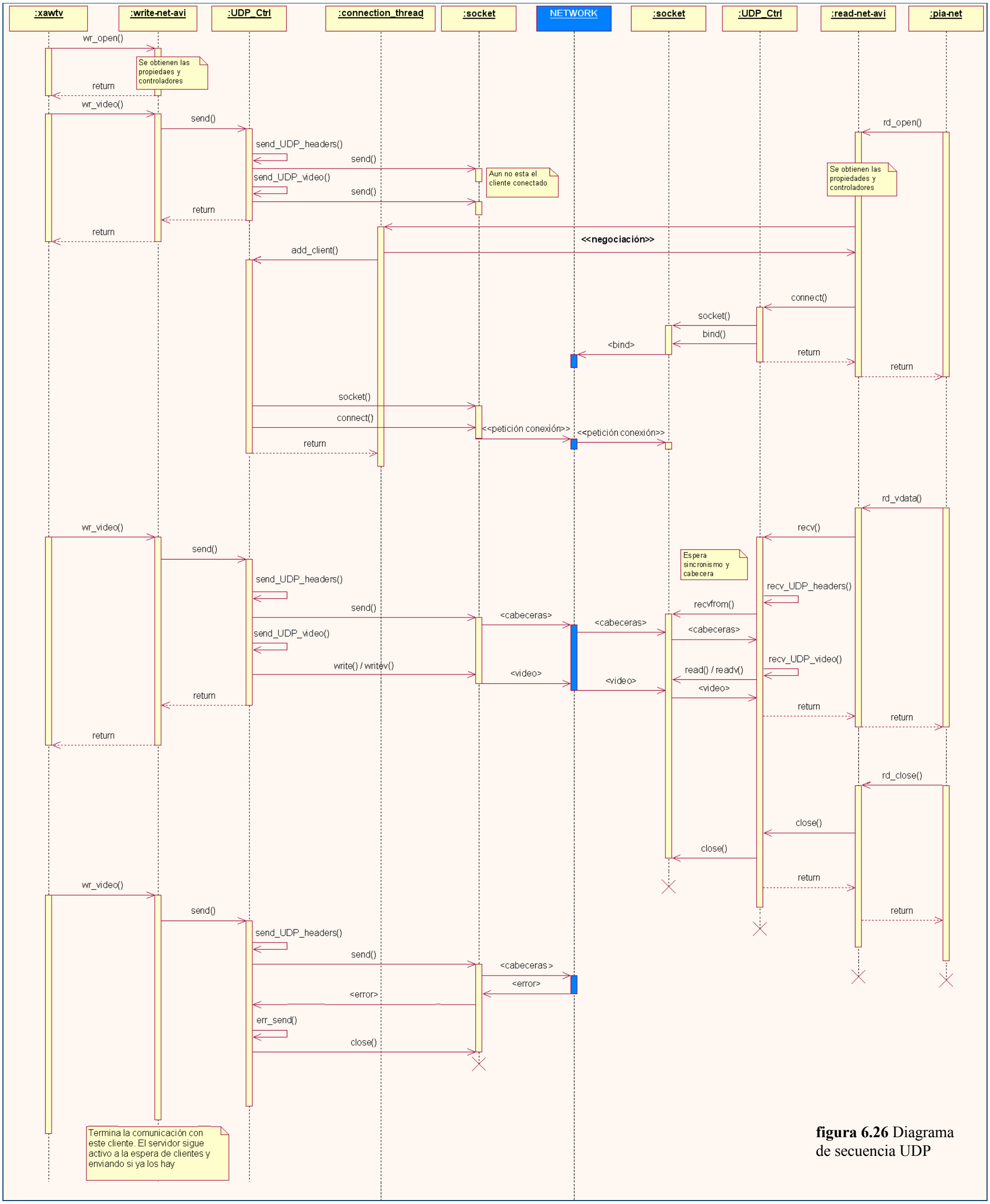


figura 6.26 Diagrama de secuencia UDP

7. Controlador para AVStreams CORBA

7.1 Introducción

En la actualidad se demanda interoperabilidad entre la gran variedad de distintas tecnologías software y hardware existentes . Por este motivo nace CORBA implementado por OMG **[14]** (*The Object Management Group)*.

The Common Object Request Broker Architecture (CORBA) es una tecnología *Middleware* desarrollada para permitir la interacción entre distintos lenguajes de programación, sistemas operativos ... Permite a las aplicaciones comunicarse unas con otras sin importar donde están ni como están implementadas. Para ello, CORBA define un lenguaje, IDL (*Interface Definition Language*), y unas interfaces, API (*Application Programming Interfaces*). Esto permite al objeto cliente/servidor interaccionar con una implementación específica de un ORB (*Object Request Broker*), que es la estructura básica de comunicaciones en CORBA basada en llamadas a métodos remotos por medio de intermediarios. Estos intermediarios son el *stub* y *skeleton*. El cliente realiza llamadas remotas haciendo uso del objeto *stub* y el servidor las recibe, interpreta y contesta por medio del *skeleton*.

Existen muchas implementaciones de ORB dependiendo del tipo de uso que se le desee dar. Para el caso de transmisión de datos en tiempo real existe Real-Time CORBA. Hay varios ORB implementados en distintos lenguajes de programación que cumplen dicha especificación. Entre ellos se puede encontrar TAO (*The ACE ORB*).

7.2 Adaptative Communication Environment (ACE). The Ace ORB (TAO)

ACE **[15]** se compone de un gran conjunto de utilidades de programación en lenguaje C++ disponibles tanto en *Linux* como en *Windows*. Es un *framework* de libre distribución y código abierto. Simplifica el desarrollo de aplicaciones de comunicación en tiempo real.
Provee un sistema de comunicación entre procesos, manejo de eventos, enlace dinámico de librerías y concurrencia. Otra gran cualidad de ACE es la automatización de configuraciones de sistema, y su reconfiguración, mediante servicios de enlace dinámico en aplicaciones en tiempo de ejecución y el procesado de dichos servicios en uno o más *threads*. Estas facilidades eliminan carga de trabajo al programador.

Dentro de las características que definen ACE, podemos encontrar un incremento de la portabilidad y de la calidad del software mediante la mejora de características como modularidad, flexibilidad ... y una transición más fácil a un *standard* de *middleware* de alto nivel. Dicha transición ofrece componentes reutilizables y patrones útiles para TAO.

TAO **[16]** es la implementación de la especificación de *Real-Time CORBA* ofrecida dentro de las utilidades de ACE. Tiene cuatro componentes básicos:

IDL.Compiler. Genera los *stubs* y *skeletons*, necesarios para las llamadas remotas a métodos.

Inter-ORB Protocol Engine. Ofrece una capacidad de realizar operaciones entre ORBs que usen el protocolo *standard* IIOP. Permite tanto modelos estáticos (SII/SS) como dinámicos (DII/DSI) de programación en CORBA.

ORB Core. Permite comunicaciones unidireccionales, bidireccionales y comunicaciones fiables unidireccionales, tanto síncronas como asíncronas. También permite varios modelos de concurrencia.

Portable Object Adapter **(POA)**. Diseñado usando patrones que ofrecen un conjunto de estrategias (request demultiplexing strategies) útiles para la detección de referencias de objetos, tanto persistentes como transitivos.

Al ser TAO una implementación de *Real-Time CORBA*, se ofrecen los siguientes servicios:

1. Servicio de flujos de Audio/Video
2. Concurrencia
3. Eventos
4. Ciclo de vida
5. Servicio de control de identificación (*logging service)*
6. Servicio de nombrado
7. Servicio de notificación
8. Servicio de estado persistente
9. Servicio de propiedades
10. Servicio de seguridad
11. Servicio de tiempo
12. Servicio de intercambio (*Trading service*)

Fuera de las especificaciones de CORBA se ofrecen los siguientes servicios:

1. Servicio de balance de carga
2. Servicio de eventos en tiempo real (*Real-time Event*)
3. Servicio de programación (*Scheduling Service*)

Hay disponible una documentación no gratuita, de la que no se disponía para el desarrollo del proyecto, en la que todos estos servicios están descritos con detalle.

7.3 La Estructura de TAO

Como ya se mencionó en el apartado anterior, TAO ofrece una gran cantidad de servicios, que están situados dentro de las librerías en diferentes directorios. Estos servicios se encuentran dentro del directorio *orbsvcs*. Este se encuentra a su vez dentro del directorio raíz de TAO.

La lista completa de servicios y directorios es la siguiente:

Servicio	Directorio
A/V Streams Service	orbsvcs/AV
Concurrency Service	orbsvcs/Concurrency
Event Service	orbsvcs/CosEvent
Real-time Event Service	orbsvcs/Event
LifeCycle Service	orbsvcs/LifeCycle
Load Balancing Service	orbsvcs/LoadBalancing
Logging Service	orbsvcs/Log
Naming Service	orbsvcs/Naming
Property Service	orbsvcs/Property
Scheduling Service	orbsvcs/Sched
Security Service	orbsvcs/Security
SSLIOP Pluggable Protocol	orbsvcs/SSLIOP
Trading Service	orbsvcs/Trader
Time Service	orbsvcs/Time
Notification Service	orbsvcs/Notify

En las versiones actuales de TAO, existen una serie de ficheros binarios (ejecutables) para algunos servicios. Estos también se encuentran dentro de *orbsvcs* y son los siguientes:

- Concurrenty Service
- Dump Schedule
- LifeCycle Service
- Load Balancer
- CosEvent Service
- Event Service
- Naming Service
- Scheluding Service
- Trading Service
- Time Service
- Notify Service
- ImplRepo Service

Tras este repaso de las posibilidades que ofrece TAO, se realiza a continuación una explicación más detallada de los servicios mínimos necesarios para la transmisión de audio/video: *Naming Service y AVStreams Service* **[17]**.

NOTA: Esta explicación se ha realizado al margen de la documentación oficial de TAO, ya que ésta es de pago y no se dispone de ella.

7.3.1 Naming Service

Como ya se explico anteriormente, es un archivo ejecutable que se puede encontrar dentro de las librerías de TAO. Su situación, a partir del directorio raíz de TAO, es *orbsvcs/Naming_Service/Naming_Service.*

Permite a un objeto registrarse en él y así otros poder acceder a dicho objeto.

Al registrarse, se guarda una referencia al objeto dentro del servidor de nombres por medio de la cual podrá ser localizado por cualquier cliente del servidor de nombres (TAO_Naming_Client) que acceda y use esa referencia. Este servicio tiene un comportamiento similar a un sistema de ficheros normal. Tiene una raíz de la que parten contextos, equivalentes a lo que serían directorios. Por esto, un objeto, sería equivalente a un archivo, se puede registrar dentro del contexto raíz o dentro de otros contextos. A su vez, los contextos pueden contener subcontextos, también con objetos.

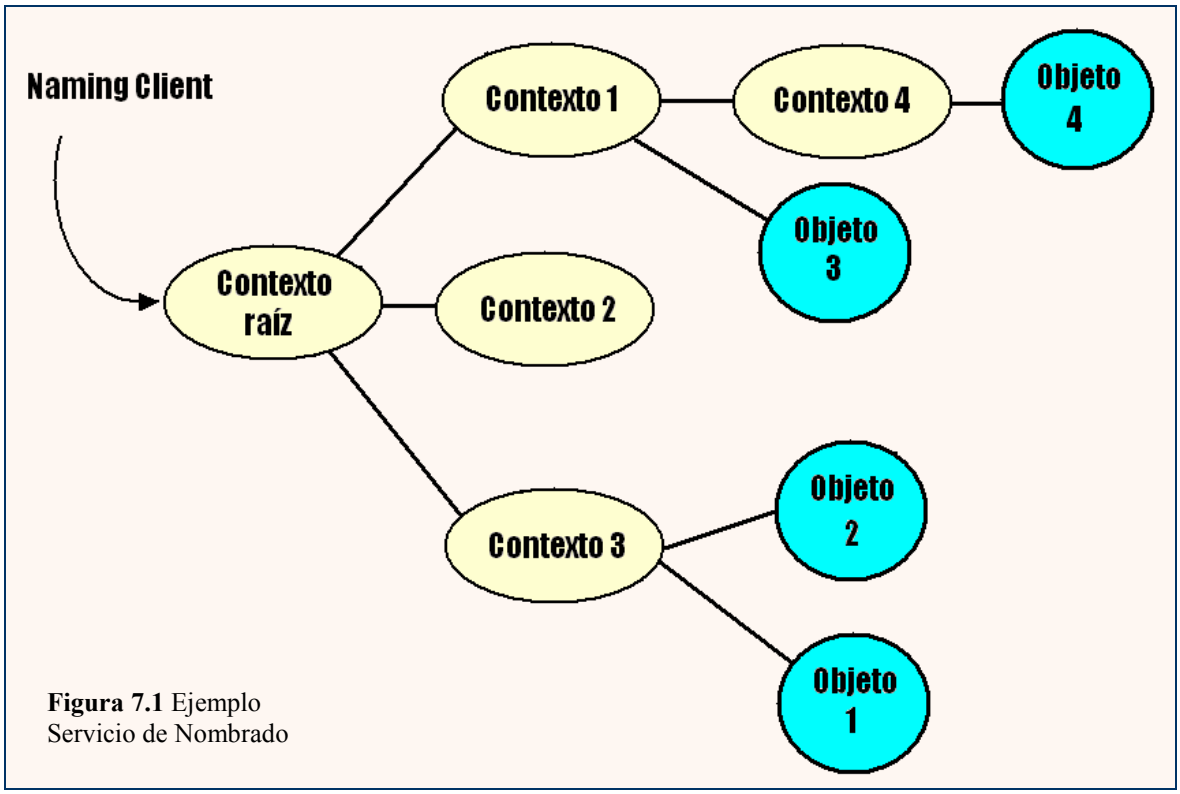


Figura 7.1 Ejemplo Servicio de Nombrado

El Servicio de Nombrado puede ser ejecutado con distintas opciones. Una de dichas opciones es la de ejecutar el servicio en un fichero o en un puerto.

Si el Servicio de Nombrado se ejecuta en un fichero, podrán acceder a él cualquiera que tenga acceso al fichero. Se deben tener en cuenta permisos de lectura/escritura. En cambio, si este se ejecuta en un puerto, podrá tener acceso cualquiera que tenga acceso a esa dirección y a ese puerto. Dependiendo de la forma de ejecución del Servicio de Nombrado, los clientes accederán a él haciendo uso de distintas opciones del ORB.

Para ejecución en un fichero:

```
[dir_raíz_de_TAO]/orbsvcs/Naming_Service/Naming_Service -o [fichero]
```

En este caso, *fichero* será el fichero donde se desea ejecutar el Servicio. Para localizar el servicio de nombrado en este tipo de ejecución:

```
[archivo_ejecutable] -ORBInitRef NameService=`cat [fichero]   ó

[archivo_ejecutable] -ORBInitRef NameService=file://[fichero]
```

Archivo_ejecutable sería cualquier ejecutable que quisiera tener acceso al Servicio de Nombrado pasando esa referencia al ORB como argumento. El uso de cualquiera de estas dos sintaxis es correcto.

Para ejecución en un puerto:

```
[dir_raíz_de_TAO]/orbsvcs/Naming_Service/Naming_Service
                      -ORBEndPoint iiop://[dir_IP]:[puerto]
```

Donde dir_IP es la dirección IP de la máquina donde se encuentra el Servicio de Nombrado y puerto, el puerto por el que se accederá a dicho servicio. Existen dos formas de acceder:

```
[archivo_ejecutable] -ORBInitRef
               NameService=corbaloc:iiop:[dir_IP]:[puerto]/NameService

[archivo_ejecutable] -ORBDefaultInitRef iiop://[dir_IP]:[puerto]
```

Por motivos desconocidos, algunas veces es válida la primera y otras la segunda.

Existen gran cantidad de opciones para el Servicio de Nombrado, además de las mencionadas anteriormente. Para ver dichas opciones y una explicación detallada de las mismas, ver [dir_raíz_de_TAO]/orbsvcs/Naming_Service/README.

7.3.2 AVStreams Service

Aporta una gran cantidad de utilidades para la transmisión de Audio y Video. En este punto no se desarrollarán todas las capacidades de AVStreams, simplemente se dará un punto de vista general, aunque suficiente.

Algo importante a comentar, es que la forma tradicional de transmisión de datos usando tecnologías de objetos distribuidos requiere unas cabeceras de tamaño considerable y hace uso de algoritmos de retransmisión de datos. Como ya es sabido de puntos anteriores, se debe de prescindir de muchos servicios, entre ellos retransmisión de datos, y aligerar el tamaño de cabeceras. Por ello, AVStreams se implementan de una forma algo atípica, dentro de lo que a tecnologías distribuidas se refiere, ya que se suele usar UDP y se aligera el tamaño de las cabeceras. El protocolo es un parámetro a definir.

La especificación de AVStreams dada por la OMG, define un conjunto de interfaces IDL Standard que pueden ser implementadas. Esto es lo que hace TAO.

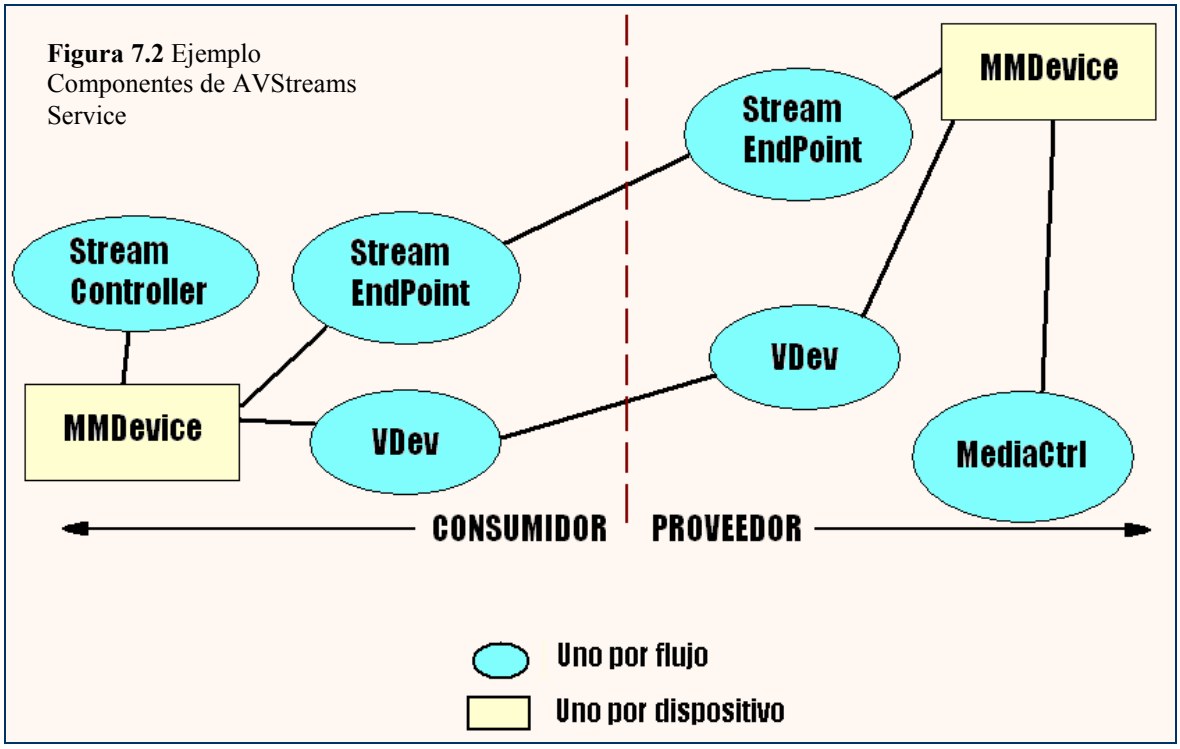


Figura 7.2 Ejemplo Componentes de AVStreams Service

7.3.2.1 Multimedia Device Factory (MMDevice)

Abstrae el comportamiento de un dispositivo de captura de datos multimedia. Los dispositivos pueden ser tanto físicos, como cámaras, micrófonos... Estos dispositivos también pueden ser lógicos, como un programa que lea video de un archivo o de alguna base de datos.

El objeto MMDevice encapsula también los parámetros específicos del dispositivo. Estos parámetros son denominados propiedades de MMDevice y pueden ser usados por medio del Servicio de Propiedades (*Property Service*).

También es responsable de crear nuevos *EndPoints* para nuevas conexiones.

EndPoints. Consisten en un par de objetos:

- ***Virtual Device*** (**VDev**), que encapsula los parámetros de la conexión. Se verá más a fondo en el apartado 7.3.2.2.
- ***StreamEndPoint***, que encapsula los parámetros de transporte específicos de la conexión. Ver apartado 7.3.2.5.

MMDevice también puede usar **estrategias** para la creación de este par de objetos. Las hay de dos tipos:

- ***Process-based strategy***. El par de objetos es creado en un proceso a parte.
- ***Reactive strategy***. El par se crea en el mismo proceso donde se encuentra el MMDevice. Útil para cuando es un sólo proceso el que se encarga de controlar todos los flujos.

También hay estrategias específicas, de cada uno de los dos tipos anteriores, para objetos de tipo receptor y de tipo emisor.

7.3.2.2 Virtual Device (VDev)

Es el componente creado en respuesta a una nueva conexión sobre un flujo. Hay uno por flujo. Como ya se ha dicho, encapsula los parámetros de una conexión, por lo que está totalmente relacionado con las propiedades sobre ésta. Cuando se quiere establecer una propiedad específica, por ejemplo formato de video o un tipo de compresión, se invoca a su método **`configure()`**.

Esto sería útil para un proceso de negociación de propiedades de conexión. La secuencia sería:

- Cada extremo tiene establecidas sus propiedades, usando sus respectivos *VDev*. Durante el establecimiento del flujo, cada *Virtual Device* puede invocar el método **`get_property_value()`** para confirmar que cada uno de los extremos comparten las mismas propiedades necesarias para la comunicación.
- Cuando el nuevo par de **VDev** ha sido creado, cada uno usa su método **`configure()`** para establecer los parámetros de la comunicación.
- Si la negociación falla, los recursos se liberan inmediatamente.

7.3.2.3 Media Controller (MediaCtrl)

Consiste en una interfaz IDL que define las operaciones que controlan el flujo. Esta interfaz es implementada por el programador.

7.3.2.4 Stream Controller (StreamCtrl)

Componente encargado de controlar la transferencia continua de audio/video entre dos *Virtual Devices*. Puede realizar la operación **`bind()`** entre dos *MMDevices*, esto es, puede establecer una conexión entre dos *MMDevices*, uno de ellos es la fuente y otro es el destinatario del flujo multimedia. Esta interfaz también provee métodos para parar, pausar y arrancar el flujo, además de poder especificar parámetros de calidad de servicio (QoS).

Hay un *StreamCtrl* por flujo.

7.3.2.5 Stream EndPoint

Es el encargado de encapsular los parámetros relacionados con el transporte de la información. Establece, entre otros parámetros, el protocolo de transporte a usar, el nombre de *host* y el puerto para identificar la conexión.

7.3.2.6 Pluggable Protocols

A la hora de establecer un protocolo de transporte (*Transport Protocol*) en un flujo de AVStreams hay dos opciones, una es usar las herramientas que se nos ofrece y otra es crear nuestro propio *pluggable protocol* a partir del *framework* ofrecido por TAO. Se recuerda que la transmisión se suele realizar sobre UDP, pero ésto es parametrizable, también se podría usar TCP o RTP.
Una buena forma de lograr entender como se realiza la transmisión de datos multimedia a través de AVStreams es observar los componentes dentro del apartado *Transport Protocols* y *Flow Protocol*.

7.3.2.6.1 Componentes de Transport Protocols

Acceptor y Connector [7], ya implementados para varios protocolos de transporte como son TCP, RTP y UDP. Indica que se hará en caso de realizar la operación de conectar y de aceptar, por ejemplo, la creación de manejadores. En el caso de UDP, los manejadores se crean inmediatamente después de la conexión.

Transport Factory, ofrece la interfaz común para todos los protocolos con operaciones como **`open(), close(), send() y recv()`**.

Flow handlers, todos los manejadores de los protocolos de transporte (*transport handlers*) derivan de TAO_AV_Flow_Handler. Dentro de su interfaz hay definidos métodos **`start()`** y **`stop()`.** También son usados por las funciones de Callback, por ejemplo estableciendo *timeouts*.

7.3.2.6.2 Componentes de Flow Protocols

Flow Protocol Factory, interfaz para la creación de *flow protocols objects*.

Protocol Object, es una implementación concreta de la funcionalidad de *flow protocol* usando un protocolo específico (UDP, RTP o TCP). Las aplicaciones utilizan su interfaz para enviar *frames,* mediante la función **`send_frame()`**, y éstos se reciben por medio de los objetos de Callback.

7.3.2.7 Bloques de Mensaje. ACE_Message_Block

Dentro de las librerías de AVStreams se puede encontrar un tipo de objeto llamado **`ACE_Message_Block`**. Esta clase de objeto contiene un *array* de caracteres, que contiene la información a enviar, variables extra para información sobre los datos y un conjunto de métodos útiles para el manejo de los datos y dichas variables.

7.3.2.8 Funciones de Callback. TAO_AV_Callback

Es una interfaz a implementar por el programador. Las funciones de *Callback* son manejadoras de eventos. En especial, los métodos de **`TAO_AV_Callback`** han de ser implementados para responder a los eventos de recepción de *frames* e incluso *timers*. Por ejemplo, los métodos **`receive_frame()`** y **`send_frame()`**, ver apartado 7.3.2.6.2, están relacionados entre si. **`receive_frame()`** ha de ser implementado para poder recibir los *frames* de **`send_frame()`**.

7.4 Introducción a la creación del controlador AV

El propósito de la creación de un controlador basado en flujos de audio y video de TAO (*AVStreams*), es el aprendizaje de como las tecnologías *Middleware* pueden ser aplicadas en este campo. Es un estilo de programación totalmente distinto al explicado anteriormente en el apartado 6. Una complicación añadida fue la integración de una parte importante del controlador AV implementada en C++**[15]**, ya que ACE/TAO así lo requiere.

El primer paso a realizar fue el establecimiento de una conexión punto a punto entre el cliente y el servidor. En este caso solo se hacia uso de un cliente, un servidor y el servicio de nombrado de TAO. Una limitación fue la necesidad de que el cliente estuviera conectado antes que el servidor. Esto se debe a que por debajo del nivel de aplicación, los papeles de servidor y cliente se intercambian.

Este problema se eliminó mediante el uso de un Distribuidor. Esta inclusión permitió automáticamente que el servidor pudiera funcionar indefinidamente junto con dicho Distribuidor y que éste aceptara conexiones procedentes de distintos clientes. Así apareció la necesidad de una desconexión correcta de dichos clientes, sin romper la estabilidad del Servicio de Nombrado.

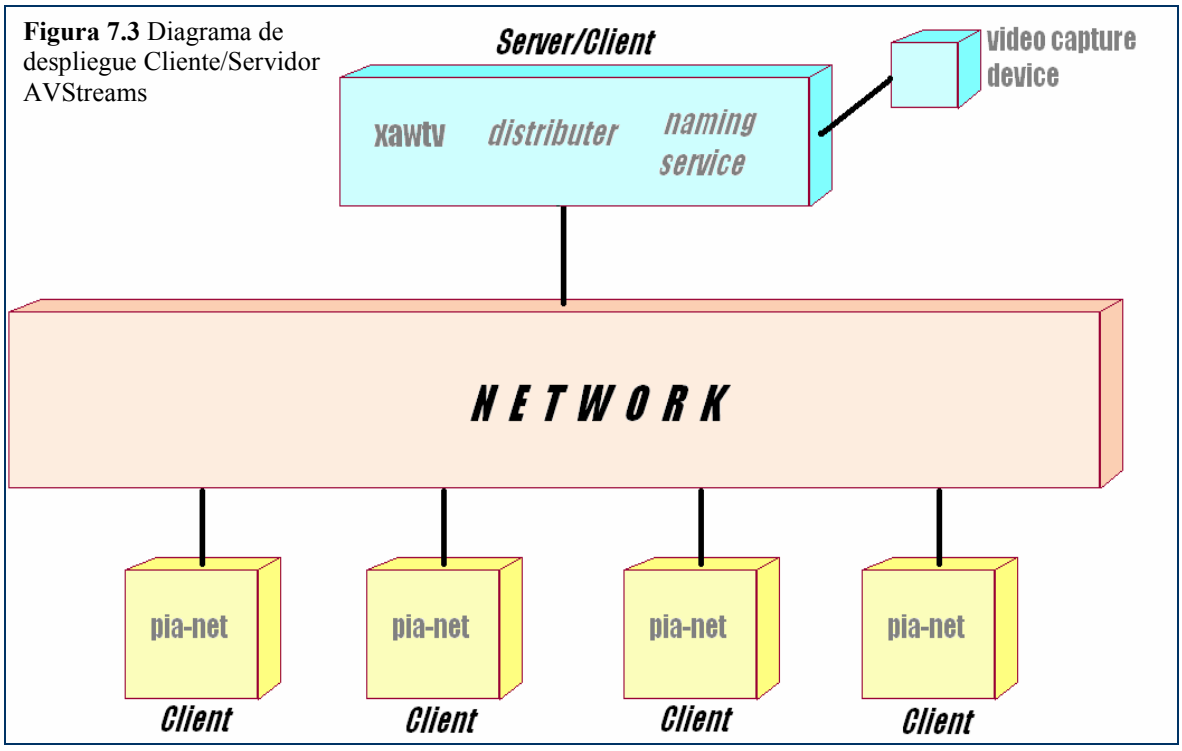


Figura 7.3 Diagrama de despliegue Cliente/Servidor AVStreams

Se debió de tener en cuenta la eliminación de la negociación TCP para la obtención de los parámetros de transmisión de video. Esto permitirá en un futuro implementar la negociación de las propiedades por medio de los servicios implementados específicamente para ello en *AVStreams* (*Property Service y VDevs*), aunque introduce en cliente el problema de desconocer ciertos parámetros de video (compresión y formato de video).

Finalmente se agrupó el código de forma que pudiera implementar la interfaz controlador. Se automatizó el arranque de los procesos Distribuidor y *Naming Service* desde el propio *xawtv*.

La mayor dificultad que se encontró en el desarrollo de este controlador fue la utilización, como ya se ha mencionado varias veces, de *AVStreams*. Esta dificultad venía dada en gran medida por la falta de documentación.

7.4.1 Un servidor, varios clientes.

La problemática propuesta al inicio del apartado 6.3.3 queda directamente resuelta mediante el uso del Servicio de Nombrado. Ya no es necesaria la implementación de un *thread* encargado del establecimiento de la comunicación inicial con los clientes. En cambio, la idea persiste. En vez de implementar un *thread*, se lanza un proceso encargado de ejecutar el Servicio de Nombrado de TAO. Los clientes se registrarán en el y así podrán ser detectados.

Como ya se ha comentado, no es aconsejable usar simplemente un servidor. Por ello aparece un proceso Distribuidor, encargado de la interacción directa con los clientes. Este proceso puede estar en la misma localización física que el servidor de video o no. En nuestro caso ambos procesos están funcionando en el mismo equipo.

El proceso distribuidor detecta y establece conexiones con nuevos clientes por medio del Servicio de Nombrado y de cierta estrategias, ya vistas en el apartado 7.3.2.1. De la misma forma detecta el verdadero servidor de video al Distribuidor, aunque su creación y destrucción sean simultaneas.

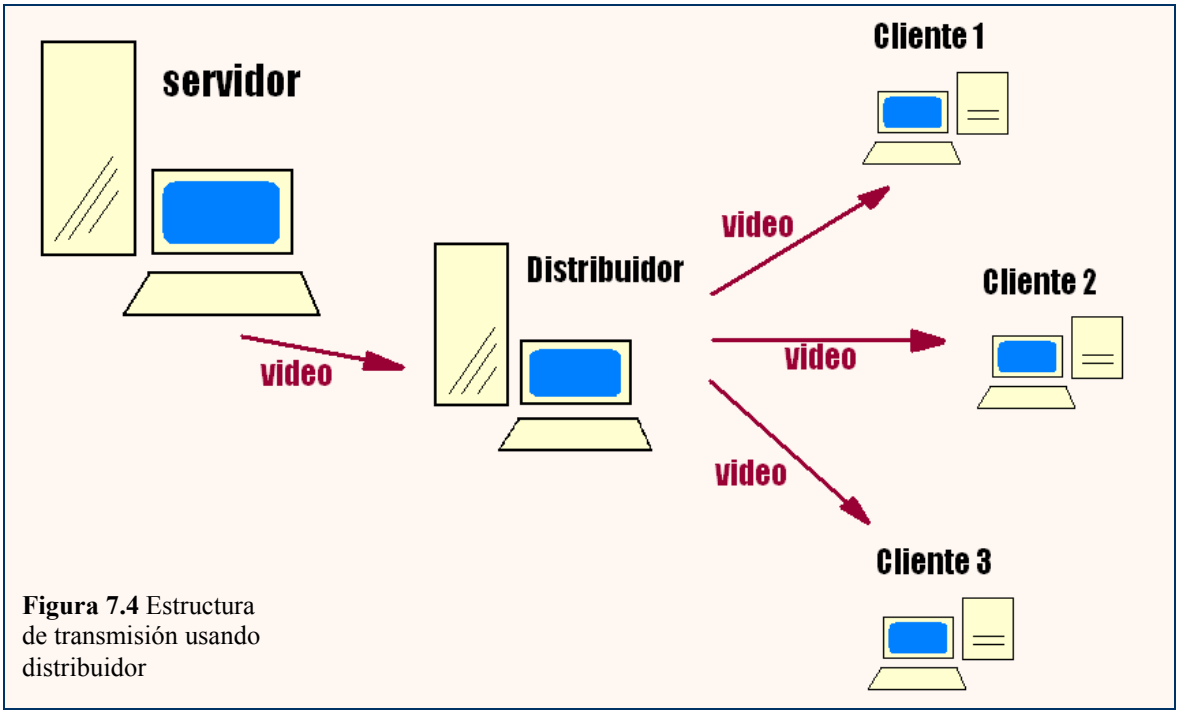


Figura 7.4 Estructura de transmisión usando distribuidor

El establecimiento y libertación de conexiones se facilita mediante una buena ordenación de los objetos dentro del Servicio de Nombrado. Dentro del contexto principal del Servicio de Nombrado se crearán los contextos de los emisores. Estos albergarán un objeto emisor y otro contexto donde se almacenarán todos los receptores de dicho objeto. Haciendo uso de esta estructura junto con las estrategias y clases de *Callback*, los

objetos emisores, **`sender`** y **`distributer`**, consiguen crear una lista de receptores de forma local.

Otra parte importante es diferenciar los diferentes flujos de transmisión de video. Esto se logra por medio de una cadena de texto asociada a los flujos llamada **`flowname`**. La nomenclatura para un **`flowname`** será **`emisor_receptor`**. Así, por ejemplo, el flujo entre el objeto **`distributer`** y el objeto receptor **`recv-2`** será **`distributer_recv-2.`**

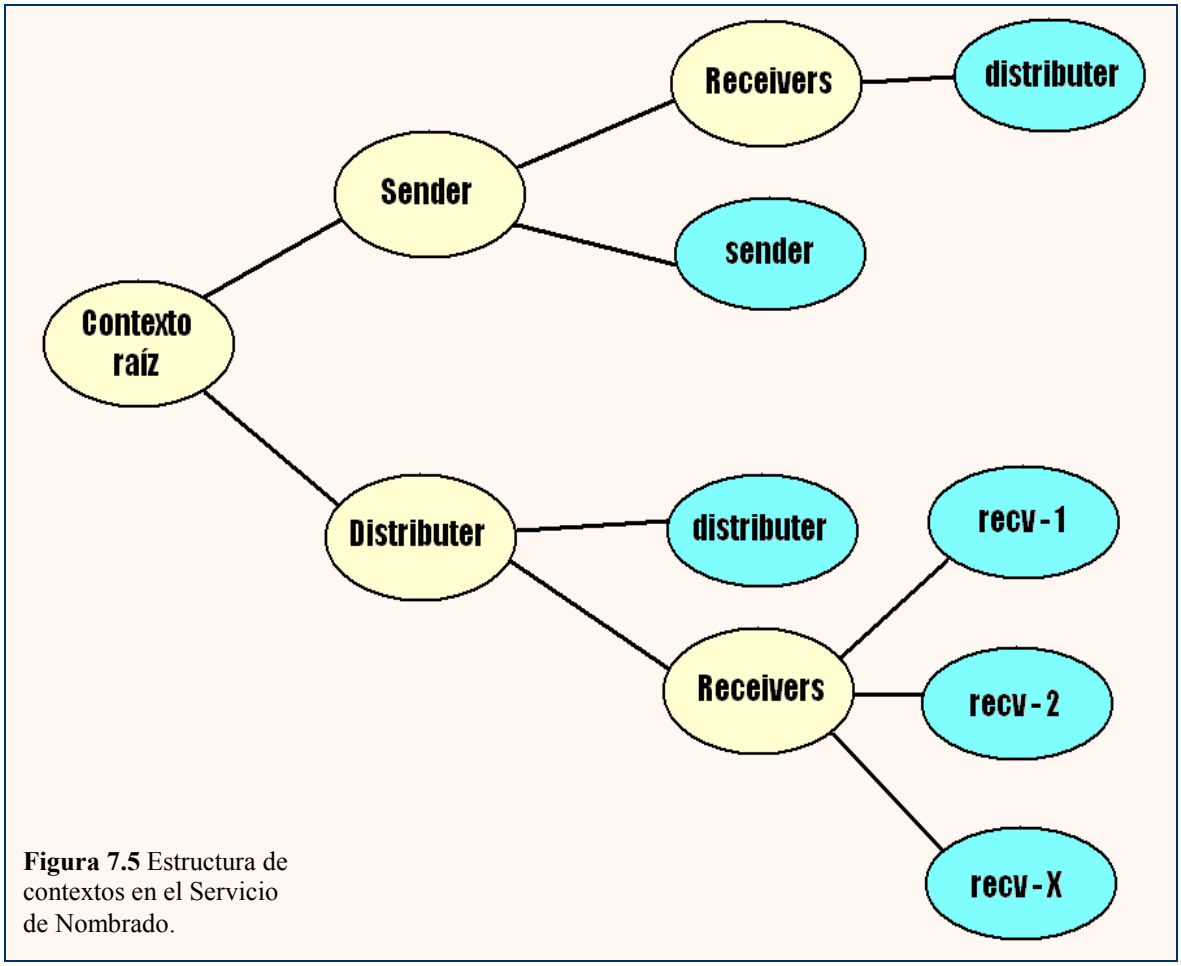


Figura 7.5 Estructura de contextos en el Servicio de Nombrado.

7.4.2 Desconexión

Siempre es un punto importante a tener en cuenta. Una mala gestión del estado de los clientes puede llevar a la inestabilidad del servidor.

En la estructura usada, se debe tratar cuidadosamente el estado del Servicio de Nombrado. En el caso de que un cliente se desconectara por cualquier motivo y no fuese debidamente eliminado del servicio de nombrado, no podría volver a conectar, al menos usando el mismo nombre. La única solución sería reiniciar el Servicio de Nombrado, eso si no fue lanzado en modo persistente en un fichero, porque en este caso seguirían registrados todos los objetos aún reiniciando. Aún cuando fuera eliminada correctamente la referencia del cliente en el Servicio de Nombrado, si no se elimina la información de

éste dentro del distribuidor, tampoco podría reconectar ya que quedarían residuos de dicha conexión.

El procedimiento propuesto es el siguiente. Ya que el distribuidor es el encargado de detectar nuevos clientes y enviarles video, éste también deberá detectar cuando un cliente se ha desconectado. Para ello el cliente también debe de seguir un procedimiento de desconexión adecuado. Esto es, el cliente que desea desconectarse accede al Servicio de Nombrado y se elimina del contexto **`Receivers`** dentro del contexto **`Distributer`** (ver Figura 7.5). Por otro lado, el distribuidor está continuamente actualizando una lista de conexiones de forma local. Cuando acceda al servicio de nombrado y detecte que un cliente de su antigua lista de conexiones ya no está, eliminará correctamente toda la información referente a él.

7.4.3 Negociación Manual

El controlador AV no requiere negociación TCP. Como ya se vio en el apartado 4.2.3.5, existe la posibilidad de no realizar la negociación en los plugins read-net-avi y write-net-avi. Esto se aplica al controlador AV.

No se requiere negociación ya que, como se ha dicho anteriormente, TAO dispone de un Servicio de Propiedades con el que se puede realizar la negociación. Esta parte no está implementada, simplemente se obtienen los valores obtenidos del fichero y se usan directamente como valores negociados. Si estos valores son incorrectos, no conectará.

7.4.4 Compresión

Se debió de usar una compresión que redujera el tamaño de paquete (MJPEG) ya que la transmisión por medio de *AVStreams* permite un tamaño máximo de imagen de 16834 *bytes*. Esto se debe a que podemos crear bloques de mensaje, ver apartado 7.3.2.7, del tamaño que queramos, pero éstos serán fragmentados, si es necesario, en dos paquetes de 8192 *bytes*.

Si la imagen excede 16834, se debería de aplicar un algoritmo de fragmentación y dividirlo en varios fragmentos, tal y como se hizo en el caso de RGB 24 en el controlador UDP, ver apartado 6.3.2. En la implementación realizada, si excede de 16834 no se envía la imagen.

7.4.5 Procesos que intervienen

Debido a la gran cantidad de tareas a realizar, se ha programado la aplicación para que se ejecute en varios procesos. Esto pasa tanto en el cliente como en el servidor.

Por parte del servidor podemos encontrar tres procesos. Uno es el proceso *xawtv*, que genera video y lo envía al distribuidor, otro *naming*, encargado del servicio de nombrado, y otro distribuidor de video, que mantiene la lista de todos los clientes y retransmite el video. Como se observa, los tres procesos son necesarios, ya que si todas las operaciones estuvieran en el mismo proceso se bloquearían mutuamente.

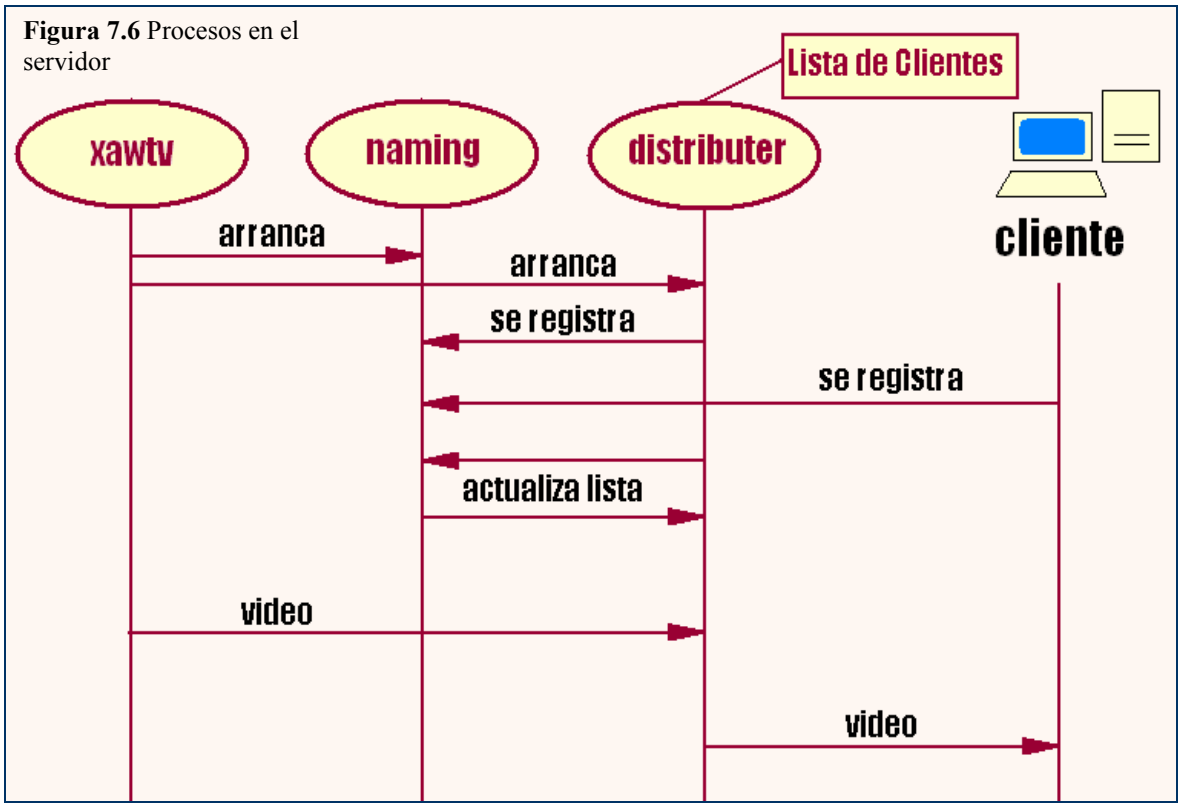


Figura 7.6 Procesos en el servidor

En el extremo del cliente, también son necesarios dos procesos. Si el ORB fuera arrancado dentro del proceso *pia-net*, éste último sólo podría realizar las operaciones relacionadas con el ORB. Por ello se crea un proceso independiente encargado de ejecutar las operaciones del ORB. Como en este caso el proceso *pia-net* ya no tiene control sobre la recepción de video, debe esperar de alguna forma hasta que llegue. Para ésto se crean unos semáforos para controlar el acceso a la variable compartida, que es el video. Hasta que el video no llega, *pia-net* permanece inactivo.

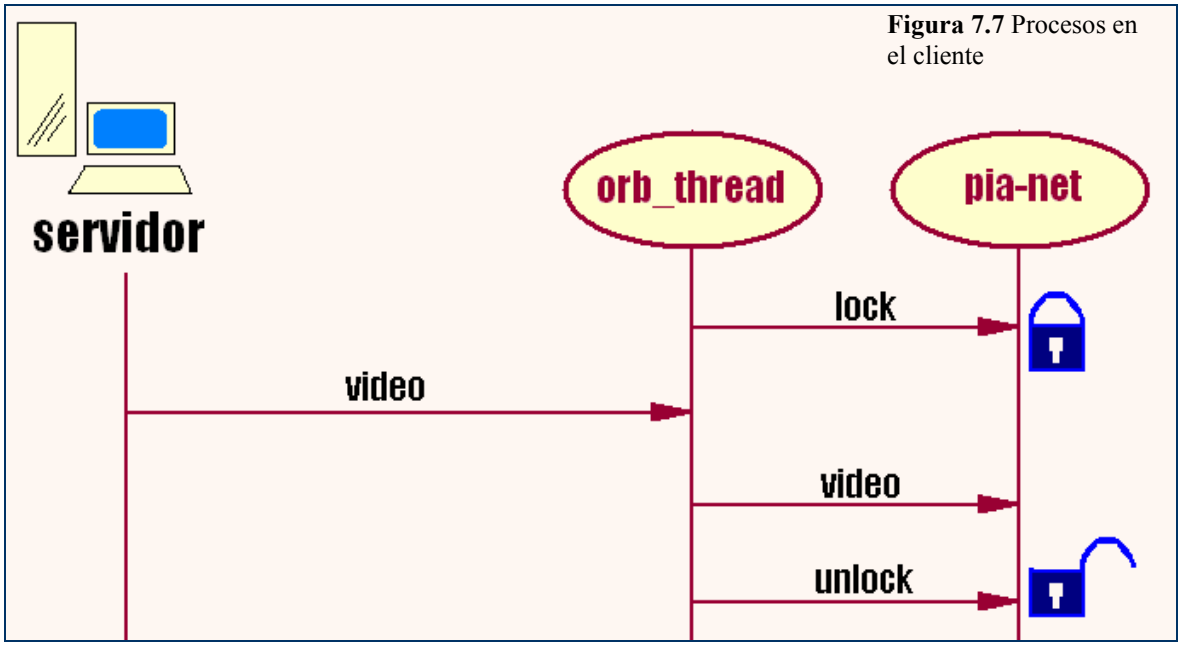


Figura 7.7 Procesos en el cliente

7.5 Implementación del controlador AV

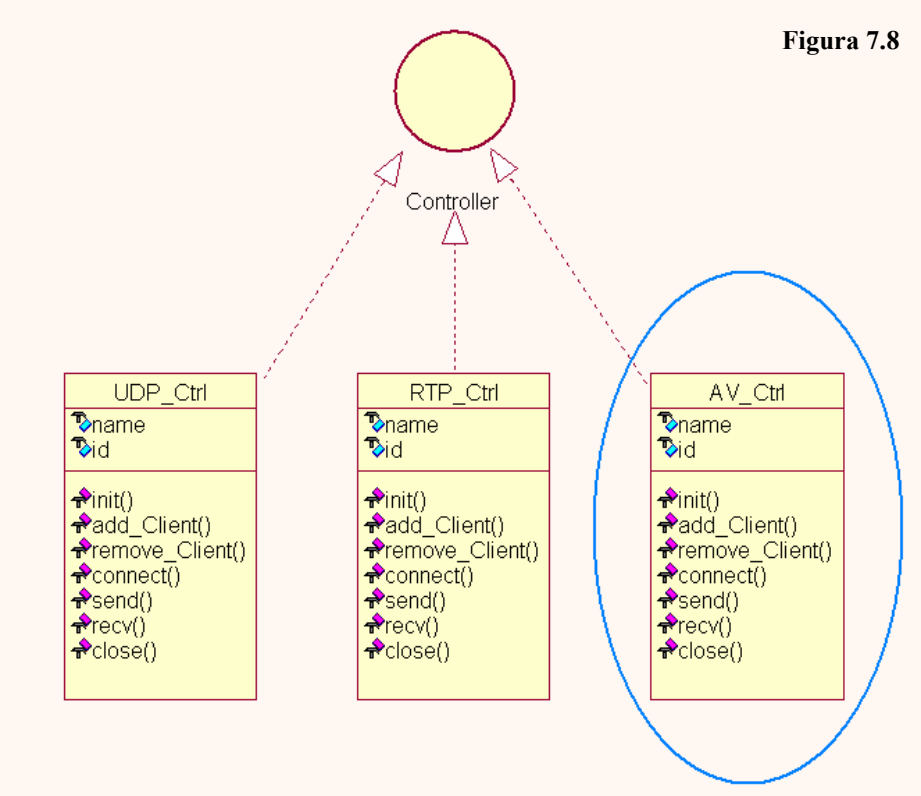
7.5.1 La estructura AV_Ctrl

Implementa a la estructura **`controller`.**

```
struct Controller AV_Ctrl = {

      name:           "AV",
      id:             2,
      init:           init_AV,                //SERVIDOR
      add_Client:     add_AV_Client,           //SERVIDOR
      remove_Client:  remove_AV_Client,       //SERVIDOR
      connect:        connect_AV,              //CLIENTE
      send:           send_AV,                 //SERVIDOR
      recv:           recv_Av,                 //CLIENTE
      close:          close_AV,                //SERVIDOR Y CLIENTE
};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
init_AV()	init()	Carga librerías dinámicas
add_AV_Client()	add_Client()	Añade cliente al servidor
remove_AV_Client()	remove_Client()	Borra cliente del servidor
connect_AV()	connect()	Carga librerías dinámicas. Inicialización
send_AV()	send()	Inicialización y envío de video
recv_AV()	recv()	Recibe video
close_AV()	close()	Cierra conexión

Figura 7.8



7.5.2 AV_Ctrl

7.5.2.1 Implementación de funciones servidor

Un controlador tiene definidas interfaces que usa el servidor e interfaces que usa el cliente. A continuación se describen las funciones relacionadas con el servidor. Se recuerda que solo se muestran las partes más importantes de las funciones. Para conocer más a fondo su implementación se debe de ir al código fuente.

7.5.2.1.1 *init_AV()*

```
int init_AV()
{
      lib_handle_sender = dlopen
      ("/usr/local/lib/xawtv/controllers/corba/libstream_sender.so",
      RTLD_NOW);

      strSendInit_ = dlsym(lib_handle_sender, "strSendInit");

      strSendData_ = dlsym(lib_handle_sender, "strSendData");

      return 0;
}
```

Se encarga de abrir la librería dinámica **`libstream_sender.so`** y cargar las funciones **`strSendInit()`** y **`strSendData()`**. Ver apartados 7.6.2.1 y 7.6.2.2.

7.5.2.1.2 *add_AV_Client()*

```
int add_AV_Client(struct sockaddr_in cli_ad, struct negotiation ng){}
```

Sin implementar.

7.5.2.1.3 *remove_AV_Client()*

```
int remove_AV_Client(struct sockaddr_in cli_ad){}
```

Sin implementar.

7.5.2.1.4 *send_AV()*

```
int send_AV(void *handle, struct ng_video_buf *buf){

      if (first_time == 1)
      {
            int argc = 1;
            h=gethostbyname("localhost”);
            char *IP = inet_ntoa(*((struct in_addr *)h->h_addr));
            strcpy(aux,"-ORBDefaultInitRef iiop://");
            strcat(aux,inet_ntoa(*((struct in_addr *)h->h_addr)));
            strcat(aux,":35000");
            strcat(aux," -s sender");
            vArgs[0]=aux;

            pthread_create(&naming_thread, NULL, naming, (void *)IP);
            pthread_create(&dist_thread, NULL, distributer, (void *)IP);

            (*strSendInit_)(argc, vArgs);
            first_time = 0;
      }
      (*strSendData_)(buf->data, buf->size);

      return 0;
}
```

La función **`send_AV()`** tiene dos cometidos claramente definidos:

Para comenzar, la primera vez que sea llamada esta función, nada más iniciar *xawtv*, lanzará dos *threads*. En uno de ellos se ejecutará el Servicio de Nombrado y en el otro el distribuidor. También llamará a la función **`strSendInit()`**, ver apartado 7.6.2.1, con los argumentos que se pueden observar en el extracto de código.

Las funciones para la creación de los *threads* son muy parecidas. A continuación se puede observar su implementación, es interesante ver los parámetros usados para lanzar cada proceso.

```
void* naming(char *IP){

      naming_pid = fork();
      if (naming_pid == 0)
      {
            char *argv[4];
            argv[0] = "sh";
            argv[1] = "-c";
            argv[2] = (char*) malloc (sizeof(char)*150);

            strcpy(argv[2],"$TAO_ROOT/orbsvcs/Naming_Service/Naming_Serv
            ice -ORBEndPoint iiop://");
            strcat(argv[2],IP);
            strcat(argv[2],":35000");
            argv[3] = "NULL";
            execv("/bin/sh", argv);
      }
}
```

Arrancará el Servicio de Nombrado en un proceso hijo.

```
void* distributer(char *IP)
{
      dist_pid = fork();
      if (dist_pid == 0)
       {
            char *argv[4];
            argv[0] = "sh";
            argv[1] = "-c";
            argv[2] = (char*) malloc (sizeof(char)*150);
            strcpy(argv[2],"/usr/local/lib/xawtv/controllers/corba/distribu
            ter -ORBInitRef NameService=corbaloc:iiop:");
            strcat(argv[2],IP);
            strcat(argv[2],":35000/NameService -s sender -r distributer");
            argv[3] = "NULL";
            execv("/bin/sh", argv);
      }
}
```

Arrancará el distribuidor en un proceso hijo.

El segundo cometido de **`send_AV()`** será realizar llamadas a la función **`strSendData()`**, ver apartado 7.6.2.2. A esta función se le pasará como argumentos el *array* de caracteres donde irá el video y una variable donde se guardará el tamaño de de éste.

7.5.2.1.5 *close_AV()*

```
int close_AV(void *handle,int cs)
```

Al igual que sucedía en el apartado 6.4.3.2.5, la variable **`cs`** indica si es el cliente o el servidor quien llama a la función **`close_AV()`**. Al estar describiendo el funcionamiento desde el punto de vista del servidor, observamos:

```
if (cs == SERVER)
      {
            kill(dist_pid, SIGKILL);
            kill(naming_pid, SIGKILL);
            dlclose(lib_handle_sender);
      }
```

Se “matan” los procesos hijo donde se ejecutaban el Servicio de Nombrado y distribuidor. También se cierra la librería dinámica **`libstream_sender.so`**.

7.5.2.2 Implementación de funciones cliente

7.5.2.2.1 *connect_AV()*

```
int connect_AV(void *handle, struct negotiation ng)
```

Cuando **connect_AV()** es llamada, comienza la inicialización en el cliente. De igual forma que el servidor, el cliente abre una librería dinámica, en este caso **libstream_receiver.so**, y carga funciones útiles para su funcionamiento.

```
lib_handle_receiver = dlopen
("/usr/local/lib/xawtv/controllers/corba/libstream_receiver.so", RTLD_NOW);

strRecvInit_ = dlsym(lib_handle_receiver, "strRecvInit");

strRecvData_ = dlsym(lib_handle_receiver, "strRecvData");

recv_flow_close_ = dlsym(lib_handle_receiver, "flow_close");
```

Otra característica de esta función es que establece la configuración del video mediante un proceso similar al explicado en el apartado 6.4.3.2.1. Los valores resultantes de la negociación, en este caso “manual”, están guardados en **ng**, y ***handle** se guarda en ***h**, de tipo **struct avi_handle.**

Se obtiene el formato de video y compresión de la estructura **ng**, para guardarlos en el manejador ***h.** Este manejador contiene también la dirección IP del servidor al que se deberá conectar el cliente en **h->IP.**

Por último, se llama a la función **StrRecvInit()** con unos parámetros específicos, entre ellos la dirección IP recién mencionada.

```
int argc = 1;
strcpy(aux,"-ORBDefaultInitRef iiop://");
strcat(aux,h->IP);
strcat(aux,":35000");
vArgs[0]=aux;

(*strRecvInit_)(argc, vArgs);
```

7.5.2.2.2 *recv_AV()*

```
struct ng_video_buf* recv_AV(void *handle)
{
	struct avi_handle *h = handle;
	buf = ng_malloc_video_buf(&h->vfmt, 400000);
	h->frames++;
	buf->info.seq = h->frames;
	size = (*strRecvData_)(buf->data);
	frame_rcv = frame_rcv +1;

	return buf;
}
```

Se pasa como argumento el manejador **`*handle.`** Este se usa para poder inicializar y rellenar con datos actuales una estructura de tipo **`*ng_video_buf.`** Esta se pasará como argumento a la función **`strRecvData()`** que la rellenará con una imagen y la información referente a ella. Para concluir devolverá la estructura.

7.5.2.2.3 *close_AV()*

```
int close_AV(void *handle,int cs)
```

Cuando **`cs == CLIENT`**, indica que la función ha sido llamada por el cliente.

```
if (cs == CLIENT)
	{
		(*recv_flow_close_)();
		dlclose(lib_handle_receiver);
	}
```

Se realiza una llamada a la función **`recv_flow_close()`**, ver apartado 7.6.2.2.3, cargada de la librería dinámica **`libstream_receiver.so`**, que también se cerrará.

7.5.3 Diagrama de secuencia de *AV_Ctrl*

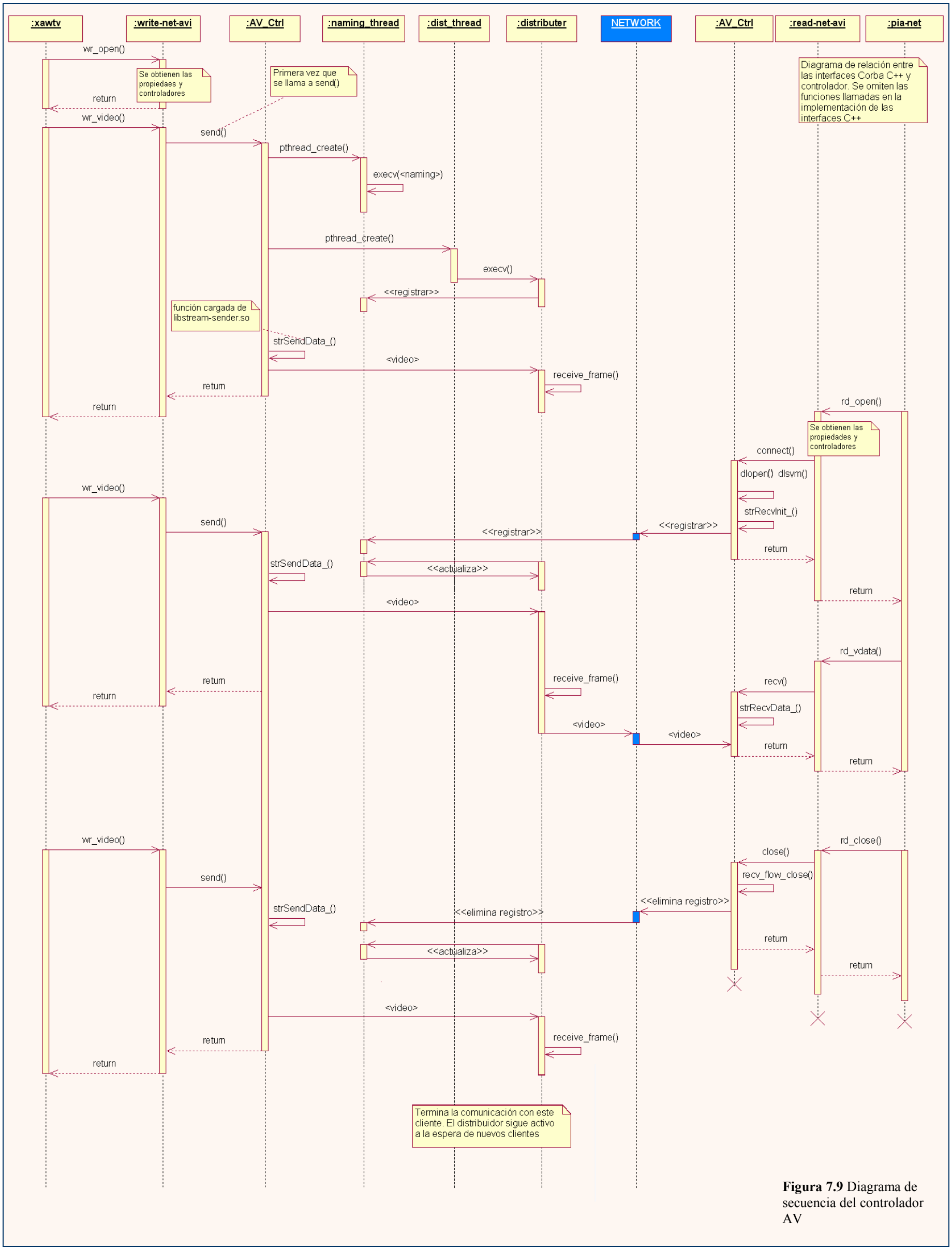


Figura 7.9 Diagrama de secuencia del controlador AV

7.6 Librerías y ejecutables basados en AVStreams

Como se habrá podido observar en los apartados del punto 7.5, el controlador AV tiene un comportamiento muy superficial, esto es, no implementa realmente los algoritmos de transmisión y recepción de video. Simplemente es un mero intermediario entre *xawtv/pia* y un conjunto de librerías. Dentro de estas librerías se encuentra la verdadera implementación.

Existen unos objetos **distributer**, **sender** y **receiver** implementados usando las utilidades ofrecidas por ACE y TAO. Dichos objetos se comunican por medio de unas interfaces con el controlador AV y le ofrecen una verdadera funcionalidad.

7.6.1 Localización fuentes y compilados

Los archivos fuentes implementados en C++, junto con sus makefiles, se encuentran en el siguiente *path.*

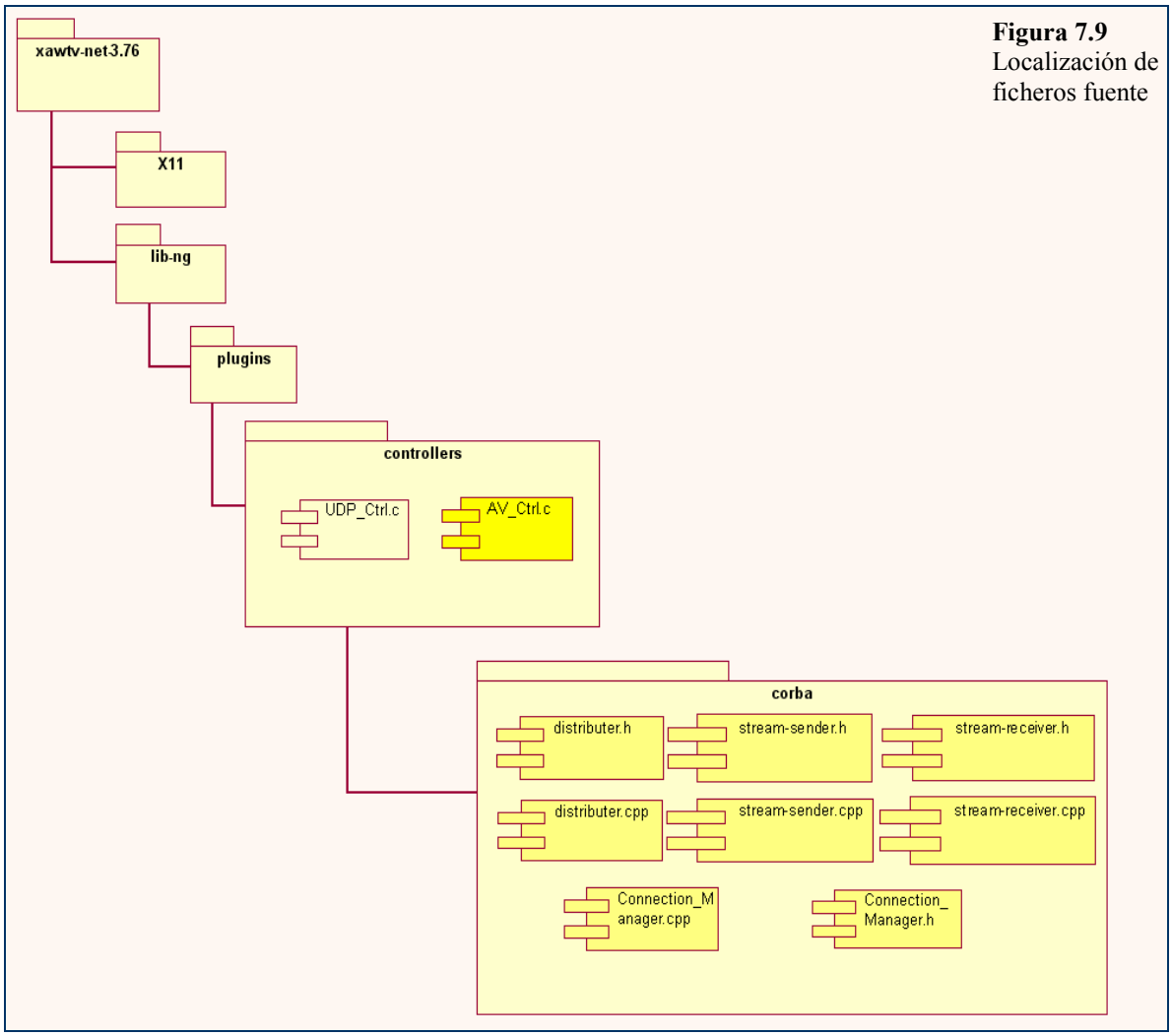


Figura 7.9 Localización de ficheros fuente

Los archivos compilados se podrán encontrar en este otro *path.*

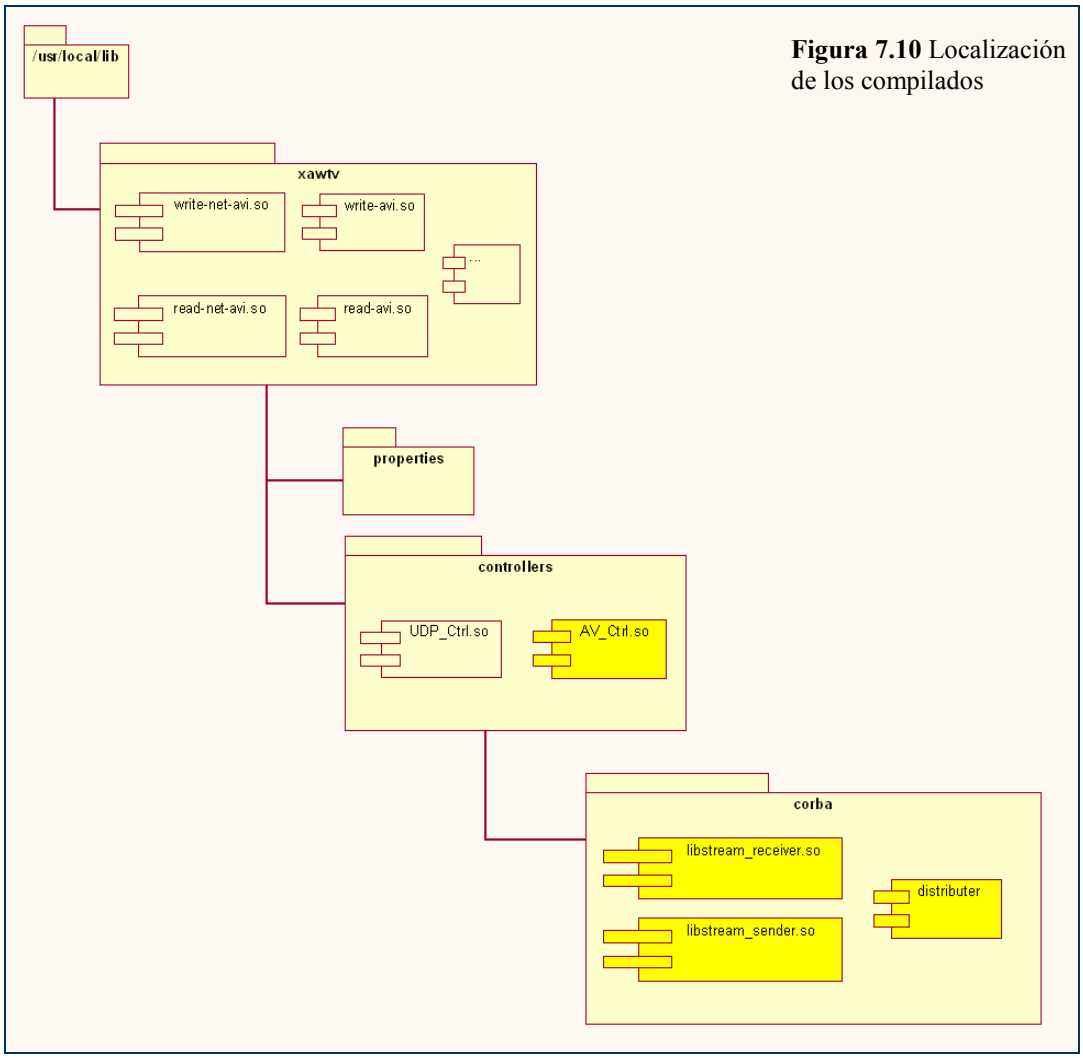


Figura 7.10 Localización de los compilados

7.6.2 Enlace con controlador AV

Debido a la utilización de dos lenguajes de programación distintos, aparece el problema inherente de integrar librerías implementadas en C++ en una aplicación en C.
El caso contrario al que tenemos no ofrece problema alguno, ya que la programación en C++ admite fácilmente código C.

La solución se encuentra en algo intermedio. Se pueden implementar unas funciones en C dentro de las librerías en C++ que nos den acceso a los métodos que se necesiten. El siguiente paso hacia la solución completa, consiste en compilar esas librerías como librerías dinámicas y cargar éstas desde el controlador, tal y como se ha ido viendo en los apartados anteriores.

Aún existe otro problema. Cuando se compila un código C dentro de otro código C++, las funciones cambian de nombre debido al diferente sistema de nombrado de funciones que hay de un lenguaje a otro. Para ello se debe escribir, delante de cada función que necesitemos, la sentencia **`extern "C"`** y el nombre de las funciones no cambiará.

Tras realizar todo lo anterior, ya se tiene una interfaz para comunicar el controlador AV con las implementaciones basadas en AVStreams.

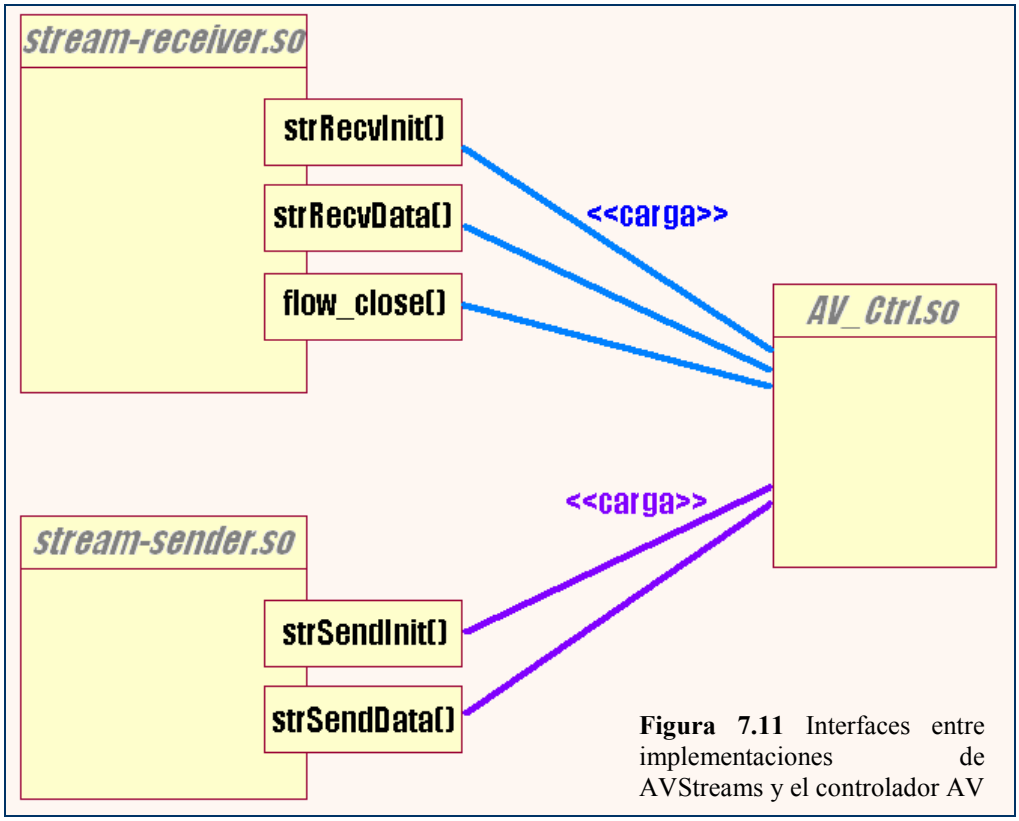


Figura 7.11 Interfaces entre implementaciones de AVStreams y el controlador AV

A continuación, se realiza una descripción de cada una de las funciones mencionadas.

7.6.2.1 Funciones dentro del emisor

Estas funciones se encuentran dentro del fichero stream-sender.cpp y stream-sender.h, ver apartado 7.6.4.4. Son utilizadas por las funciones del controlador AV relacionadas con el emisor.

7.6.2.1.1 *strSendInit()*

```
extern "C" int strSendInit(int argc, char **argv)
{
      if (is_sender_init == 0)
      {
            int result = SENDER::instance() -> initialize(argc, argv);
            is_sender_init = 1;

            return result;
      }

   return 0;
}
```

Es llamada por la función **send_AV(),** del controlador AV, la primera vez que es ejecutada. Los argumentos pasados son necesarios para que el método **initialize(),** de la clase **Sender,** los use para inicializaciones. Debe de ser llamada solamente una vez, ya que los componentes a inicializar solo deben ser creados una vez por ejecución.

7.6.2.1.2 *strSendData()*

```
extern "C" int strSendData(unsigned char * buffer, int size)
{
	int result = SENDER::instance() -> sendData(buffer, size);

	return result;
}
```

Las llamadas se realizan desde la función **send_AV(),** del controlador AV, cada vez que ésta es invocada.

Realiza una llamada al método **sendData()** de la clase **Sender**. Esta es la función que envía el video y para ello se le ha de pasar el *array* de caracteres que lo contiene (**buffer**), y el tamaño de éste (**size**).

7.6.2.2 Funciones dentro del receptor

Estas funciones se encuentran dentro del fichero stream-receiver.cpp y stream-receiver.h, ver apartado 7.6.4.3. Son utilizadas por las funciones del controlador AV relacionadas con el receptor.

7.6.2.2.1 *strRecvInit()*

```
extern "C" int strRecvInit(int arc, char **argv)
{
	if (is_recv_init == 0)
	{
		is_recv_init = 1;
		return RECEIVER::instance()-> initialize(arc,argv);
	}

	return 0;
}
```

Su utilidad es la misma que la de **strSendInit()**. Es llamada por la función **connect_AV(),** del controlador AV. Los argumentos pasados son necesarios para que el método **initialize(),** de la clase **Receiver,** los use para inicializaciones. Debe de ser llamada solamente una vez, ya que los componentes a inicializar solo deben ser creados una vez por ejecución.

7.6.2.2.2 *strRecvData()*

```
extern "C" int strRecvData(unsigned char *buffer)
{
	pthread_mutex_lock( &v_lock);
	while( writeable == 1)
	{
		pthread_cond_wait( &v_cond, &v_lock);
	}

	bcopy(vbuf, buffer, len_aux);
	free(vbuf);
	writeable = 1;
	pthread_cond_broadcast( &v_cond );
	pthread_mutex_unlock( &v_lock );

	return len_aux;
}
```

Como se puede observar, esta función no es meramente un enlace entre el controlador y el objeto receptor. Existen unos semáforos que detienen la ejecución de esta función esperando una condición, ver Figura 7.7. Esto se debe a que **`*buffer`** es modificado fuera de esta función, dentro del método **`receive_frame()`** en la clase **`Receiver_Callback`**, ver apartado 7.3.2.8**,** que se procesa en un *thread* a parte. En la variable **`*buffer`** se guarda el video y la condición indica el momento en el que éste se guarda dentro de la variable. Nunca se debe devolver a read-net-avi, y como consecuencia a la aplicación *pia-net,* una imagen vacía.

7.6.2.2.3 *flow_close()*

```
extern "C" int flow_close(void)
{
	receiver.flow_close();
	orb->destroy();

	return 0;
}
```

Es llamada por la función **`close_AV()`,** del controlador AV, cuando ésta es llamada por el cliente. Se encarga de llamar a una función homónima dentro de la clase **`Receiver`**, útil para una correcta desconexión.

También se encarga de eliminar el ORB.

7.6.3 Diagrama completo de clases

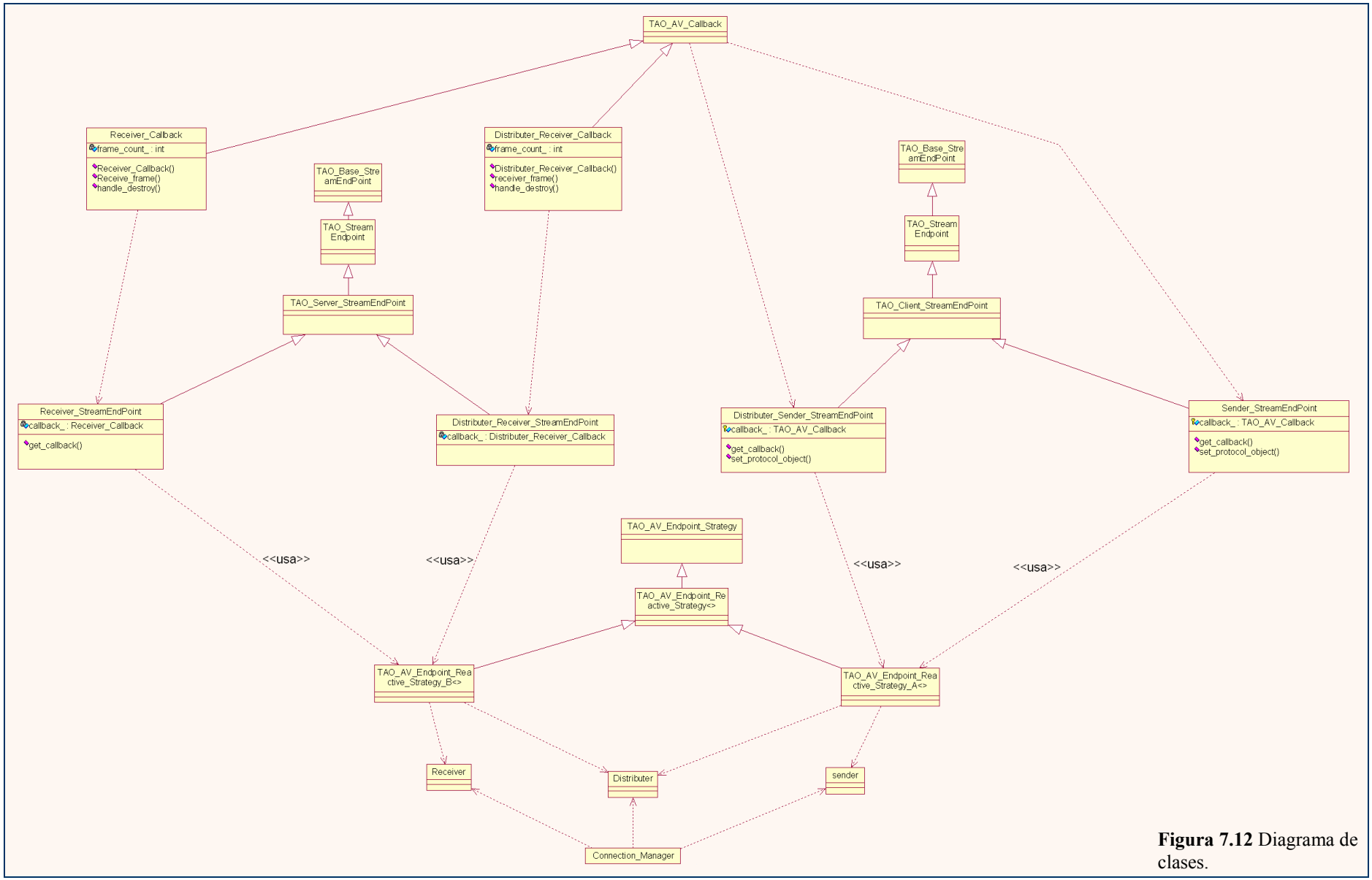


Figura 7.12 Diagrama de clases.

7.6.4 Ficheros y tipo de compilación

Como ya se pudo observar en el apartado 7.6.1, existen varios ficheros implementados en C++. Cada uno de ellos tiene una utilidad distinta. Dependiendo de esa utilidad, también deberán ser compilados de distinta forma (como librería estática, librería dinámica o ejecutable).

A continuación, se realiza un repaso de éstos.

7.6.4.1 Connection_Manager

Los ficheros más útiles diseñados son *Connection_Manager.h* y *Connection_Manager.cpp*.
Dentro de éstos ficheros se define una clase de igual nombre. Dicha clase contiene la mayoría de los métodos a usar por los emisores (**`sender`** y **`distributer`**) y los receptores (**`receiver`**) y variables para guardar estado de objetos relacionados con las comunicaciones, los mapas. Estos métodos son, en su mayoría, utilidades para el manejo del servicio de nombrado.
Cada objeto diseñado, que tenga alguna función de enviar/recibir video, deberá tener un **`Connection_Manager`**.

La compilación de esta clase está asociada a la de los ficheros que contienen las clases que harán uso de ella. Se compila como una librería estática que es incluida dentro librerías dinámicas y ejecutables. No hay un archivo objeto **`Connection_Manager.o`**.

7.6.4.1.1 Mapas

Son plantillas implementadas para el recorrido de listas de objetos. **`Connection_Manager`** contiene tres, junto con sus métodos de acceso a ellos.

```
// Mapa de receivers.
      typedef
      ACE_Hash_Map_Manager<ACE_CString,AVStreams::MMDevice_var,
      ACE_Null_Mutex> Receivers;
// Mapa de protocol objects.
      typedef
       ACE_Hash_Map_Manager<ACE_CString,TAO_AV_Protocol_Object
      *,ACE_Null_Mutex> Protocol_Objects;
// Mapa de streamctrl.
      typedef
      ACE_Hash_Map_Manager<ACE_CString,AVStreams::StreamCtrl_var,
      ACE_Null_Mutex> StreamCtrls;
```

Se observa que contiene un **Hash_Map_Manager** de objetos **MMDevice_var**, otro de objetos **Protocol_Object** y otro de objetos **StreamCtrl_var**. Ver apartados 7.3.2.1, 7.3.2.6.2 y 7.3.2.4 respectivamente.

Los objetos que dan acceso a los mapas se denominan *map accessors* y son los tres métodos siguientes:

```
Receivers &receivers (void);
Protocol_Objects &protocol_objects (void);
StreamCtrls &streamctrls (void);
```

Devuelven punteros a los **Hash_Map_Manager.**

7.6.4.1.2 Métodos

init()

```
int
Connection_Manager::init (CORBA::ORB_ptr orb)
{
      if (this->naming_client_.init (orb) != 0)
      {
            return -1;
      }
      return 0;
}
```

Es el encargado de inicializar el cliente del servicio de nombrado. Este es **naming_client** que es de tipo **TAO_Naming_Client**. Este tipo de objeto apunta al contexto raíz del *Naming Service*, como se vio en el apartado 7.3.1.

bind_to_receivers()

```
void Connection_Manager::bind_to_receivers(
                    const ACE_Cstring &sender_name,
                    AVStreams::MMDevice_ptr sender ACE_ENV_ARG_DECL)
```

La finalidad de este método es registrar el objeto emisor con el contexto emisor.
Intenta crear el contexto del emisor bajo el contexto raíz del Naming Service, el contexto del emisor puede ser **distributer** o **sender**, depende de la variable **sender_name**. Esta operación puede provocar una Excepción o no:

- No salta excepción: No estaba creado el contexto emisor, lo crea. Dentro de este contexto emisor, crea contexto **Receivers**, usando el método **bind_new_context()** (por el hecho de crear el contexto, ya podemos acceder a él, pues se guardo la referencia). Se registra el objeto emisor dentro del contexto emisor.

- Salta excepción: otro cliente ya ha realizado el paso anterior. Como el contexto está ya creado, tenemos que acceder a él. Se accede al contexto del emisor, después accede al contexto **Receivers** y se guarda la referencia. Los accesos se realizan usando los métodos **resolve()** y **_narrow()**. Tras esto, se llama al método **find_receivers()**.

Por último, registra el objeto emisor con el contexto emisor usando el método **rebind()**.

find_receivers()

```
void
Connection_Manager::find_receivers (ACE_ENV_SINGLE_ARG_DECL)
```

Este método obtiene la lista de receptores a partir del servidor de nombres usando el método **list()** aplicado a un contexto, éste es un objeto de la clase **CosNaming::NamingContext_var**. El listado se guarda en un objeto de tipo **CosNaming::BindingList_var**, que contiene métodos para el recorrido de éste.
Tras esto, se realizan sucesivas llamadas a **add_to_receivers()** para cada uno de los receptores listados.

add_to_receivers()

```
void
Connection_Manager::add_to_receivers (
      CosNaming::BindingList &binding_list ACE_ENV_ARG_DECL)
```

La llamada a este método procede de **find_receivers()**. Se utiliza la lista obtenida en dicho método (**binding_list**) para ir añadiendo cada receptor al objeto **receivers_**, que es un mapa de tipo **Receivers**. Se añaden por medio de la función **bind()**, la que asocia también a cada objeto, de tipo **MMDevice_var**, un nombre de flujo (**flowname**) que lo identifica.

check_receivers()

```
int Connection_Manager::check_receivers(void)
```

Es usado por el distribuidor. Realiza el control de los clientes a eliminar de los mapas, guardados dentro del objeto **Connection_Manager** del distribuidor. Esos clientes a eliminar son los que ya no están conectados. El procedimiento es el siguiente:

- Accede al contexto del emisor, que es el distribuidor.
- Accede al contexto **Receivers** dentro del contexto **distributer**.
- Lista todos los clientes y los guarda. Comprueba si es la primera vez que se accede:
  - Si es la primera vez: inicializa la lista anterior (en blanco).
  - Si no es la primera vez: no hace nada

- Recorre la lista antigua (vacía si es la primera vez)
- Recorre la lista nueva y la compara con la antigua. Se comparan los nombres que contienen las listas (son los nombres de los objetos registrados)
  - Si son iguales las dos listas: no hace nada
  - Si hay algún elemento que se encontraba en la lista antigua, y no está en la nueva, es que se ha borrado del servidor de nombres. Entonces se hace una llamada al método **`destroy()`** con argumento el nombre del flujo (**`flowname`**). Esta acción elimina todo lo relacionado con ese flujo, que ya no existe.

`connect_to_receivers()`

```
void
Connection_Manager::connect_to_receivers (ACE_ENV_SINGLE_ARG_DECL)
```

El cometido de esta función es conectar un emisor con todos sus receptores conocidos.

Para ello recorre todo el mapa de tipo **`Receivers`** por medio de un iterador preparado para ello, ésta es una de la utilidad de usar mapas. Estos objetos de la clase **`MMDevice_var`** se añadieron anteriormente mediante el método **`add_to_receivers()`**. Para cada receptor:

- Se crea la descripción del flujo. Es un objeto de la clase **`AVStreams::flowSpec`**.
- Crea un objeto de la clase **`TAO_StreamCtrl`**, ver apartado 7.3.2.4.
- Registra el objeto de tipo **`StreamCtrl`** en el mapa **`streamctrls_`** con el nombre de **`flowname`** por medio del método **`bind()`.**
- Conecta los *MMDevices* de emisor y receptor, usando el **`flowSpec`** creado, por medio del método **`bind_devs()`**.

`bind_to_sender()`

```
void
Connection_Manager::bind_to_sender (const ACE_CString &sender_name,
                                    const ACE_CString &receiver_name,
                                    AVStreams::MMDevice_ptr receiver
                                    ACE ENV ARG DECL)
```

Se usa para registrar los objetos receptores con el contexto **`Receivers`** que corresponda. Se realizan operaciones análogas a las realizadas en **`bind_to_receivers()`**, aunque en este caso hace falta conocer también el nombre con el que se registrará el receptor, éste es **`receiver_name`**.

`connect_to_sender()`

```
void
Connection_Manager::connect_to_sender (ACE_ENV_SINGLE_ARG_DECL)
```

Es análoga a la función **connect_to_receivers()**. Conecta un cliente con su emisor siguiendo el siguiente procedimiento:

- Se crea la descripción del flujo. Es un objeto de la clase **AVStreams::flowSpec**.
- Crea un objeto de la clase **TAO_StreamCtrl**, en este caso no lo registra en ningún mapa ya que sólo hay una conexión. ver apartado 7.3.2.4 y figura 7.2 .
- Conecta los *MMDevices* de emisor y receptor, usando el **flowSpec** creado, por medio del método **bind_devs()**.
- "Arranca" el flujo por medio del método **start()** de **TAO_StreamCtrl**

add_streamctrl()

```
void
Connection_Manager::add_streamctrl (const ACE_CString &flowname,
                                    TAO_StreamEndPoint *endpoint
                                    ACE_ENV_ARG_DECL)
```

Añade un **TAO_StreamCtrl** al mapa **StreamCtrls** . Útil cuando se desea realizar esta operación fuera de la clase **Connection_Manager**. El objeto se duplica antes de añadirlo, ya que lo normal es que sea usado por cada flujo creado. No se puede usar un solo *streamctrl* para controlar varios flujos, ver figura 7.2.

destroy()

```
void
Connection_Manager::destroy (const ACE_CString &flowname
                             ACE_ENV_ARG_DECL_NOT_USED)
{
      this->protocol_objects_.unbind (flowname);
      this->receivers_.unbind (flowname);
      this->streamctrls_.unbind (flowname );
}
```

Elimina un flujo de todos los mapas por medio de su **flowname**. Importante para mantener una información fiable dentro de **Connection_Manager**.

unbind_receiver()

```
void
Connection_Manager::unbind_receiver (const ACE_CString &sender_name,
                                     const ACE_CString &receiver_name)
```

Elimina el registro de un objeto receptor del Servicio de Nombrado a partir de su nombre. Los pasos que sigue este método son:

- Acceder al contexto del emisor dado por el parámetro **`sender_name`**, puede ser **`sender`** o **`distributer`**.
- Accede al contexto **`Receivers`**.
- Llama a la función **`unbind()`** de **`CosNaming::NamingContext_var`** usando el nombre del receptor dado por **`receiver_name`**. Esto elimina el objeto del Servicio de Nombrado.

7.6.4.2 Distributer

Los archivos distributer.cpp y distributer.h contienen las clases necesarias para la funcionalidad del distribuidor, éstas llevan métodos útiles para recepción de datos y el reenvío de éstos. La relación entre éstas se puede observar en el diagrama de clases de la figura 7.12.

El archivo generado en la compilación de estos dos ficheros es un binario de nombre **`distributer`**. Este binario se podrá ejecutar usando los siguientes parámetros, en el orden indicado:

```
-ORBInitRef NameService =
corbaloc:iiop:[dirección_IP_servicio_nombrado]:[puerto]/NameService

-s [nombre emisor]

-r [nombre receptor]
```

El nombre del emisor deberá ser *sender* y el nombre del receptor *distributer*.

7.6.4.2.1 Clases y Métodos

No se realizará una explicación detalla de los métodos, ya que la mayoría de ellos siguen una secuencia predefinida por AVStreams. Esto se puede observar en el código de la implementación de distributer.cpp.

7.6.4.2.1.1 Clase Distributer_Sender_StreamEndPoint

Para la compresión de esta clase ver Figura 7.12 y apartado 7.3.2.5. Usada cuando el distribuidor actúa como emisor.

`get_callback()`

```
int
Distributer_Sender_StreamEndPoint::get_callback (const char *,
                                                 TAO_AV_Callback *&callback)
```

Crea y devuelve la aplicación de *callback* a *AVStreams* para futuras llamada cuando el distribuidor actúa de emisor.

set_protocol_object()

```
int
Distributer_Sender_StreamEndPoint::set_protocol_object (
                                    const char *flowname,
                                    TAO_AV_Protocol_Object *object)
```

Obtiene el mapa de tipo **Protocol_Objects** de su **Connection_Manager** y añade un objeto a él identificado por el nombre de flujo **flowname**.

Tras ésto, usa como argumento de **add_streamctrl()** el objeto que contiene el método. A partir de un objeto **Distributer_Sender_StreamEndPoint** se puede crear un objeto **StreamCtrl**, ya que este hereda de **StreamEndPoint**. Se duplicará dicho **StreamCtrl** y se añadirá al mapa de **StreamCtrls**.

7.6.4.2.1.2 Clase Distributer_Receiver_StreamEndPoint

Para la compresión de esta clase ver Figura 7.12 y apartado 7.3.2.5. Usada cuando el distribuidor actúa como receptor.

get_callback()

```
int
Distributer_Receiver_StreamEndPoint::get_callback (
                                        const char *,
                                        TAO_AV_Callback *&callback)
```

Crea y devuelve la aplicación de *callback* a *AVStreams* para futuras llamadas cuando el distribuidor actúa de receptor.

7.6.4.2.1.3 Clase Distributer_Receiver_Callback

Clase usada cuando el distribuidor actúa como receptor. Contiene métodos de *Callback* implementados para recibir *frames* y reenviarlos.

Distributer_Receiver_Callback()

```
Distributer_Receiver_Callback::Distributer_Receiver_Callback (void)
```

Constructor. Inicializa la variable interna **frame_count**, contador de *frames* recibidos.

receive_frame()

```
int
Distributer_Receiver_Callback::receive_frame (ACE_Message_Block *frame,
                                              TAO_AV_frame_info *,
                                              const ACE_Addr &)
```

Método llamado automáticamente cada vez que un *frame* es recibido por el distribuidor.

El video va encapsulado dentro de un objeto de tipo **`ACE_Message_Block`**. Este es reenviado a todos los receptores registrados haciendo uso del mapa **`Protocol_Objects`** contenido en su **`Connection_Manager`** como se muestra a continuación.

```
for
(Connection_Manager::Protocol_Objects::iterator
iterator = protocol_objects.begin ();
iterator != protocol_objects.end (); ++iterator)
{
      (*iterator).int_id_->send_frame (frame);
}
```

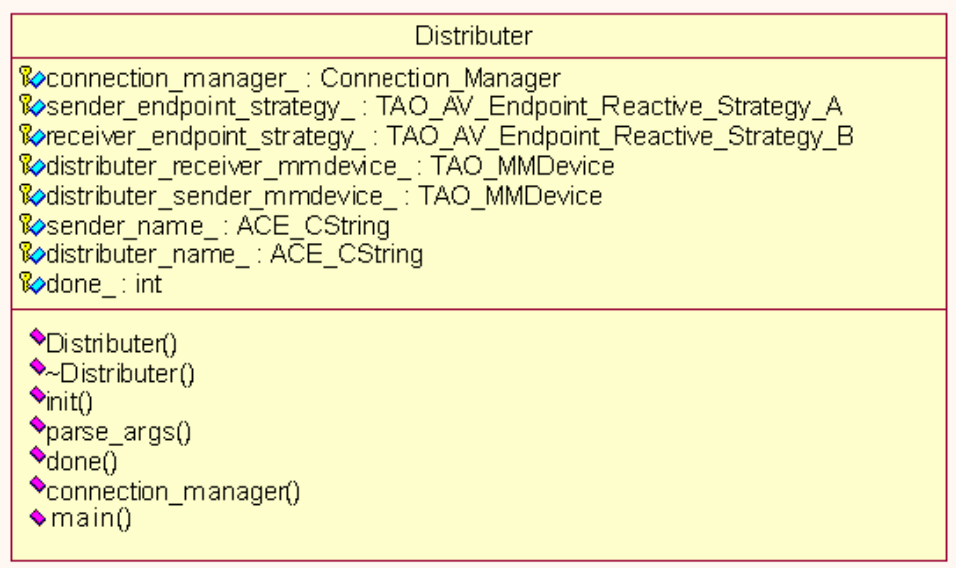
Se recuerda que dentro del mapa **`Protocol_Objects`** hay un objeto de tipo **`Protocol_Object`** por cada receptor.

`handle_destroy()`

```
int
Distributer_Receiver_Callback::handle_destroy (void)
```

Sirve para indicar al distribuidor cuando debe ser finalizado. Para ello se coloca el *flag* **`done = 1`** .

7.6.4.2.1.4 Clase Distributer



La clase **`Distributer`** es la encargada de recibir el video y reenviarlo a todos los clientes. Hace uso de todas las clases explicadas anteriormente.

Distributer() y ~Distributer()

```
Distributer::Distributer (void)
Distributer::~Distributer (void)
```

Constructor y Destructor.

El método constructor inicializa las variables **sender_name** y **receiver_name,** a los valores por defecto **sender** y **distributer,** y **done = 0,** para indicar que el distribuidor está activo.

El destructor está sin implementar.

connection_manager()

```
Connection_Manager &
Distributer::connection_manager (void)
```

Método de acceso al **Connection_Manager** del distribuidor.

paser_args()

```
int
Distributer::parse_args (int argc, char **argv)
```

Modifica variables internas de **Distributer** a partir de los argumentos de entrada del programa.

init()

```
int
Distributer::init (int argc, char ** argv ACE_ENV_ARG_DECL)
```

Inicializa el **Connection_Manager** del distribuidor, las estrategias y otros componentes. Una vez realizadas las inicializaciones, se realizan llamadas a métodos de **Connection_Manager** útiles para inicializar la comunicación. Estos son **bind_to_receivers(), connect_to_receivers(), bind_to_sender()** y **connect_to_sender()**, ver apartado 7.6.4.1.2.

main()

```
int main (int argc, char **argv)
```

Realiza inicializaciones de componentes típicos de comunicaciones CORBA y de *AVStreams*. Entre éstas se encuentra la inicialización del ORB y la obtención del POA. Antes de finalizar el método llama a **init()** explicado en el apartado anterior y entra en un bucle, éste solo se rompe cuando el *flag* **done == 1**.

En el interior de dicho bucle llama continuamente a dos métodos. El primero es **`perform_work()`** de la clase **`CORBA::ORB_var`**, le indica al ORB que procese el trabajo pendiente. El segundo es **`check_receivers()`** de **`Connection_Manager`**, ver apartado 7.6.4.1.2.

7.6.4.2.2 Estrategias

Como ya se vio en el apartado 7.3.2.1, se pueden aplicar estrategias a la creación de *EndPoints*. También éstas podían estar diferenciadas según fueran emisores o receptores, ver Figura 7.12. En el caso del distribuidor existen dos *EndPoints*. Uno para cuando actúa de receptor, con una estrategia de receptor, y otro para cuando lo hace de emisor, con una estrategia de emisor. Estas son las siguientes:

La estrategia de emisor.

```
typedef TAO_AV_Endpoint_Reactive_Strategy_A
          <Distributer_Sender_StreamEndPoint,
           TAO_VDev,
           AV_Null_MediaCtrl>
        SENDER_ENDPOINT_STRATEGY;
```

La estrategia de receptor.

```
typedef TAO_AV_Endpoint_Reactive_Strategy_B

<Distributer_Receiver_StreamEndPoint,
           TAO_VDev,
           AV_Null_MediaCtrl>
        RECEIVER_ENDPOINT_STRATEGY;
```

Las estrategias se aplican a los dos *EndPoints* definidos anteriormente y son inicializadas en el método **`init().`** Para mejor compresión, ver Figura 7.12

7.6.4.3 Stream-receiver

Los archivos **`stream-reveiver.cpp`** y **`stream-receiver.h`** contienen las clases y métodos necesarios para el funcionamiento del cliente.

La relación entre clases se puede observar en la figura 7.12.

El archivo generado en la compilación es una librería dinámica. Puede ser cargada desde el controlador AV y usar las funciones **`extern "C"`** para comunicarse.

7.6.4.3.1 Clases y Métodos

No se realizará una explicación detalla de los métodos, ya que la mayoría de ellos siguen una secuencia predefinida por AVStreams. Esto se puede observar en el código de la implementación de stream-receiver.cpp.

7.6.4.3.1.1 Clase Receiver_StreamEndPoint

Para la compresión de esta clase ver Figura 7.12 y apartado 7.3.2.5. Es un *EndPoint*, usado por la clase **`Receiver`**, para la recepción de datos.

`get_callback()`

```
int
Receiver_StreamEndPoint::get_callback (const char *,
                                       TAO_AV_Callback *&callback)
```

Crea y devuelve la aplicación de *Callback* a *AVStreams* para futuras llamadas. El receptor solo usará este *EndPoint*, ya que solamente actúa de receptor.

7.6.4.3.1.2 Clase Receiver_Callback

`Receiver_Callback()`

```
Receiver_Callback::Receiver_Callback (void)
```

Constructor. Inicializa la variable interna **`frame_count`**, contador de *frames* recibidos.

`receive_frame()`

```
int
Receiver_Callback::receive_frame (ACE_Message_Block *frame,
                                  TAO_AV_frame_info *frame_info,
                                  const ACE_Addr &address)
```

Método llamado automáticamente cada vez que un *frame* es recibido por el distribuidor.

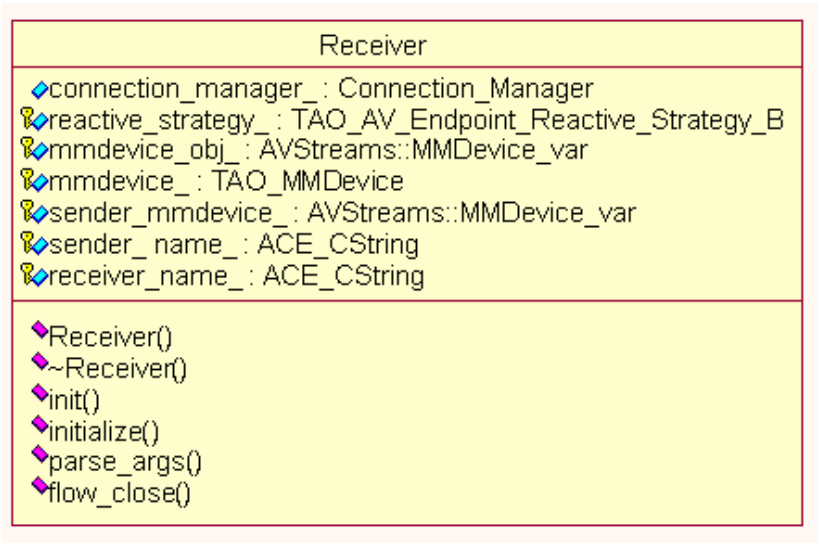
Obtiene la información del *frame* y lo copia a una variable. Esa variable es usada por la función **`strRecvData()`**, mencionada en el apartado 7.6.2.2.2, que está bloqueada por un semáforo hasta que el video sea recibido, ver Figura 7.7. Tras haber recibido el video, **`strRecvData()`** puede continuar y devolver el video al controlador AV, que a su vez lo devolverá a **`read-net-avi`**.

handle_destroy()

```
int
Receiver_Callback::handle_destroy (void)
```

Utilizado para indicar el final de la ejecución del cliente. Pone el flag **done = 1**.

7.6.4.3.1.3 Clase Receiver



Receiver es la clase encargada del paso del video, recibido desde el distribuidor, al controlador AV. Este a su vez lo entregará a *pia-net* para su reproducción.

Receiver() y ~Receiver()

```
Receiver::Receiver (void)
Receiver::~Receiver (void)
```

Constructor y destructor.

La función principal del constructor es inicializar las variables **sender_name** y **receiver_name**. La variable **sender_name** será **distributer** y **receiver_name** se rellenará haciendo uso de un método que creará un nombre aleatorio para ella. Dicho método es **client_name( )**, que está implementado para generar un nombre de receptor del tipo **recv-[número aleatorio]**.

init()

```
int
Receiver::init (int, char ** ACE_ENV_ARG_DECL)
```

Inicializa el **Connection_Manager** del receptor, las estrategias y otros componentes. Una vez realizadas las inicializaciones, se realizan llamadas a métodos de **Connection_Manager** útiles para inicializar la comunicación. Estos son **bind_to_sender()** y **connect_to_sender()**, ver apartado 7.6.4.1.2.

parse_args()

```
int
Receiver::parse_args (int argc, char **argv)
```

No es de utilidad y está sin implementar.

initialize()

```
int
Receiver::initialize (int argc, char **argv)
```

Método llamado desde **StrRecvInit().** Realiza operaciones análogas a las realizas en el método **main()** de **Distributer**. Éstas son, inicializaciones de componentes típicos de comunicaciones CORBA y de *AVStreams*. Entre éstas se encuentra la inicialización del ORB y la obtención del POA.

Antes de finalizar, el método llama a **init()** anteriormente explicado.

A diferencia que en **distributer**, el ORB no es arrancado directamente en éste método. Se debe de arrancar en un proceso distinto, ver apartado 7.4.5. El método que ejecutará el *thread* será **orb_start()**, que a su vez llamará a **run()** de la clase **CORBA::ORB_var**.

flow_close()

```
int Receiver::flow_close(void)
{
      ACE_CString flowname = send_name + "_" + this->receiver_name_ ;
      this->connection_manager_.destroy
                                    ( flowname ACE_ENV_ARG_DECL_NOT_USED );
      this->connection_manager_.unbind_receiver
                                    ( send_name, this->receiver_name_ );
      return 0;
}
```

Es llamado desde la función **extern "C" flow_close()**. Desconecta correctamente el cliente. Destruye todos los objetos del **Connection_Manager** relacionados con **flowname** mediante el método **destroy()** y elimina el registro del cliente en el Servicio de Nombrado, mediante el método **unbind_receiver().**

7.6.4.3.2 Estrategias

De igual forma a la vista en el apartado 7.6.4.2.2, estrategias del distribuidor, se pueden crear estrategias para los *EndPoint* de los receptores. Serán estrategias de tipo receptor y solo hay una por receptor.

```
TAO_AV_Endpoint_Reactive_Strategy_B

<Receiver_StreamEndPoint, TAO_VDev,
AV_Null_MediaCtrl>
 reactive_strategy_;
```

7.6.4.4 Stream-sender

Los archivos **`stream-sender.cpp`** y **`stream-sender.h`** contienen las clases y métodos necesarios para el funcionamiento del servidor.

La relación entre clases se puede observar en la figura 7.12.

El archivo generado en la compilación es una librería dinámica. Puede ser cargada desde el controlador AV y usar las funciones **`extern "C"`** para comunicarse.

7.6.4.2.1 Clases y Métodos

No se realizará una explicación detalla de los métodos, ya que la mayoría de ellos siguen una secuencia predefinida por AVStreams. Esto se puede observar en el código de la implementación de stream-sender.cpp.

7.6.4.2.1.1 Clase Sender_StreamEndPoint

Para la compresión de esta clase ver Figura 7.12 y apartado 7.3.2.5. Es un *EndPoint*, usado por la clase **`server`**, para la transmisión de datos.

`get_callback()`

```
int
Sender_StreamEndPoint::get_callback (const char *,
                                     TAO_AV_Callback *&callback)
```

Crea y devuelve la aplicación de *callback* a *AVStreams* para futuras llamadas.

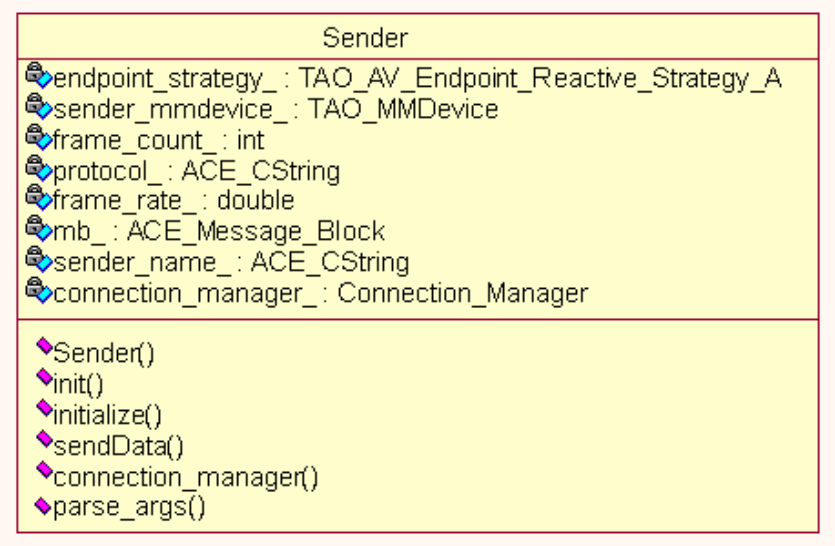
`set_protocol_object()`

```
int
Distributer_Sender_StreamEndPoint::set_protocol_object (
                                    const char *flowname,
                                   TAO_AV_Protocol_Object *object)
```

Obtiene el mapa de tipo **Protocol_Objects** de su **Connection_Manager** y añade un objeto a él identificado por el nombre de flujo **flowname**.

Tras esto, usa como argumento de **add_streamctrl()** el objeto que contiene el método. A partir de un objeto **Sender_StreamEndPoint** se puede crear un objeto **StreamCtrl**, ya que este hereda de **StreamEndPoint**. Dicho **StreamCtrl** se añadirá al mapa de tipo **StreamCtrls,** tras haber sido duplicado.

7.6.4.2.1.2 Clase Sender



Sender() y ~Sender()

```
Sender::Sender (void)
Sender::~Sender (void)
```

Constructor y Destructor.

El constructor inicializa variables a valores por defecto, entre esta variables están **frame_count**, contador de *frames* enviados, **frame_rate**, parametrización de la tasa de envío, **mb_** , de tipo **ACE_Message_Block** (ver 7.3.2.7), y **sender_name**, nombre del emisor inicializado a **sender**.

Destructor sin implementar.

parse_args()

```
int
Sender::parse_args (int argc,char **argv)
```

No es de utilidad y está sin implementar.

init()

```
int
Sender::init (int, char ** ACE_ENV_ARG_DECL)
```

Inicializa el **Connection_Manager** del emisor, las estrategias y otros componentes. Una vez realizadas las inicializaciones, se realizan llamadas a métodos de **Connection_Manager** útiles para inicializar la comunicación. Estos son **bind_to_receivers()** y **connect_to_receivers()**, ver apartado 7.6.4.1.2.

sendData()

```
int
Sender::sendData (unsigned char *base, int len ACE_ENV_SINGLE_ARG_DECL)
```

Método llamado por la función **strSendData()** desde el Controlador AV. Es la encargada de enviar el video. Recibe como argumentos **base**, que es la imagen, y **len**, la longitud de la imagen.

Si se desea enviar una imagen, **sendData()** controla también la tasa de envío haciendo uso de la variable **frame_rate** y *timers*. Cuando el *timer* expira, se envía la imagen previamente encapsulada en un objeto de tipo **ACE_Message_Block**. En el envío se obtiene el mapa que contiene los **Protocol_Object** y se envía el video de forma similar a la mostrada anteriormente en el distribuidor, haciendo uso también del método **send_frame()**.

initialize()

```
int
Sender::initialize (int argc, char **argv ACE_ENV_SINGLE_ARG_DECL )
```

Llamado desde **strSendInit()**. Realiza operaciones análogas a las realizas en el método **main()** de **Distributer** e **initialize()** de **Receiver**. Éstas son, inicializaciones de componentes típicos de comunicaciones CORBA y de *AVStreams*. Entre éstas se encuentra la inicialización del ORB y la obtención del POA.

Antes de finalizar, el método llama a **init()** anteriormente explicado.

7.6.4.2.2 Estrategias

Es un tipo de estrategia reactiva para *EndPoints* que se encargaran de transmitir video.

```
TAO_AV_Endpoint_Reactive_Strategy_A
<Sender_StreamEndPoint,TAO_VDev,
AV_Null_MediaCtrl>
        SENDER_ENDPOINT_STRATEGY;
```

7.6.5 Diagramas de secuencia C++

En este apartado se expondrán diagramas de secuencia para así conseguir una mejor comprensión de las funciones y métodos explicados anteriormente.

7.6.5.1 Inicializaciones

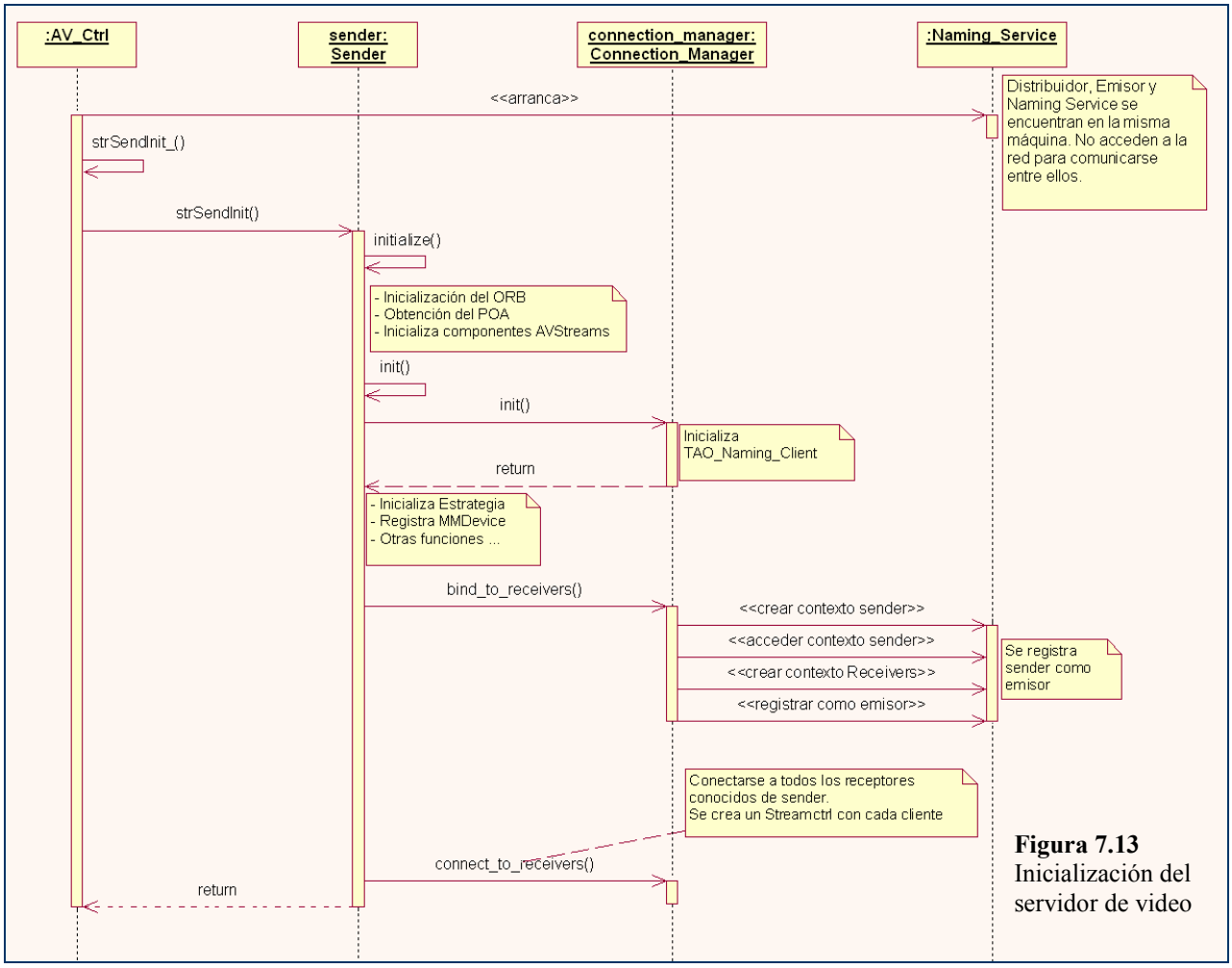


Figura 7.13 Inicialización del servidor de video

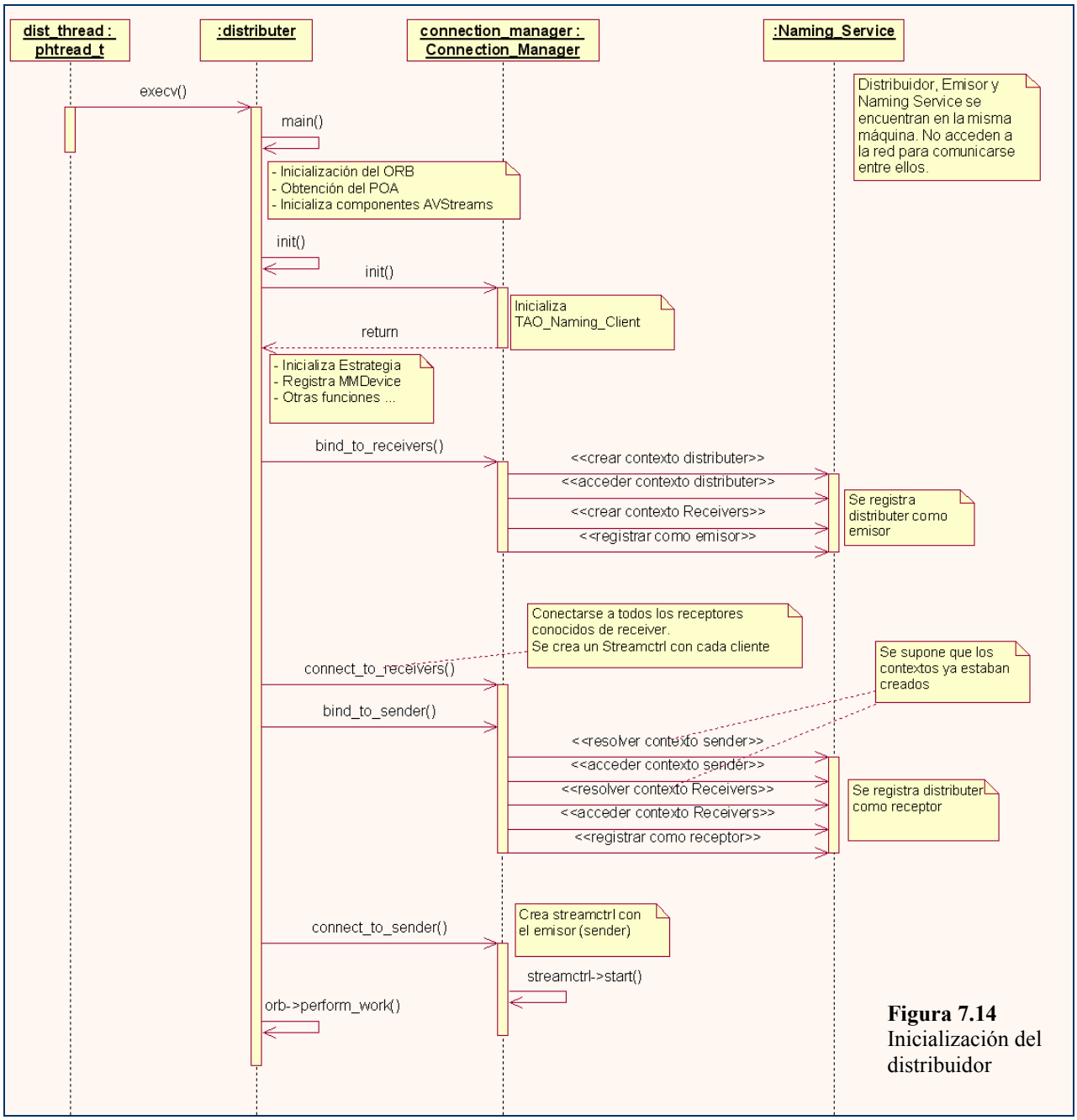
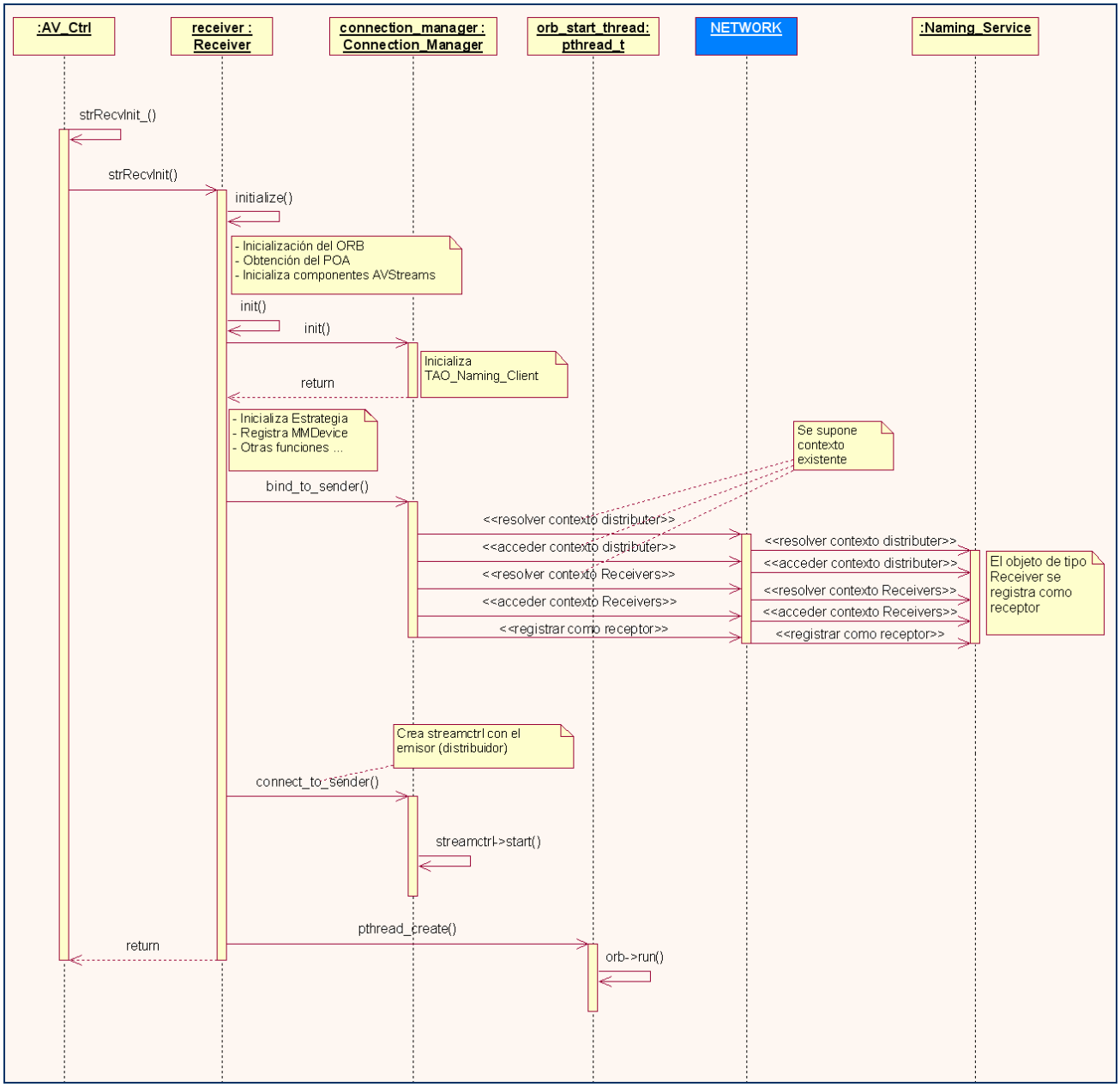


Figura 7.14 Inicialización del distribuidor


:AV_Ctrl
receiver : Receiver
connection_manager : Connection_Manager
orb_start_thread: pthread_t
NETWORK
:Naming_Service
strRecvInit_()
strRecvInit()
initialize()
- Inicialización del ORB
- Obtención del POA
- Inicializa componentes AVStreams
init()
init()
Inicializa TAO_Naming_Client
return
- Inicializa Estrategia
- Registra MMDevice
- Otras funciones ...
Se supone contexto existente
bind_to_sender()
<<resolver contexto distributer>>
<<acceder contexto distributer>>
<<resolver contexto Receivers>>
<<acceder contexto Receivers>>
<<registrar como receptor>>
<<resolver contexto distributer>>
<<acceder contexto distributer>>
<<resolver contexto Receivers>>
<<acceder contexto Receivers>>
<<registrar como receptor>>
El objeto de tipo Receiver se registra como receptor
Crea streamctrl con el emisor (distribuidor)
connect_to_sender()
streamctrl->start()
pthread_create()
return
orb->run()

7.6.5.2 Transmisión/Recepción

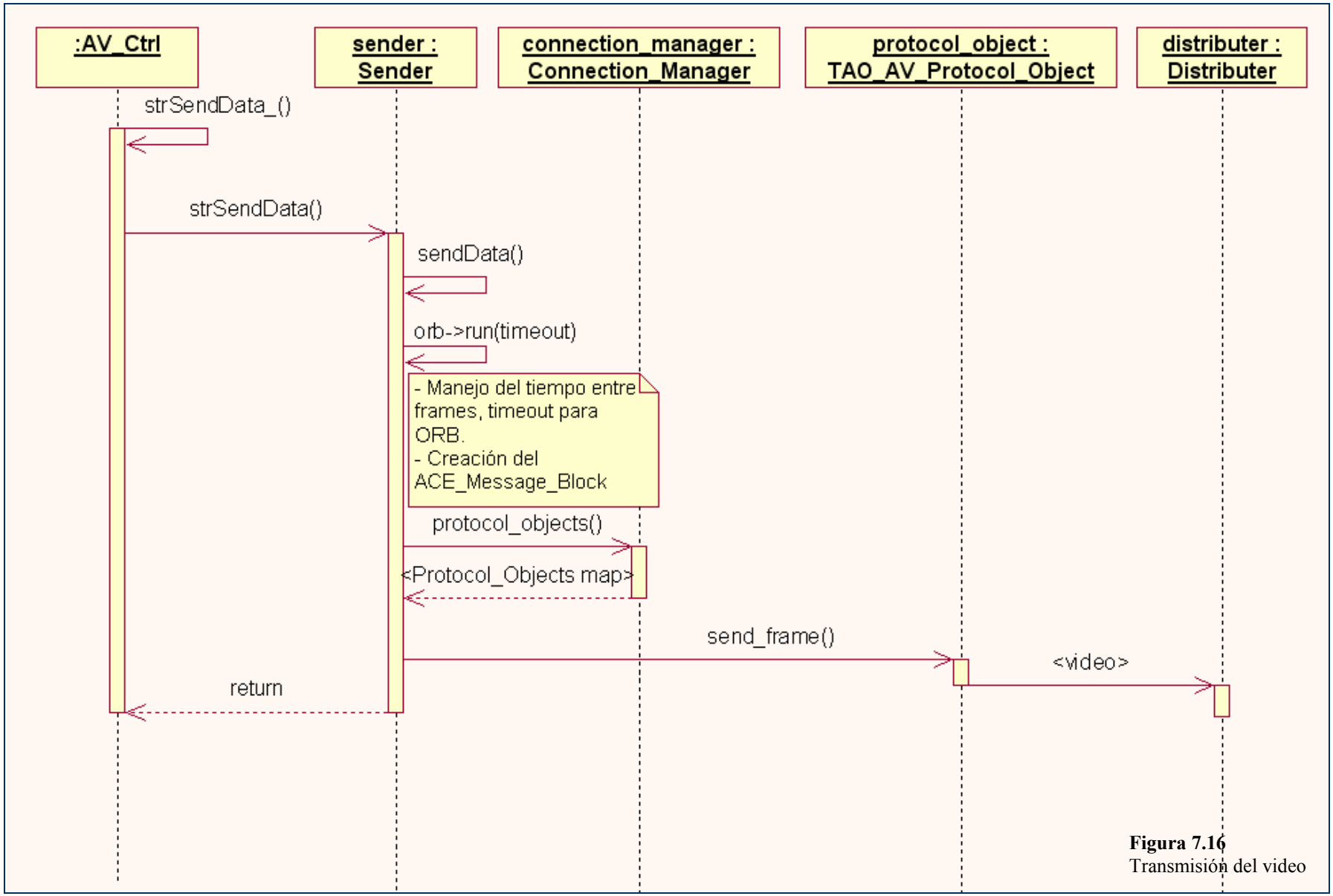


Figura 7.16
Transmisión del video

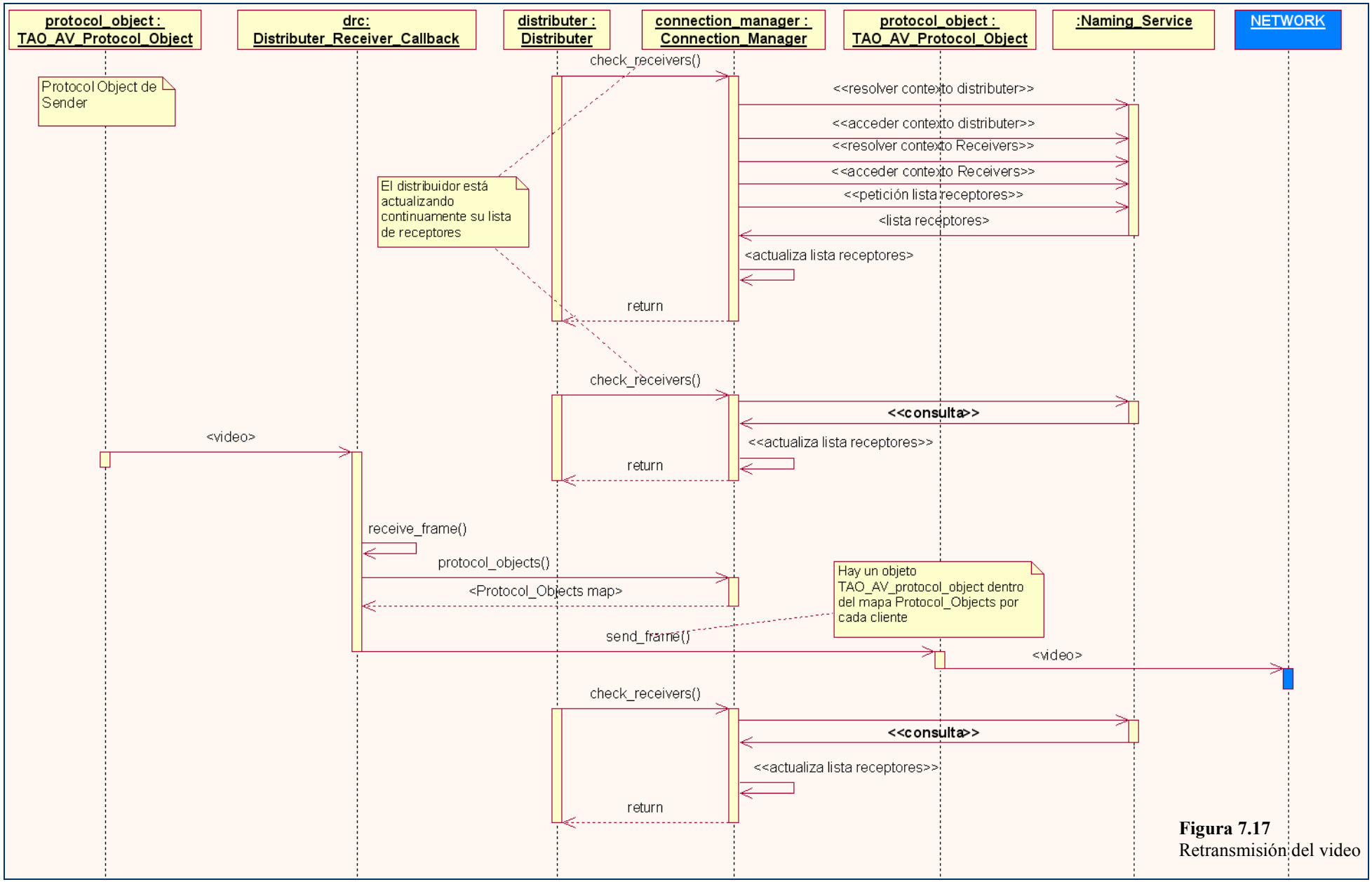


Figura 7.17
Retransmisión del video

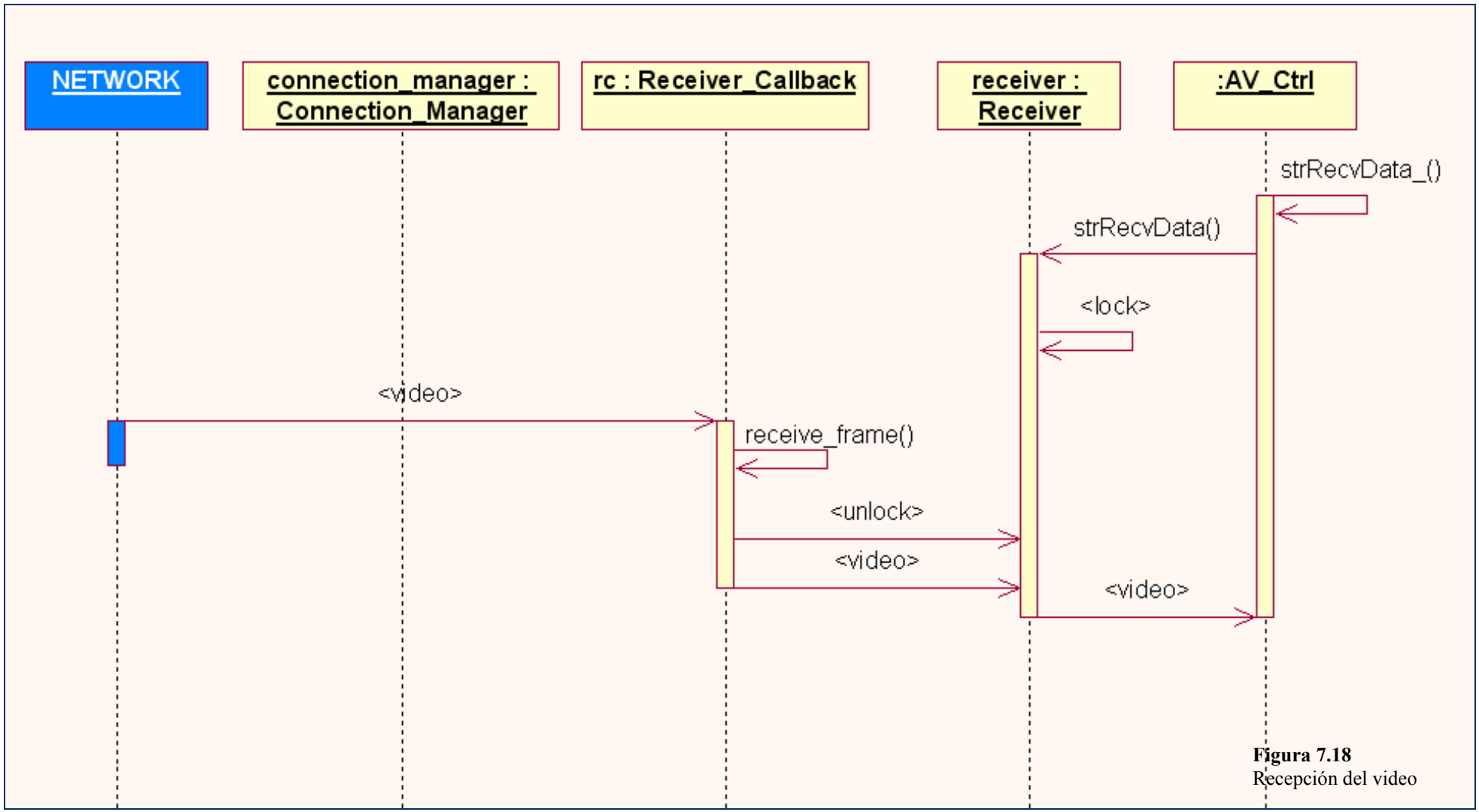


Figura 7.18
Recepción del video

7.6.5.3 Cierre de comunicación

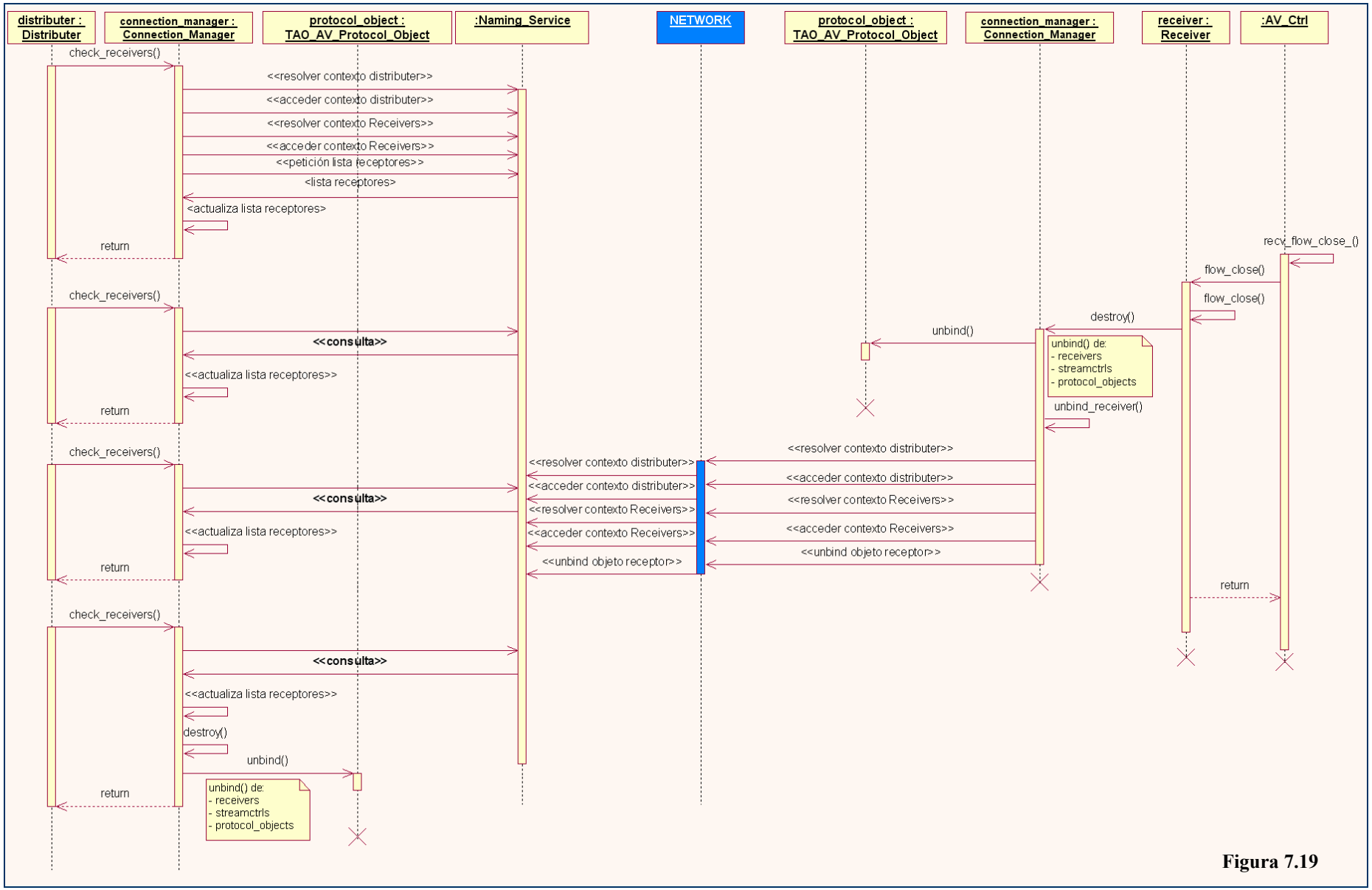


Figura 7.19

8. Compilación

Como ya se pudo apreciar en el apartado 2.6, *xawtv* tiene una estructura de compilación que permite capacidad de ampliación por medio de sus *Subdir.mk*. Estos archivos son de gran ayuda cuando se quieren añadir un *plugin* nuevo, o nuevas utilidades dentro de *xawtv*. Estas utilidades son el sistema de propiedades y el de controladores. A continuación se explica con detalle cómo se añadieron los *plugins* y utilidades, y cómo pueden añadirse otros nuevos.

8.1 *Makefile.in*

Se encuentra en el directorio raíz de *xawtv*. Los cambios en el fichero se reflejan en negrita.

```
                                    •••
# install paths
prefix          := @prefix@
exec_prefix     := @exec_prefix@
bindir          := $(DESTDIR)@bindir@
mandir          := $(DESTDIR)@mandir@
libdir          := $(DESTDIR)@libdir@/xawtv
libdir2         := $(DESTDIR)@libdir@/xawtv/properties
libdir3         := $(DESTDIR)@libdir@/xawtv/controllers
resdir          := $(DESTDIR)@resdir@
config          := @x11conf@/xawtvrc

                                    •••
```

En el bloque anterior se definen los directorios de instalación de las librerías compiladas de *xawtv*. Las variables **libdir2** (por lo general /usr/local/lib/xawtv/properties) y **libdir3** (/usr/local/lib/xawtv/controllers) contienen los directorios de instalación de propiedades y controladores respectivamente.

```
                                    •••
# CORBA libraries
ifndef TAO_ROOT
  TAO_ROOT = $(ACE_ROOT)/TAO
endif # ! TAO_ROOT
LDLIBS  := @LDLIBS@
LDLIBS += -L$(ACE_ROOT)/ace -lTAO_AV -lTAO_CosProperty -
lTAO_CosNaming -lTAO_Svc_Utils -lTAO_IORTable -
lTAO_PortableServer -lTAO -lACE

                                    •••
```

Las librerías de ACE y TAO necesarias para enlazar con el código controlador *AV_Ctrl*, durante su compilación.

```
                                    •••

# must come first
include $(srcdir)/common/Subdir.mk
# subdirs
include $(srcdir)/console/Subdir.mk
include $(srcdir)/debug/Subdir.mk
include $(srcdir)/libng/Subdir.mk
include $(srcdir)/man/Subdir.mk
include $(srcdir)/scripts/Subdir.mk
include $(srcdir)/vbistuff/Subdir.mk
include $(srcdir)/x11/Subdir.mk
include $(srcdir)/libng/plugins/controllers/corba/Subdir.mk
include $(srcdir)/libng/plugins/plugins_properties/Subdir.mk
include $(srcdir)/libng/plugins/controllers/Subdir.mk
include $(srcdir)/libng/plugins/Subdir.mk

# dependences
-include common/*.d
-include console/*.d
-include debug/*.d
-include jwz/*.d
-include libng/*.d
-include libng/plugins/*.d
-include libng/plugins/plugins_properties/*.d
-include libng/plugins/controllers/*.d
-include vbistuff/*.d
-include x11/*.d

                                    •••
```

Además, se incluyen los subdirectorios que contienen un fichero de compilación *Subdir.mk*, y se respeta el orden en que se añaden (es importante respetarlo). Se han añadido los directorios de propiedades, controladores y *plugins*.

Por último, se añaden todos los ficheros de dependencia contenidos en los directorios de *plugins*, propiedades y controladores.

8.2 *libng/Subdir.mk*

Se compilan todos los fuentes que se encuentran en el directorio */libng* y se añaden a la librería estática *libng.a*

```
OBJS-libng := \
      libng/grab-ng.o \
      libng/devices.o \
      libng/writefile.o \
      libng/color_common.o \
      libng/color_packed.o \
      libng/color_lut.o \
      libng/color_yuv2rgb.o \
      libng/convert.o \
      libng/plugins/utils.o

libng/libng.a: $(OBJS-libng)
      rm -f $@
      ar -r $@ $(OBJS-libng)
      ranlib $@

clean::
      rm -f libng.a
```

También se añade el fichero (compilado) de utilidades de la negociación *utils.o*.

8.3 *libng/plugins/Subdir.mk*

Incluye los *plugins* que deben compilarse y las librerías que deben enlazar.

```
# targets to build
TARGETS-plugins := \
     libng/plugins/flt-gamma.so \
     libng/plugins/flt-invert.so \
     libng/plugins/flt-smooth.so \
     libng/plugins/flt-disor.so \
     libng/plugins/conv-mjpeg.so \
     libng/plugins/read-avi.so \
     libng/plugins/read-net-avi.so \
     libng/plugins/write-avi.so \
     libng/plugins/write-net-avi.so

                  •••
```

Se han añadido los ficheros de *plugins write-net-avi* y *read-net-avi*.

```
                                •••

# libraries to link
libng/plugins/read-qt.so  : LDLIBS := $(QT_LIBS)
libng/plugins/write-qt.so : LDLIBS := $(QT_LIBS)
libng/plugins/read-net-avi.so : LDLIBS := libng/libng.a
libng/plugins/write-net-avi.so : LDLIBS := libng/libng.a

                                •••
```

Enlaza la librería estática *libng.a* con los dos nuevos *plugins*, y así permite hacer uso de las funciones que se definieron en *utils.c* entre otras.

8.4 *libng/plugins/plugins_properties/Subdir.mk*

Se usa para compilar todas las propiedades, además de poder instalarlas en el directorio definido por la variable **libdir2** y desinstalarlas.

```
# targets to build
TARGETS-plugins2 := \
     libng/plugins/plugins_properties/rate.so \
     libng/plugins/plugins_properties/compression.so \
     libng/plugins/plugins_properties/video_format.so \
     libng/plugins/plugins_properties/protocol.so

# global targets
all:: $(TARGETS-plugins2)

install::
     $(INSTALL_DIR) $(libdir2)
     $(INSTALL_PROGRAM) -s $(TARGETS-plugins2) $(libdir2)
     rm -f $(GONE-plugins2)

clean::
     rm -f $(TARGETS-plugins2)
```

8.5 *libng/plugins/controllers/Subdir.mk*

Empleado para compilar, instalar o desinstalar los controladores. Además incluye las librerías que deben enlazarse para cada controlador.

```
# targets to build
TARGETS-plugins3 := \
     libng/plugins/controllers/UDP_Ctrl.so \
     libng/plugins/controllers/RTP_Ctrl.so \
     libng/plugins/controllers/AV_Ctrl.so


# libraries to link

libng/plugins/controllers/RTP_Ctrl.so : LDLIBS :=
libng/plugins/controllers/rtp.lib

# global targets
all:: $(TARGETS-plugins3)

install::
     $(INSTALL_DIR) $(libdir3)
     $(INSTALL_PROGRAM) -s $(TARGETS-plugins3) $(libdir3)
     rm -f $(GONE-plugins3)

clean::
     rm -f $(TARGETS-plugins3)
```

8.6 *libng/plugins/controllers/corba/Subdir.mk*

Está implementado a modo de *script*. Hace llamadas a comandos de sistema para ejecutar *makefiles* y luego copiar los archivos compilados a ciertos directorios. Esto se debe a que dentro del subdirectorio *corba*, se encuentran archivos fuente implementados en C++ y han de ser compilados de forma distinta. Los *makefile* a ejecutar son los siguientes:

Makefile.stream-sender. Es un *makefile* para la compilación de los archivos *stream-sender.h* y *stream-sender.cpp*. Esta compilación enlaza los archivos con las librerías de *AVStreams* de TAO.

Los archivos se compilan generando una librería dinámica llamada *libstream_sender.so*.

Makefile.stream-receiver. De igual forma que en *Makefile.stream-sender*, se compilan los archivos *stream-receiver.h y stream-sender.cpp* usando las librerías de *AVStreams* de TAO.

Los archivos se compilan generando una librería dinámica llamada *libstream_receiver.so*.

Makefile.distributer. Sigue una secuencia parecida a la explicada anteriormente, pero esta vez se generará un archivo ejecutable en vez de una librería dinámica.

```
build:
      cd $(srcdir)/libng/plugins/controllers/corba/; $(MAKE) -f
            Makefile.stream-sender
      cd $(srcdir)/libng/plugins/controllers/corba/; $(MAKE) -f
            Makefile.stream-receiver
      cd $(srcdir)/libng/plugins/controllers/corba/; $(MAKE) -f
            Makefile.distributer

install::
      cd $(srcdir)/libng/plugins/controllers/corba/; $(MAKE) -f
            Makefile.stream-sender
      cd $(srcdir)/libng/plugins/controllers/corba/; $(MAKE) -f
            Makefile.stream-receiver
      cd $(srcdir)/libng/plugins/controllers/corba/; $(MAKE) -f
            Makefile.distributer

      rm -rf /usr/local/lib/xawtv/controllers/corba/
      rm -rf /usr/local/lib/xawtv/controllers/
      rm -rf /usr/local/lib/xawtv/

      mkdir /usr/local/lib/xawtv/
      mkdir /usr/local/lib/xawtv/controllers/
      mkdir /usr/local/lib/xawtv/controllers/corba/

      cp -f
$(srcdir)/libng/plugins/controllers/corba/libstream_sender.so*
            /usr/local/lib/xawtv/controllers/corba/
      cp -f
$(srcdir)/libng/plugins/controllers/corba/libstream_receiver.so*
            /usr/local/lib/xawtv/controllers/corba/
      cp -f $(srcdir)/libng/plugins/controllers/corba/distributer
            /usr/local/lib/xawtv/controllers/corba/
clean::
      rm -f
$(srcdir)/libng/plugins/controllers/corba/libstream_sender.so*
      rm -f
$(srcdir)/libng/plugins/controllers/corba/libstream_receiver.so*
      rm -f $(srcdir)/libng/plugins/controllers/corba/distributer

      rm -f
/usr/local/lib/xawtv/plugins/controllers/corba/libstream_sender.so*
      rm -f
/usr/local/lib/xawtv/plugins/controllers/corba/libstream_receiver.so*
      rm -f $(srcdir)/distributer

      rm -rf /usr/local/lib/xawtv/controllers/corba/
```

9. La Interfaz Gráfica *PiaGui*

9.1 Introducción

PiaGui es una interfaz de usuario realizada en Java **[19]**, muy útil a la hora de ejecutar la aplicación **`pia-net`** y que facilita tareas como asignar valores a propiedades de negociación, mantener listas con direcciones de servidores y otras opciones de usuario.

La forma de ejecutar esta aplicación es la siguiente:

```
java –jar PiaGui.jar
```

Al ser una aplicación desarrollada en *Java*, es necesario tener instalada la máquina virtual para la plataforma en la que se ejecute. La versión mínima para ejecutar *PiaGui* es **JRE 1.4**.

9.2 Paquetes y Clases

9.2.1 Paquete Graphics Components

Es donde se encuentran todas las clases que contienen una interfaz gráfica para el usuario y sus manejadores de eventos.
Las clases incluidas aquí siguen el siguiente esquema.

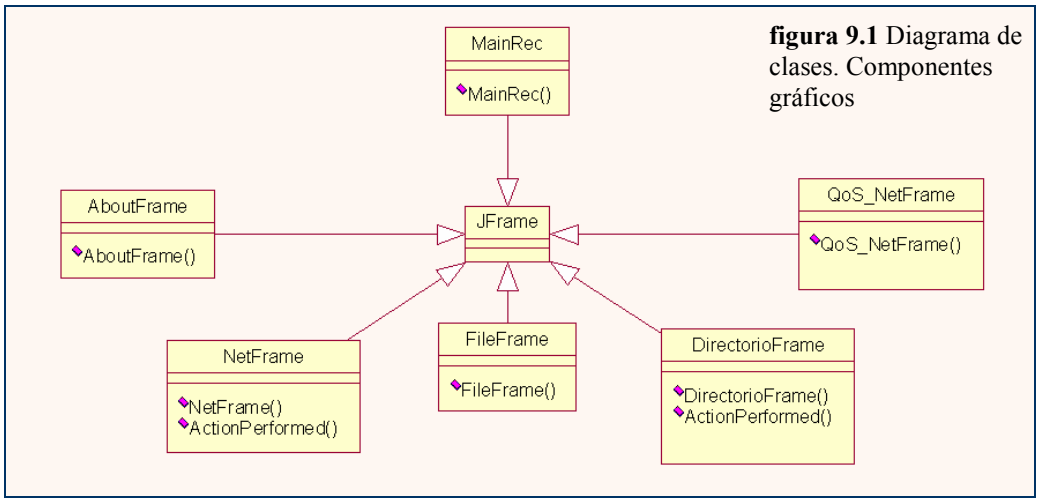


figura 9.1 Diagrama de clases. Componentes gráficos

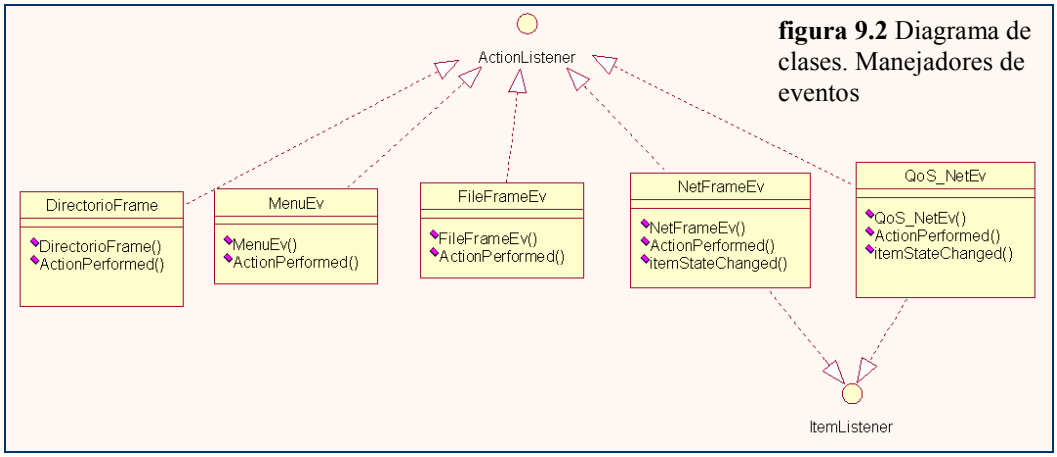


figura 9.2 Diagrama de clases. Manejadores de eventos

AboutFrame.class

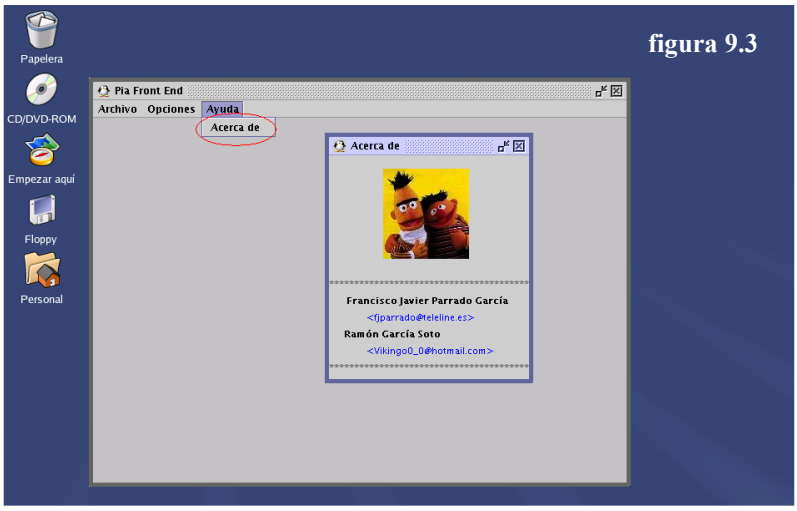


figura 9.3

Es un componente gráfico donde se pueden observar los créditos del programa. No es una clase importante.

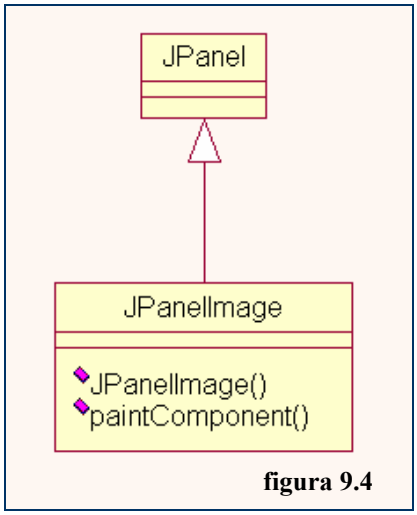
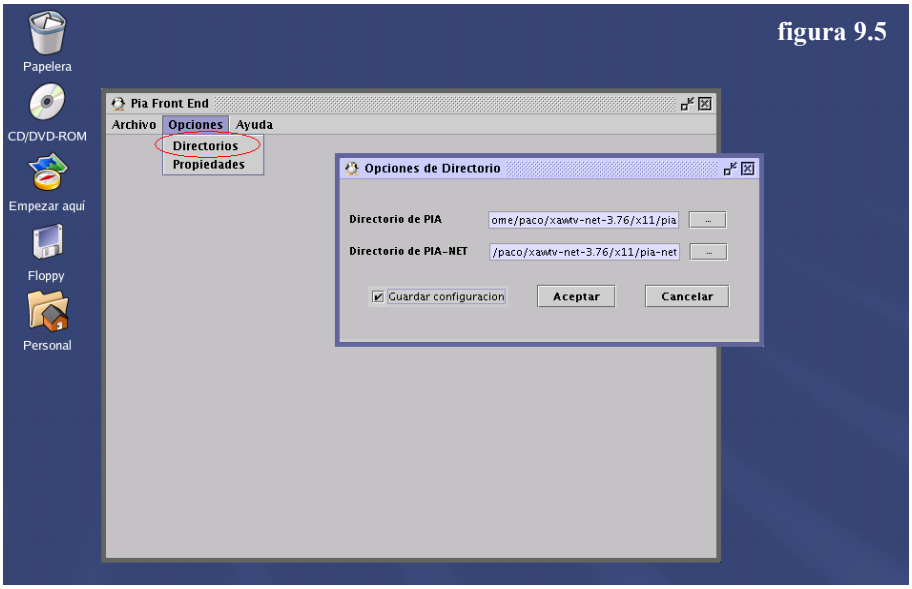


figura 9.4

En *AboutFrame.java* se define una clase llamada *JPanelImage* usada para insertar imágenes dentro de un *JPanel*.

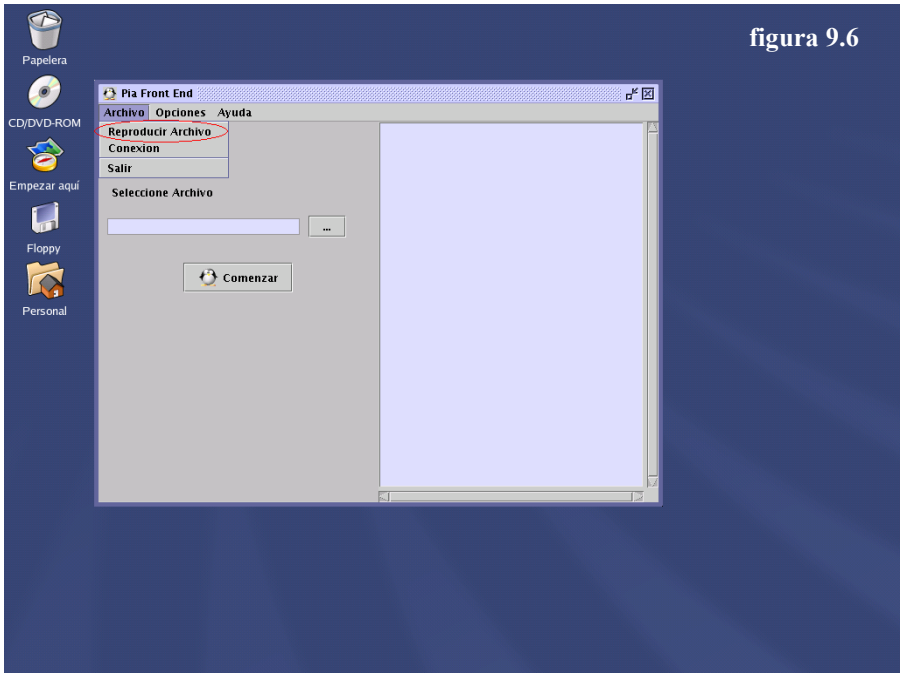
DirectorioFrame.class



Es un componente gráfico útil para localizar los archivos ejecutables **`pia`** y **`pia-net`**. Modifica para este propósito las variables **`DIR_PIA`** y **`DIR_PIA_NET`** contenidas por la clase *PiaGui*.

El manejo de los eventos lo realiza la misma clase.

FileFrame.class



Este *frame* se encarga de la reproducción de archivos locales de video usando el programa de consola **`pia`**. El manejo de los eventos generados lo realiza *FileFrameEv.class*

Esta interfaz gráfica consta de una zona para seleccionar un archivo por medio de una exploración y un área de texto a modo de *log*. Al pulsar el botón "comenzar" se genera un evento que lanza un *thread* encargado de la ejecución del proceso de reproducción de video.

FileFrameEv.class

Como ya se ha comentado antes, *FileFrameEv* es el manejador de eventos y su función más importante es la de lanzar un *Thread* de tipo *FileFrameThread* cuando se quiera comenzar a ejecutar `pia`.

MainRec.class

Esta clase contiene el *Frame* principal de nuestro programa. El manejo de los eventos generados por el menu lo realiza *MenuEv.class*

MenuEv.class

Se encarga de manejar los eventos de la barra de menú. Esta clase se crea nada más comenzar el programa y ésta, a su vez, crea un *frame* de cada tipo necesario para así agilizar el uso del programa. Esto es, creará un objeto de la clase *AboutFrame*, *DirectorioFrame*, *FileFrame*, *NetFrame* y *QoS_NetFrame*.
Al seleccionar una opción de menú relacionada con cualquiera de estos frames hará que este sea visible en la ventana de ejecución.

NetFrame.class

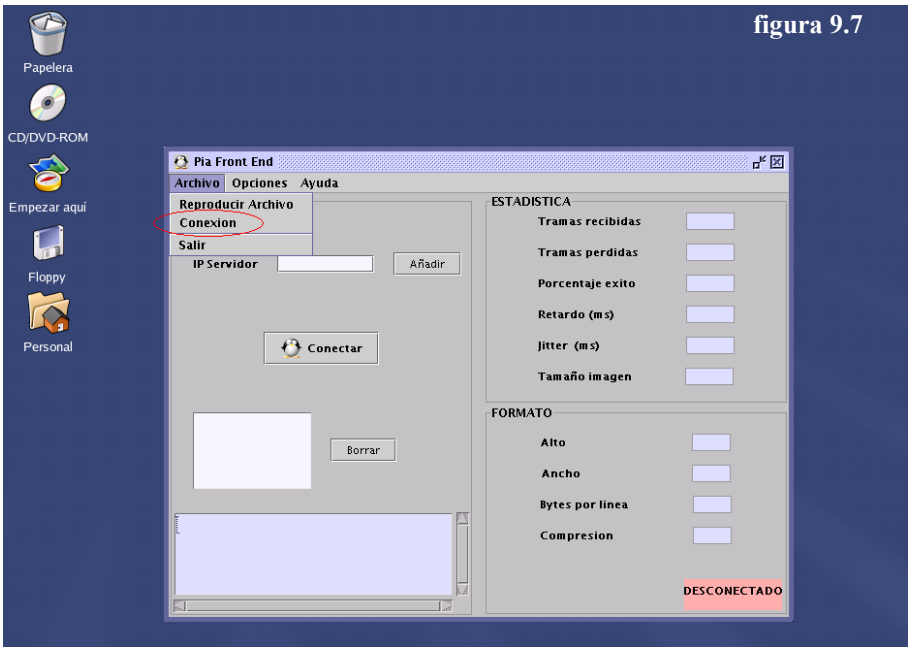


figura 9.7

Esta es una de las partes más importantes de esta interfaz. Es donde se puede observar la reproducción de video de forma remota. Consta de una zona de inicio de conexión y otra zona de captura de estadísticas, muy útiles a modo informativo.

Funciona de forma análoga a *FileFrame*. Al pulsar el botón "**Conectar**" se genera un evento que lanza un *thread* de tipo *NetFrameThread*, encargado de la ejecución del proceso de reproducción de video remoto.

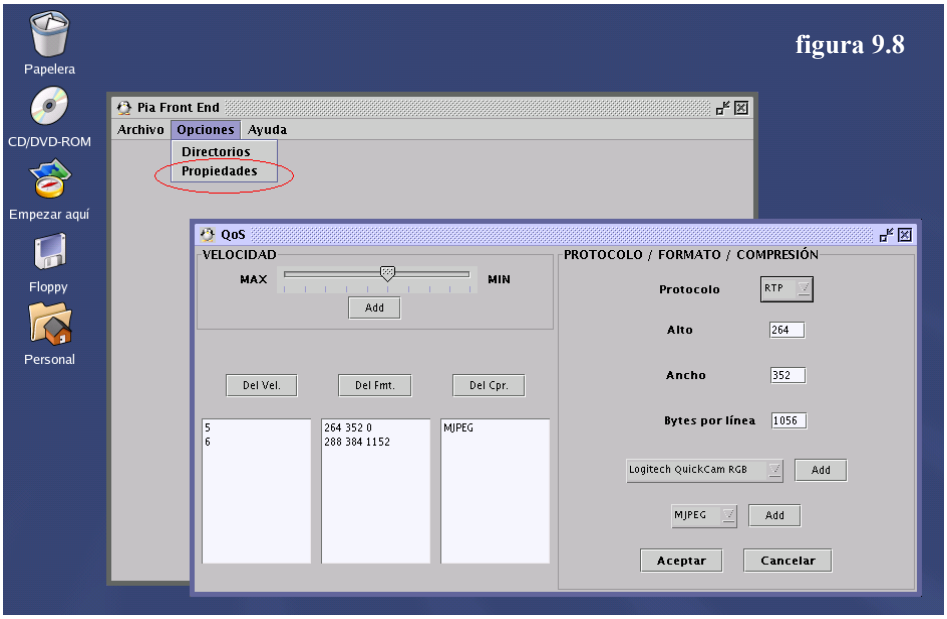
El peso del manejo de eventos se divide entre la propia clase *NetFrame* y una clase manejadora de eventos de tipo *NetFrameEv*.

Los eventos generados por el botón "**Conectar**" son tratados por *NetFrame*.

NetFrameEv.class

Este manejador se encarga de mantener la lista de direcciones IP por medio de archivos.

QoS_NetFrame.class



En este *Frame* es donde se pueden configurar todas las opciones a negociar disponibles en la versión actual de *xawtv*.

Está preparado para crear unas listas de características propias del video tal y como son compresión, alto, ancho y bytes por línea de imagen.

También se parametrizan cualidades de transmisión como *protocolo* y *rate*.
Todos estos parámetros son almacenados en un archivo (*Propiedades.dat*) para su posterior utilización por `pia-net`.

QoS_NetFrameEv.class

Es la clase encargada de manejar los eventos producidos por *QoS_NetFrame*.

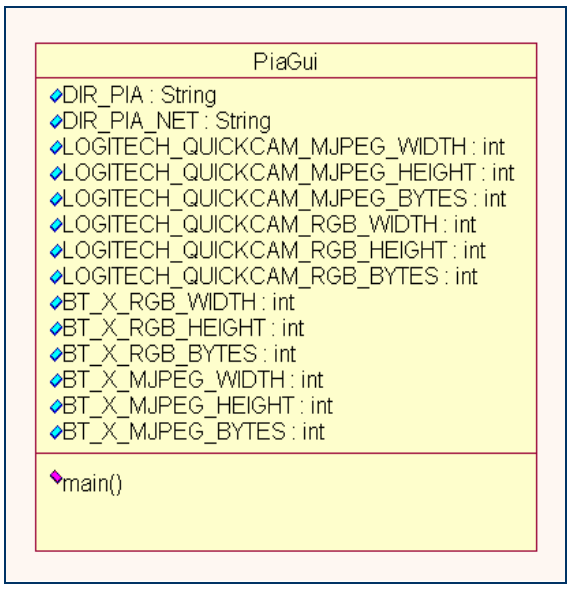
9.2.2 Paquete Images

Contiene archivos de imágenes usadas en la interfaz.

9.2.3 Paquete Main

PiaGui.class

Aquí se encuentra la función *main()* y todas las variables públicas estáticas utilizadas por las clases del programa. Estas son:



- Variables de rutas a ficheros, como ya se mencionó anteriormente almacenan la ruta hasta los ejecutables *pia* y *pia-net*.

- Variables de formato de dispositivos de captura de vídeo. Son usadas por *QoS_NetFrame*.

9.2.4 Paquete Threads

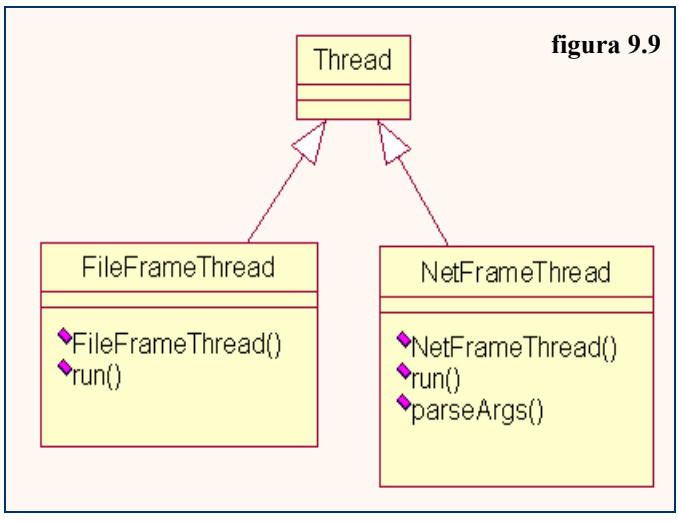


figura 9.9

Es el paquete que contiene a las clases encargadas de ejecutar los programas `pia` y `pia-net` en threads.

FileFrameThread.class

Es el thread lanzado cuando se quiere reproducir un archivo de video local. Lanza el programa *pia* y captura sus flujos de salida.

NetFrameThread.class

Es el *thread* lanzado al reproducir video de forma remota. Lanza el programa *pia-net* con los parámetros especificados en *NetFrame*, captura sus flujos de salida y actualiza las estadísticas en *NetFrame*.

La actualización de las estadísticas en la interfaz gráfica se realiza gracias a la captura de los flujos de salida que contienen unos valores etiquetados dependientes de cada tipo de mensaje que se quiera mostrar. Estas etiquetas son:

ETIQUETA	*TIPO DE DATO*	*DESCRIPCIÓN*
<CAN>	*int*	ancho de la imagen
<CAL>	*int*	largo de la imagen
<CBY>	*int*	bytes por línea de la imagen
<SCN>	Ninguno	Indica cuando a conectado el cliente
<FRN>	*int*	Número de *frame* enviado por el servidor desde que se arranca. Usando este dato se actualizan el número de frames recibidos y el porcentaje de éxito.
<TDS>	*int*	Retardo en milisegundos
<JIT>	*int*	Medición del *jitter*
<DRO>	*int*	Número de imágenes perdidas
<COD>	*string*	Indica el tipo de compresión
<ERR>	*string*	Indica un parámetro, formato o compresión, no aceptado por el servidor. Se deben usar valores por defecto e indicarlo en la interfaz.
<SIZ>	*int*	Tamaño de la imagen
<MSG>	*string*	Indica mensaje de *log*

9.2.5 Paquete utils

CargaDir.class

Realiza operaciones apertura y lectura de archivos. Según se utilice un constructor u otro:

CONSTRUCTOR	*OPERACIÓN*
CargaDir( )	Carga el fichero **Directorio.cfg**. Rellena las variables estaticas que señalizan el *path* de los ejecutables `pia` y `pia-net` en *DirectorioFrame*.
CargaDir(List)	Carga el fichero ***Servidores.dat***. Rellena la lista que se encuentra en *NetFrame* con las direcciones IP encontradas.
CargaDir(List, List, List)	Carga el fichero **Propiedades.dat**. Rellena las listas dentro *QoS_NetFrame* con las propiedades encontradas.

Es importante diferenciar cada una de estas tres operaciones ya que cada archivo contiene un formato distinto para su información, se verá más adelante dicho formato.

GuardaDir.class

Guarda los archivos de configuración. Como antes, según se use un constructor y otro:

CONSTRUCTOR	*OPERACIÓN*	*FORMATO ARCHIVO*
GuardaDir()	Guarda en el fichero **Directorio.cfg.** Rellenará el fichero con las variables estáticas encargadas de señalizar el path de los ejecutables.	`1 [ruta pia]` `2 [ruta pia-net]`
GuardaDir(List)	Guarda una lista de direcciones IP en el fichero **Servidores.dat**.	Cada línea del archivo es una dirección IP.

GuardaDir(List, List, List, Choice)	Guarda las propiedades dentro del fichero **Propiedades.dat**	`#1 RATE`
		`<valor1>`
		`...`
		`<valor10>`
		`#2 VIDEO_FORMAT`
		`<valor1>`
		`<valor2>`
		`<valor3>`
		`#3 COMPRESSION`
		`<valor1>`
		`...`
		`<valor10>`
		`#4 PROTOCOL`
		`<valor>`

TextFilter.class

Utilidad para exploración de carpetas en búsqueda de ficheros.

WindowCloser.class y *WindowCloser2.class*

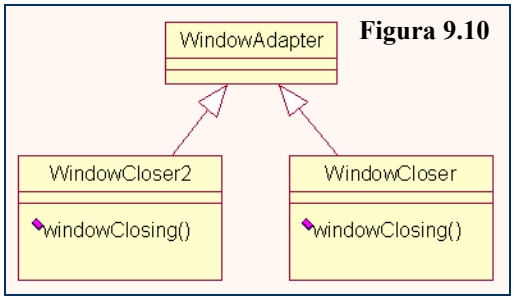


Figura 9.10

Manejador para cierre de ventanas.

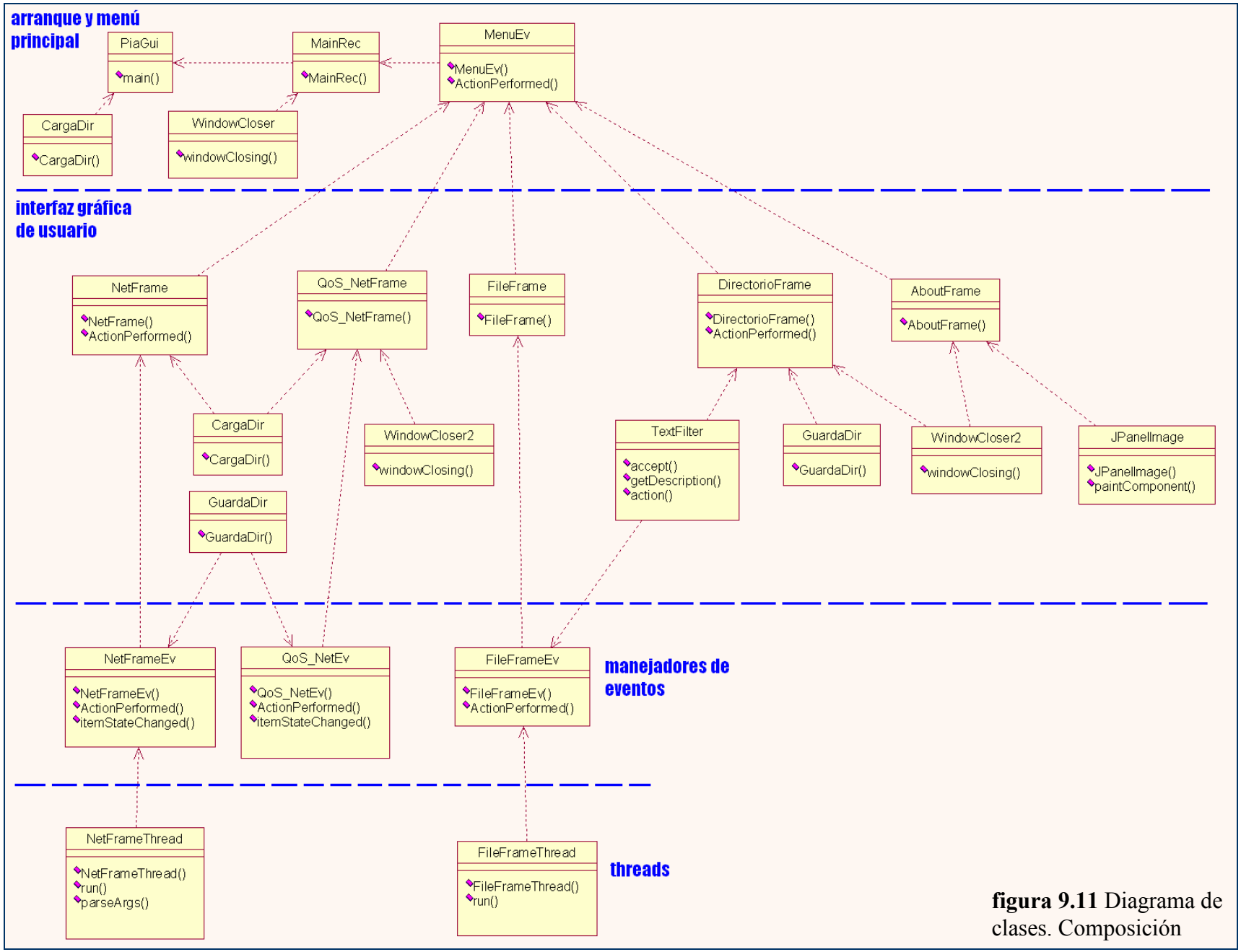


figura 9.11 Diagrama de clases. Composición

10 Conclusión y líneas futuras

Se ha llevado a cabo un trabajo que, a grandes rasgos, ha consistido en:

- **Buscar un software de captura y reproducción de vídeo** creado por otro autor, estudiarlo y decidir si era adecuado o no para tomarlo como punto de partida.

- **No modificar, sino ampliar** dicho software para que permita transmitir y recibir video en tiempo real. Se ha creado un *plugin* para la aplicación que permite transmitir video, y otro que permite recibirlo.

- Crear un servidor que permite **servir a varios clientes**, y tener control sobre cada conexión a través de una tabla interna de clientes.

- **Proveer calidad de servicio** mediante la **negociación de propiedades**:

    - Se ha establecido un sistema de creación y manejo de propiedades a través de implementación de interfaces.
    - Las **propiedades permiten**:
        - Utilizar **varios formatos de compresión de video**: MJPEG es ideal para transmisión en red local o incluso para conexión de alta velocidad a internet. Mientras que RGB24 es útil para aplicaciones de tratamiento de imágenes y visión artificial.
        - **Seleccionar** una **tasa de generación** de imágenes.
        - **Elegir el protocolo de transmisión** de video:
            - UDP
            - RTP
            - AV Streams
    - Cada propiedad se compila por separado y es automáticamente detectada y cargada por la aplicación.
    - El cliente asigna valores a las propiedades a través de una interfaz gráfica.

- Permitir al servidor **enviar video usando un protocolo de transmisión diferente para cada cliente al mismo tiempo** (no se conoce aplicación comercial que sea capaz hacerlo). Para ello se ha creado un sistema de creación y manejo de controladores a través de la implementación de interfaces:

    - Un único controlador permite definir todas las funciones necesarias para el establecimiento de conexión, envío y recepción de video entre cliente y servidor, usando un protocolo determinado.
    - Cada controlador se compila por separado y se detecta y carga de forma automática por la aplicación.

- **Diseñar una aplicación modificable y ampliable**: los *plugins* y los sistemas de propiedades y controladores son prueba de ello.

- **Crear una aplicación estable:**
    - El servidor borra a un cliente de su tabla si éste se ha desconectado. Detectándolo:
        - Si fracasa el envío de video.
        - A través de mensajes de control que intercambian cliente y servidor.
    - Se establece un tiempo mínimo de espera entre conexiones simultáneas. En caso contrario, esto podría provocar errores.
    - Si el cliente intenta negociar una propiedad que no posee el servidor (o viceversa), es el cliente el que se desconecta, nunca el servidor.

- **Ampliar el sistema de compilación** de la aplicación de la que se parte. Se han modificado y añadido nuevos ficheros de compilación que permiten compilar, instalar o desinstalar los nuevos *plugins* creados, al mismo tiempo que se compila, instala o desinstala la aplicación principal. Ha sido preciso un estudio de la estructura de compilación original para llevarlo a cabo, como también se ha estudiado con detenimiento el código fuente de la aplicación.

- **Desarrollar una interfaz gráfica** que permite al cliente establecer parámetros de conexión y **obtener estadísticas a tiempo real**: la aplicación de recepción de video escrita en lenguaje C y la interfaz gráfica programada en JAVA se comunican con objeto de que todo sea más fácil para el usuario.

En los siguientes apartados se detallan posibles líneas futuras de trabajo.

10.1 Formatos de vídeo y fragmentación de paquetes.

El controlador UDP permite la transmisión de vídeo usando los formatos de compresión de vídeo MJPEG y RGB24 como ya se vió en el apartado 6.3.1. El resto de controladores tan sólo permiten MJPEG, pues no necesita fragmentar la imagen en varios paquetes y consume poco ancho de banda.

Ambos formatos no son los idóneos para la transmisión de vídeom, lo ideal es el uso de *streaming video*: flujos multimedia que el cliente almacena en un *buffer* y los reproduce. Es un flujo en el que no tiene sentido el concepto de “fragmentación de imágenes”, sino que se trata de un flujo de datos de vídeo que se envían en paquetes. Formatos de vídeo como *RealMedia* (*rm*) o *Advanced Streaming Format* **[20]** (*asf*) de Microsoft son más adecuados.

Aún así, aunque no esté implementado, los *frames* que se envían en formato MJPEG pueden fragmentarse. Para otros controladores que no sean UDP también puede implementarse fragmentación RGB24, la imagen se fragmenta usando vectores de memoria, pero ya no se puede enviar información sobre la fragmentación en una cabecera (de cuántos paquetes se compone una imagen, cuántos *bytes* se van a recibir…), puesto que AVStreams tiene sus propias cabeceras.

Esto último no es problema, para AVStreams el tamaño de paquete de transmisión es fijo, es decir, aunque la imagen (o el fragmento) no ocupara todo el paquete, éste se rellena con bits de *padding*, por lo que el cliente sabe cuántos *bytes* va a recibir en cada paquete. En recepción, el cliente puede diferenciar en cada paquete los campos de datos y los de información. A partir de esa información se obtiene el número de *bytes* relacionados con la imagen que contiene el paquete.

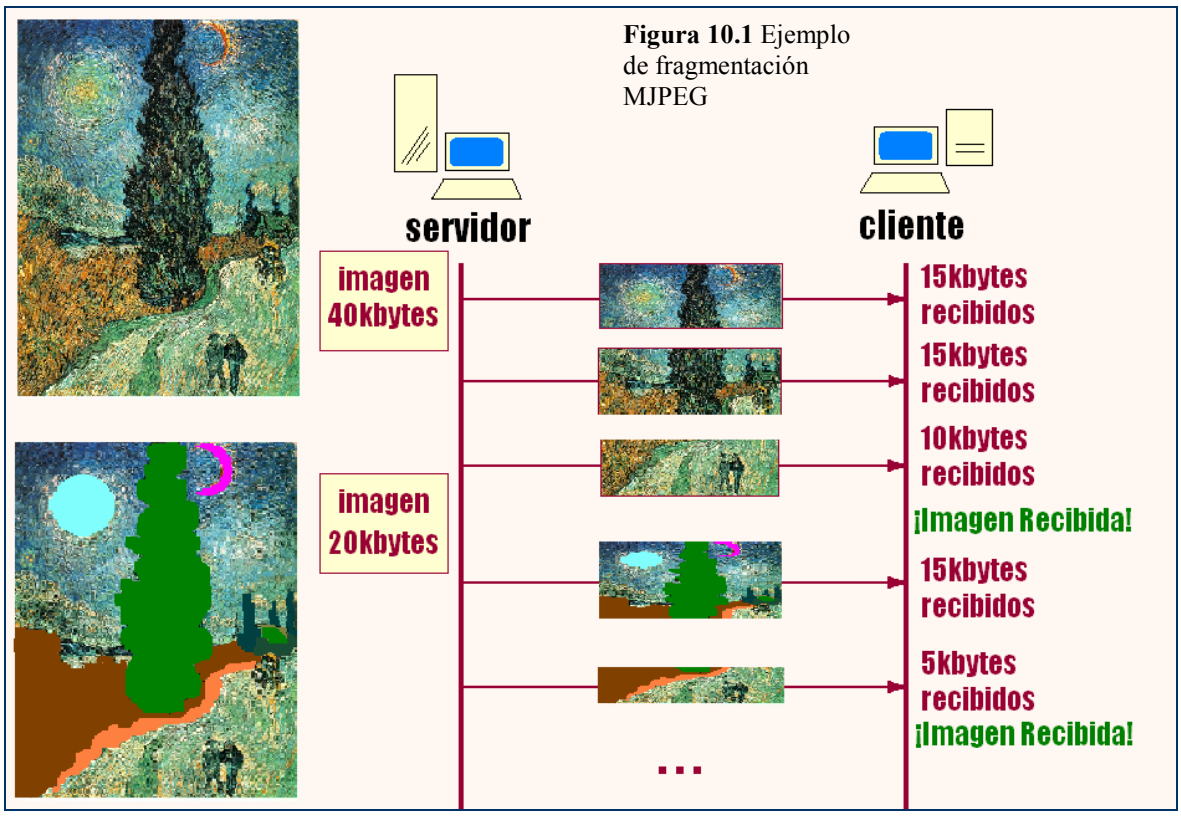


Figura 10.1 Ejemplo de fragmentación MJPEG

Por ejemplo (ver figura 10.1), si el tamaño máximo de información de datos que puede llevar un paquete es de 15 *Kbytes* y una imagen fuera de tamaño 40 *Kbytes*, el cliente recibiría dos paquetes que contendrían fragmentos de 15 *Kbytes* y un último paquete con 10 *Kbytes* de imagen. Para saber de cuántos paquetes se compone una imagen es fácil, el cliente conoce el tamaño máximo de *bytes* destinados a imagen que permite cada paquete, si estos *bytes* están ocupados en su totalidad, se sabe que se trata de un fragmento, si en cambio sólo sólo están ocupados en parte, se sabe que se trata del último. Así, el cliente va almacenando fragmentos hasta que detecta el último y recompone la imagen.

El único problema que puede darse, es que existe una probabilidad (aunque muy pequeña) de que la imagen sea múltiplo del tamaño de los datos que porta un paquete RTP, y el cliente nunca pueda detectar cuál es el último fragmento. En realidad, como ya se ha dicho, lo ideal es el uso de *streaming video*, sobre todo para protocolos de transmisión a tiempo real, pero si se opta por otros formatos de compresión, ésta puede ser una solución válida, aunque no la mejor.

10.2 Negociación durante la transmisión y recepción de video

Sólo se permite negociar parámetros de la transmisión en el instante de conexión. Sería interesante poder renegociar nuevos parámetros durante la recepción de vídeo sin tener que desconectar el cliente.

Esto puede ofrecer muchas ventajas, y no es difícil de implementar. Un cliente, cada vez que desee renegociar sus parámetros, puede realizar una nueva conexión TCP con el servidor, éste comprueba en su tabla si está ya conectado, vuelven a negociar y se actualizan los valores. Las razones por las que no se ha implementado son:

- Si el cliente decide durante la transmisión cambiar el protocolo de recepción de vídeo, la conexión debe reiniciarse.
- *xawtv* no permite transmitir usando dos formatos de compresión al mismo tiempo. Si un cliente está recibiendo vídeo MJPEG, aunque desee recibir RGB24, no lo hará. Para permitir esto, se tendrían que realizar cambios en el programa *xawtv.*
- *xawtv* sólo captura de un único dispositivo. La captura es en un formato de vídeo determinado, para que pueda capturar desde dos dispositivos diferentes debe modificarse el programa *xawtv*. Si el cliente recibe en un formato determinado y desea cambiar las dimensiones del vídeo, no podrá hacerlo.
- Tan sólo la propiedad *rate* podría actualizarse a tiempo real sin requerir reconexión.
- Se debe crear un nuevo mecanismo de comunicación entre la interfaz gráfica *PiaGUI* y *pia-net* para que este último pueda detectar los nuevos valores que se introducen a través de la interfaz a tiempo real. *pia-net* puede consultar cada cierto tiempo si ha habido cambios en el fichero *Propiedades.dat*.

10.3 Audio

Los *plugins read-net-avi.c* y *write-net-avi.c* y los controladores no implementan funciones de envío y recepción de audio. Pueden implementarse esas funciones, pero hay que tener en cuenta:

- El audio sin **comprimir** ocupa mucho ancho de banda, debe usarse un formato de compresión que reduzca el tamaño de los *frames* de audio sin perder calidad.
- En recepción, el audio debe **reproducirse de forma fluida**, es mucho más molesto escuchar un sonido que se entrecorta que un video que no es fluido.
- Deben **sincronizarse audio y vídeo**. Los paquetes de vídeo y audio pueden enviarse por separado (por ejemplo, si se crea una sesión RTP para audio y otra para vídeo) y sincronizarse a través de sus *timestamps*, o también puede enviarse la información de audio y vídeo en un mismo paquete (por ejemplo en el caso de utilizar el protocolo UDP).
- Es preferible **evitar la fragmentación** de los *frames* de audio (por ejemplo, enviándolo en paquetes distintos a los del video), pero en el caso de que haya que hacerlo, la sincronización puede ser muy compleja.
- La **calidad** del sonido puede negociarse, o incluso un cliente puede no desear audio. Deben definirse nuevas propiedades.

10.4 Ampliación de funcionalidad del Controlador AV

10.4.1 Negociación de propiedades

Como ya se comentó anteriormente, el controlador AV es de tipo *"NOTCP"*, por lo cual llama a una función de negociación manual. Esto impide la negociación con el servidor de importantes parámetros que no tienen porque ser conocidos por el cliente. Por ello, una importante mejora sería la inclusión de una negociación al margen de TCP.

Dicha negociación se podría realizar a través del *Property Service* y *Trading Service* que nos ofrece TAO. Este primer servicio permite asociar propiedades a objetos de forma dinámica. El segundo ofrece una secuencia de referencias de objetos basada en una propiedad que todos ellos comparten. Además, también se deberían utilizar las características de *Virtual Device* (VDev), muy útiles para este propósito.

Los comentarios en este apartado son posibles aproximaciones, no se ha podido profundizar en este tema debido a la escasa documentación que se posee.

10.4.2 Mejora en la información ofrecida durante la transmisión

En la versión actual del controlador AV existe escasa información sobre los *frames* que han sido enviados o recibidos. Simplemente se puede llevar la cuenta de cuantos fueron y el tamaño de la imagen que se envió. Sería interesante poder obtener información sobre otros aspectos de la transmisión. Un ejemplo sería la obtención de estampas temporales asociadas a cada *frame*.

Se ha observado la existencia de clases, como **`TAO_AV_frame_info`**, que ofrecen información adicional. Dicha información resulta realmente útil a la hora de solventar problemas, como podría ser el *jitter*, y proporcionar estadísticas en tiempo real sobre el rendimiento, como medición de retardos.

10.4.3 Elección de distintos protocolos en AVStreams

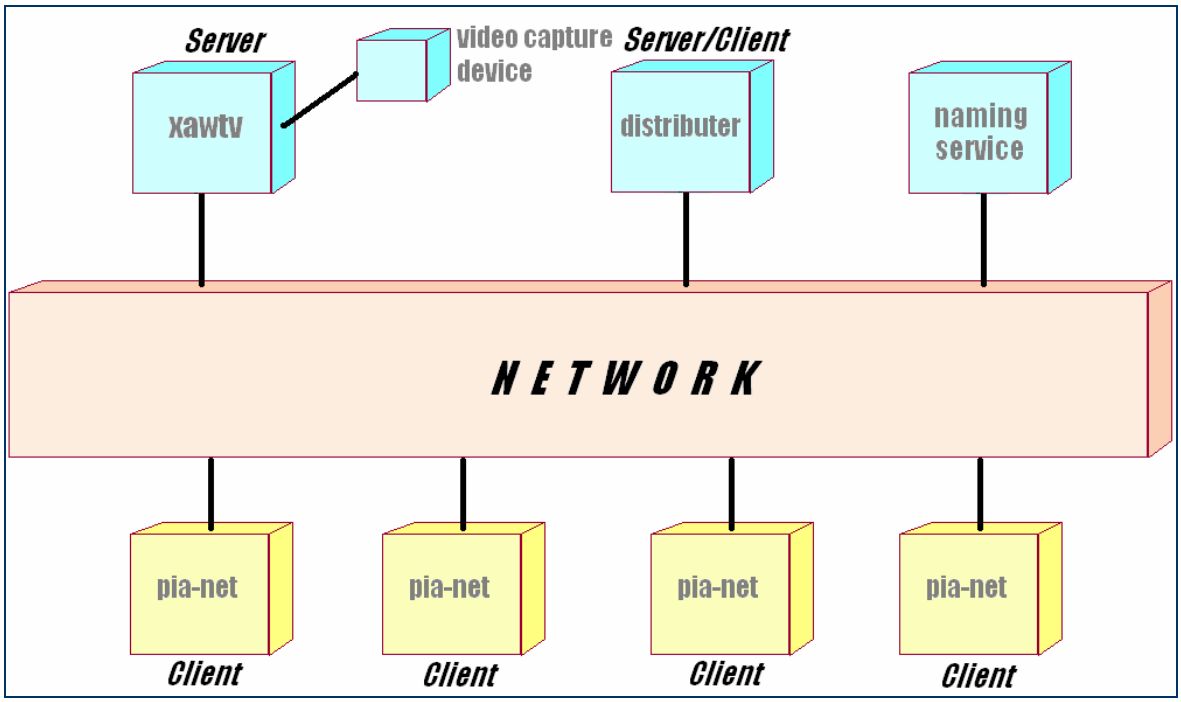
Recordando lo visto en el apartado 7.3.2.6, sería útil en un futuro poder usar la implementación del Controlador AV teniendo la opción de parametrizar el protocolo sobre el que se desea transmitir el video. En este caso, las opciones aumentarían. Por un lado se podrían usar los controladores implementados. Por otro lado se podría usar el controlador AV con tres protocolos distintos. Se podría transmitir usando *AVStreams* sobre UDP, TCP y RTP.

10.4.4 Mejora en la estructura de comunicación

La implementación actual es físicamente muy poco flexible. Encontramos dentro de la misma máquina el proceso encargado de generar video, el proceso distribuidor y el proceso encargado del Servicio de Nombrado. Genera un problema debido a la gran carga de procesado y de flujo de datos en dicha máquina, además de no poder crear estructuras con varios servidores y distribuidores.

Un cambio interesante a realizar sería la separación de estos tres procesos en máquinas diferentes. Agilizaría el procesado de datos y haría que no hubiera tanto tráfico entrante/saliente en una sola máquina. En principio, estaba totalmente diseñado para que fuera así, cada proceso arrancado manualmente y de forma independiente uno de otro. Por facilitar el uso de la aplicación, se automatizó todo de forma que al arrancar *xawtv* fueran lanzados todos los procesos necesarios.

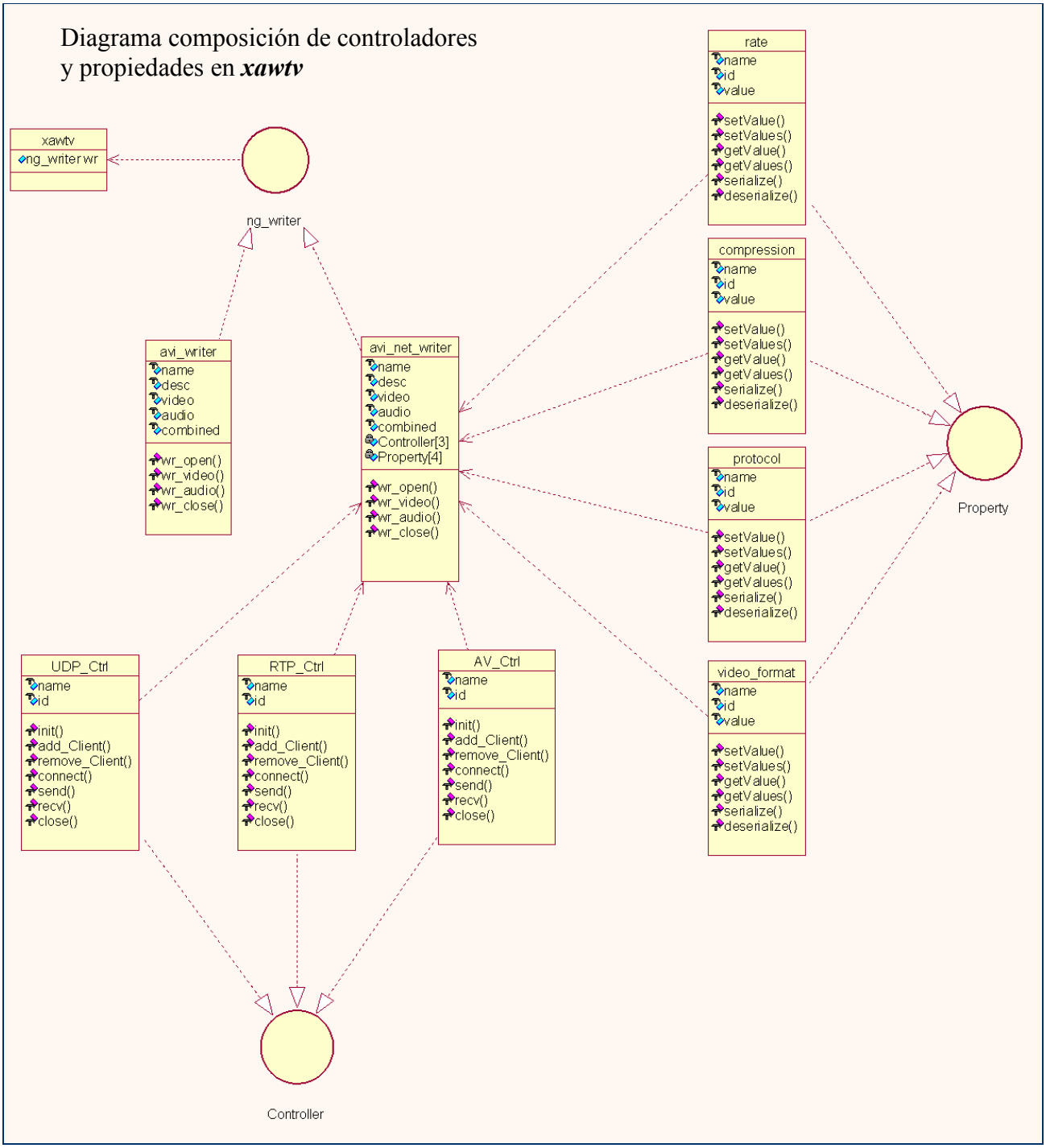
Figura 10.2 Diagrama de despliegue



De esta forma, podrían usarse varios distribuidores y varias fuentes de video, aunque para ello habría que realizar también modificaciones en el código de ambos.

ANEXO I

Diagramas de *xawtv* y *pia-net*



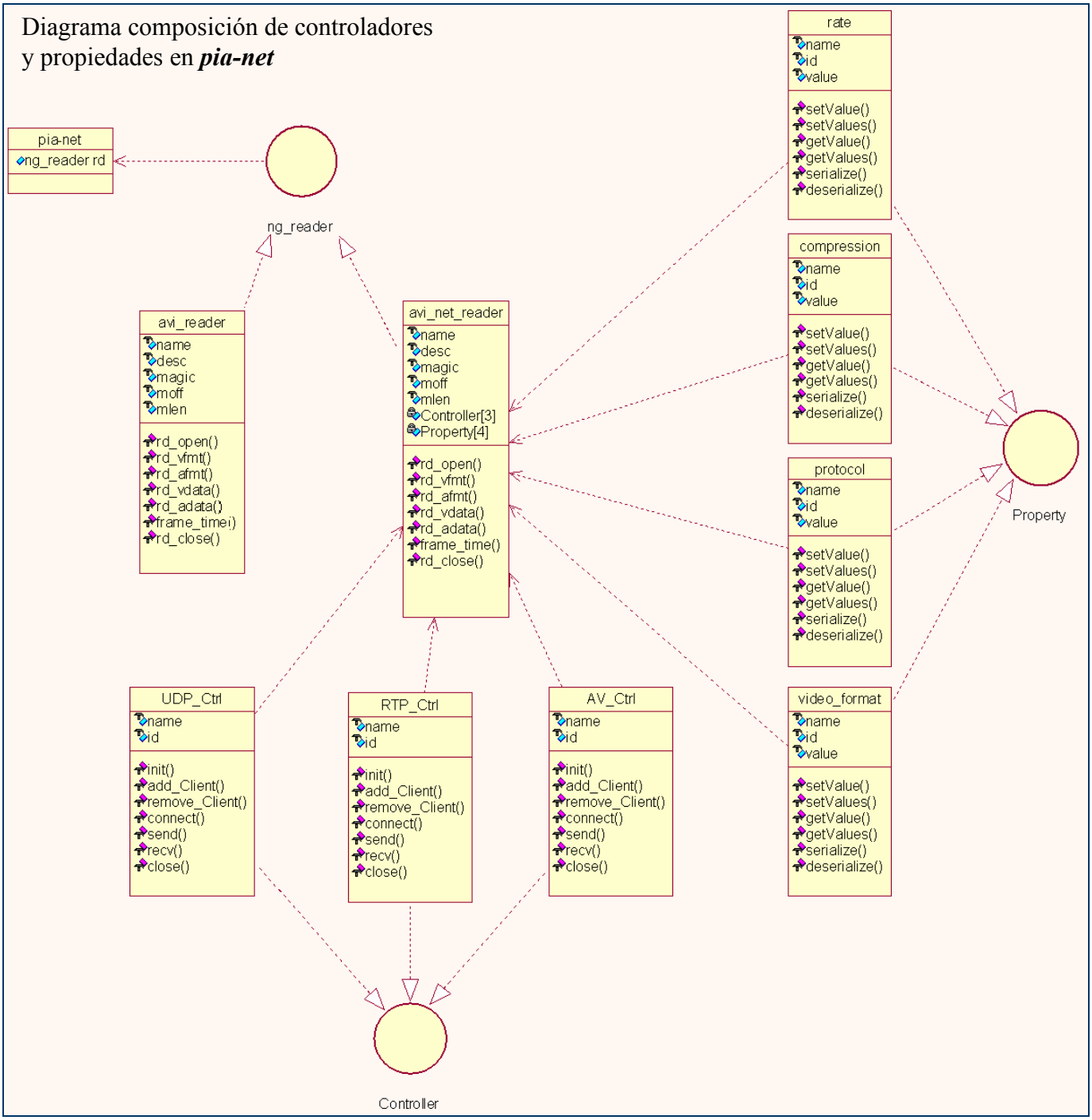
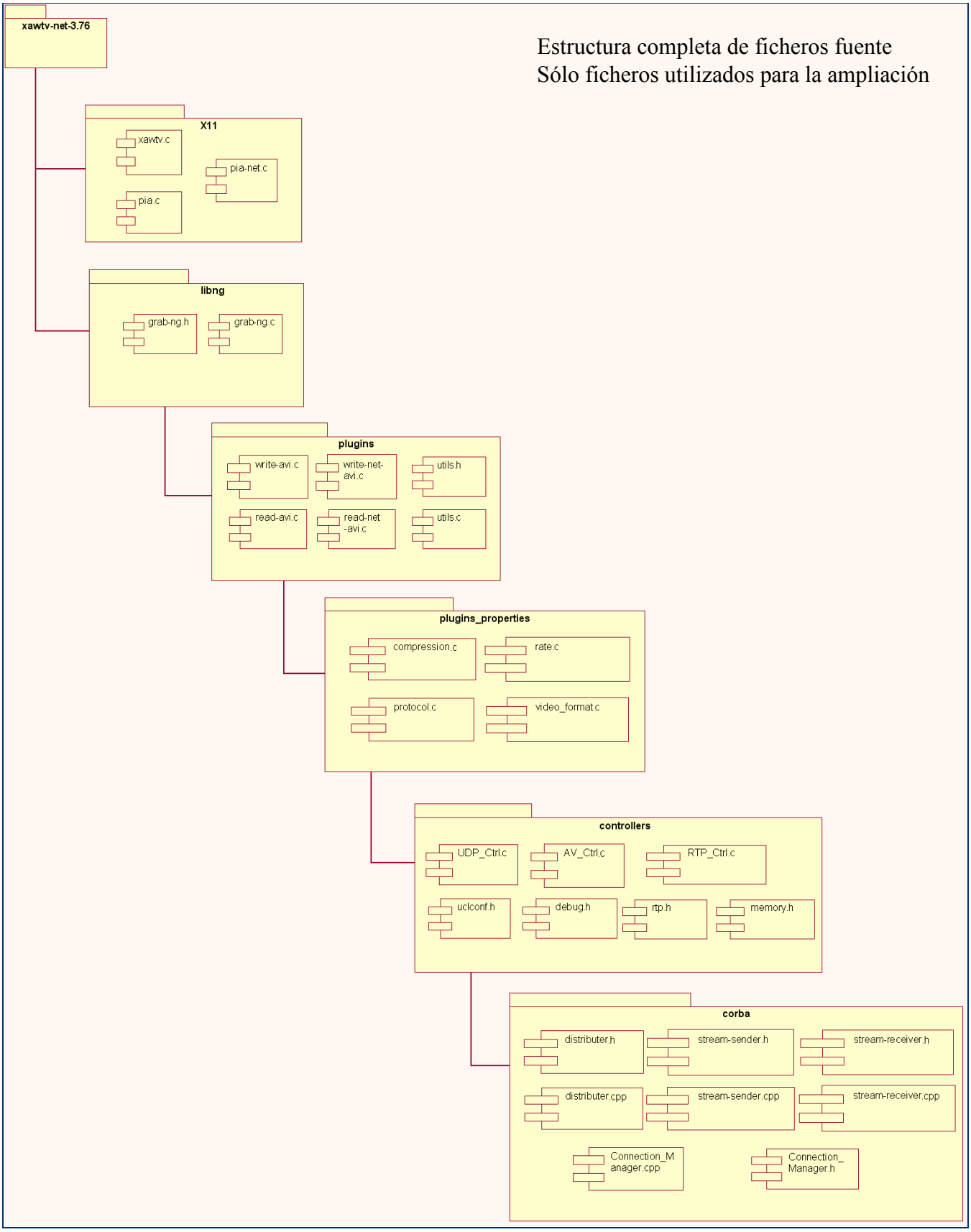
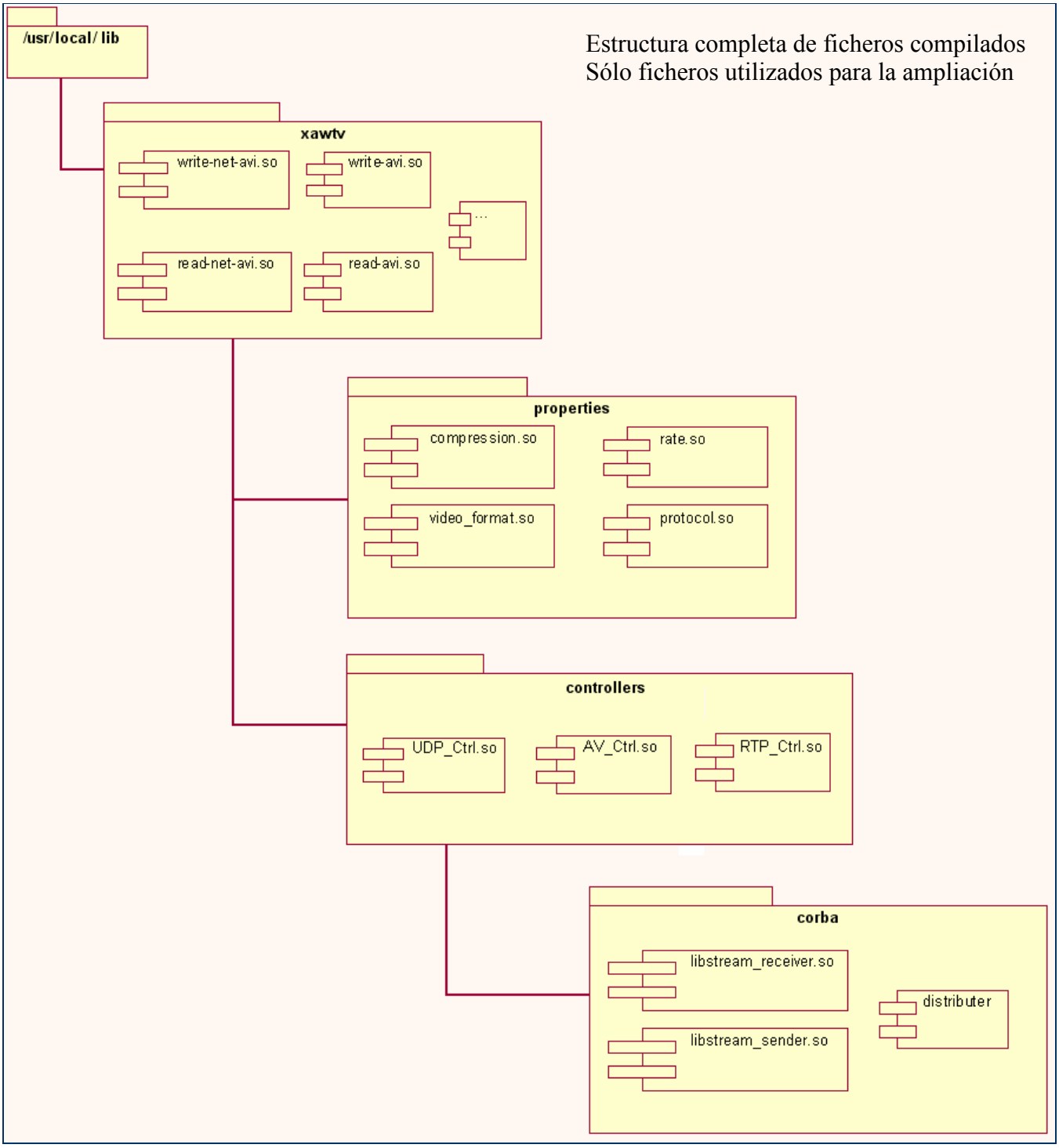


Diagrama composición de controladores
y propiedades en ***pia-net***
pia-net
ng_reader rd
ng_reader
avi_reader
name
desc
magic
moff
mlen
rd_open()
rd_vfmt()
rd_afmt()
rd_vdata()
rd_adata()
frame_time()
rd_close()
avi_net_reader
name
desc
magic
moff
mlen
Controller[3]
Property[4]
rd_open()
rd_vfmt()
rd_afmt()
rd_vdata()
rd_adata()
frame_time()
rd_close()
rate
name
id
value
setValue()
setValues()
getValue()
getValues()
serialize()
deserialize()
compression
name
id
value
setValue()
setValues()
getValue()
getValues()
serialize()
deserialize()
protocol
name
id
value
setValue()
setValues()
getValue()
getValues()
serialize()
deserialize()
video_format
name
id
value
setValue()
setValues()
getValue()
getValues()
serialize()
deserialize()
Property
UDP_Ctrl
name
id
init()
add_Client()
remove_Client()
connect()
send()
recv()
close()
RTP_Ctrl
name
id
init()
add_Client()
remove_Client()
connect()
send()
recv()
close()
AV_Ctrl
name
id
init()
add_Client()
remove_Client()
connect()
send()
recv()
close()
Controller

Estructura completa de ficheros fuente
Sólo ficheros utilizados para la ampliación




/usr/local/lib
Estructura completa de ficheros compilados
Sólo ficheros utilizados para la ampliación
xawtv
write-net-avi.so
write-avi.so
...
read-net-avi.so
read-avi.so
properties
compression.so
rate.so
video_format.so
protocol.so
controllers
UDP_Ctrl.so
AV_Ctrl.so
RTP_Ctrl.so
corba
libstream_receiver.so
distributer
libstream_sender.so

ANEXO II

Multiplexación E/S síncrona

La multiplexación de E/S es necesaria cuando se pretenden mantener transferencias de datos entre el servidor y varios clientes por un mismo puerto. En este caso la transeferencia se realiza usando sockets UDP.

Linux ofrece un conjunto de utilidades. Éstas son las siguientes:

```
int select  (int n, fd_set *readfds, fd_set *writefds,
             fd_set *exceptfds, struct timeval *timeout)
```

`select()` espera a que una serie de descriptores de ficheros, en este caso descriptores de conexiones por *socket*, cambien su estado. Estos descriptores se encuentran dentro de tres grupos dependiendo de para qué se quieran usar. Estos grupos son:

- **`*readfds`**, En **`readfds`** serán observados para comprobar si hay caracteres que llegan a estar disponibles para lectura.
- **`*writefds`**, En **`writefds`** serán observados para ver si es correcto escribir inmediatamente en ellos.
- **`*exceptfds`**, Los que se encuentren en **`exceptfds`** serán observados por si ocurren excepciones.

Las tres variables son del tipo **`fd_set`**.

El entero **`n`** indica cuál es el descriptor de *socket* con mayor valor (más uno) de los tres grupos.

El último valor **`struct timeval *timeout`** se usa para control de *timeout* y retardos.

Para modificar las variables de tipo **`fd_set`** se proporcionan cuatro macros:

- **FD_ZERO**, limpia un conjunto.
- **FD_SET**, añade un descriptor al conjunto.
- **FD_CLR**, borra un descriptor del conjunto
- **FD_ISSET**, comprueba si un descriptor es parte del conjunto.

ANEXO III

Instalación de cámara Logitech Quickcam Web en Redhat 8

No existen drivers oficiales para *Linux* de la webcam *Logitech Quickcam Web*. Se han obtenido de la página http://gd.tuwien.ac.at/publishing/sf/q/qce-ga/ descargando el fichero *qce-ga-0.40c.tar.gz.*

Antes de instalar los drivers hay que asegurarse de que el SO (sistema operativo) ha detectado la cámara. Para ello hay que conectar la webcam USB mientras Linux está en funcionamiento, y ejecutar el comando

```
less /proc/bus/usb/devices | grep Camera
```

el resultado de esta ejecución debe ser como sigue (o parecido):

S: Product=USB Camera

en ese caso la cámara ha sido detectada.

Una vez que se está seguro de que el SO reconoce la cámara (si no la reconoce hay que configurar y recompilar el kernel), se descomprime el fichero que contiene los drivers:

```
tar xvfz filename.tar.gz
```

Se crea el directorio *qce-ga-0.40c*. Moverlo a */usr/src,* para ello se deben tener permisos de *root*.

Ahora se debe acceder al directorio donde se movieron los ficheros que contienen los *drivers* y se observa que existe un enlace simbólico llamado */usr/src/linux-2.4*. Crear un nuevo enlace simbólico llamado *linux* mediante el comando

```
ln -s linux-2.4 linux
```

Teclear *exit* para dejar de ser *root*.

Se procede a instalar el *driver*. Entrar al directorio */usr/src/qce-ga-0.40c* y ejecutar

```
make
```

para compilar los fuentes. Se ha compilado archivo llamado *mod_quickcam.o* (o similar).

Otra vez con permisos de *root,* escribir

```
chown root:root mod_quickcam.o
```

para que el fichero pase a pertenecer al usuario *root* y sea más seguro.

Se debe editar (aún como *root*) el archivo */etc/modules.conf* con un editor de texto. Si no existe ninguna línea dentro del archivo donde aparezca la palabra *keep,* añadirla.

Añadir la siguiente sentencia tras la línea donde encontramos *keep*:

```
path=/usr/src/qce-ga-0.40c
```

Esta sentencia añadirá el directorio a la lista de los lugares donde el SO buscará módulos para el *kernel*.

Sin dejar de ser *root*, escribir

```
depmod -a
```

Muchos de los ficheros en el directorio **`/usr/src/qce-ga-0.40c`** no pertenecen a *root*. Ignorar estas advertencias.

Es momento de reiniciar el sistema para que el driver de la cámara se cargue durante el arranque de *Linux*. Se puede observar si el *driver* se cargó correctamente ejecutando *lsmod* como usruario *root*. Debe aparecer el *driver* de *quickcam* en la lista de módulos cargados.

La webcam probablemente haya sido instalada como */dev/video0*.

ANEXO IV

Instalación de librerías de ACE/TAO

A continuación se presenta una guía rápida para la instalación y construcción de las librerías de ACE y TAO. Éstas se pueden descargar desde la web de Center for Distributed Object Computing de la Universidad de Washington **[18]**.

El primer paso es desempaquetar las librerías, se muestra a continuación:

```
tar xzvf ACE+TAO.tar.gz
```

Antes de comenzar la instalación se deben de establecer en el *boot* del sistema, las variables de entorno para ACE-TAO. Para ello debemos modificar (en cada cuenta de usuario) el *script* de inicio **.bashrc**. Por ejemplo, si se desempaquetó **ACE+TAO.tar.gz** dentro de **/home/user1**, **.bashrc** se encuentra dentro del directorio **user1**. Se deberá editar para añadir las siguientes líneas y así establecer las variables de entorno.

```
export ACE_ROOT=/home/user1/ACE_wrappers
export TAO_ROOT=$ACE_ROOT/TAO
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ACE_ROOT/ace
export PATH=$PATH:$TAO_ROOT/TAO_IDL
```

Comprobar la versión de la distribución obtenida. Se requiere TAO 1.2 o superior. Para poder comprobar ésto:

```
cat $TAO_ROOT/VERSION
```

Enlazar simbólicamente el fichero de configuración apropiado dentro de **$ACE_ROOT/ace** a **config.h**. Por ejemplo, si se está usando *linux*:

```
cd $ACE_ROOT/ace
ln -s config-linux.h config.h
```

Crear **$ACE_ROOT/include/makeinclude/platform_macros.GNU** con el siguiente contenido:

En el caso de *linux*:

```
exceptions=1
include $(ACE_ROOT)/include/makeinclude/platform_linux.GNU

TAO_ORBSVCS=Naming Property Notify AV

## For TAO 1.2.5, comment out the above line and uncomment the
##following one
#TAO_ORBSVCS=Naming Property RTEvent CosEvent RTSchedEvent Notify AV

## For Redhat 7.0 and 7.1, the following setting is required
## to work around a bug in the gcc 2.96 compiler OCFLAGS=-O
```

Para compilar los componentes importantes de ACE solo hay que ejecutar lo siguiente:

```
cd $ACE_ROOT/ace
make
cd $ACE_ROOT/apps/gperf
make
```

En algunas versiones nuevas de ACE (por ejemplo 5.2.5) puede ser necesario compilar también el componente ACEXML.

```
cd $ACE_ROOT/ACEXML
make
```

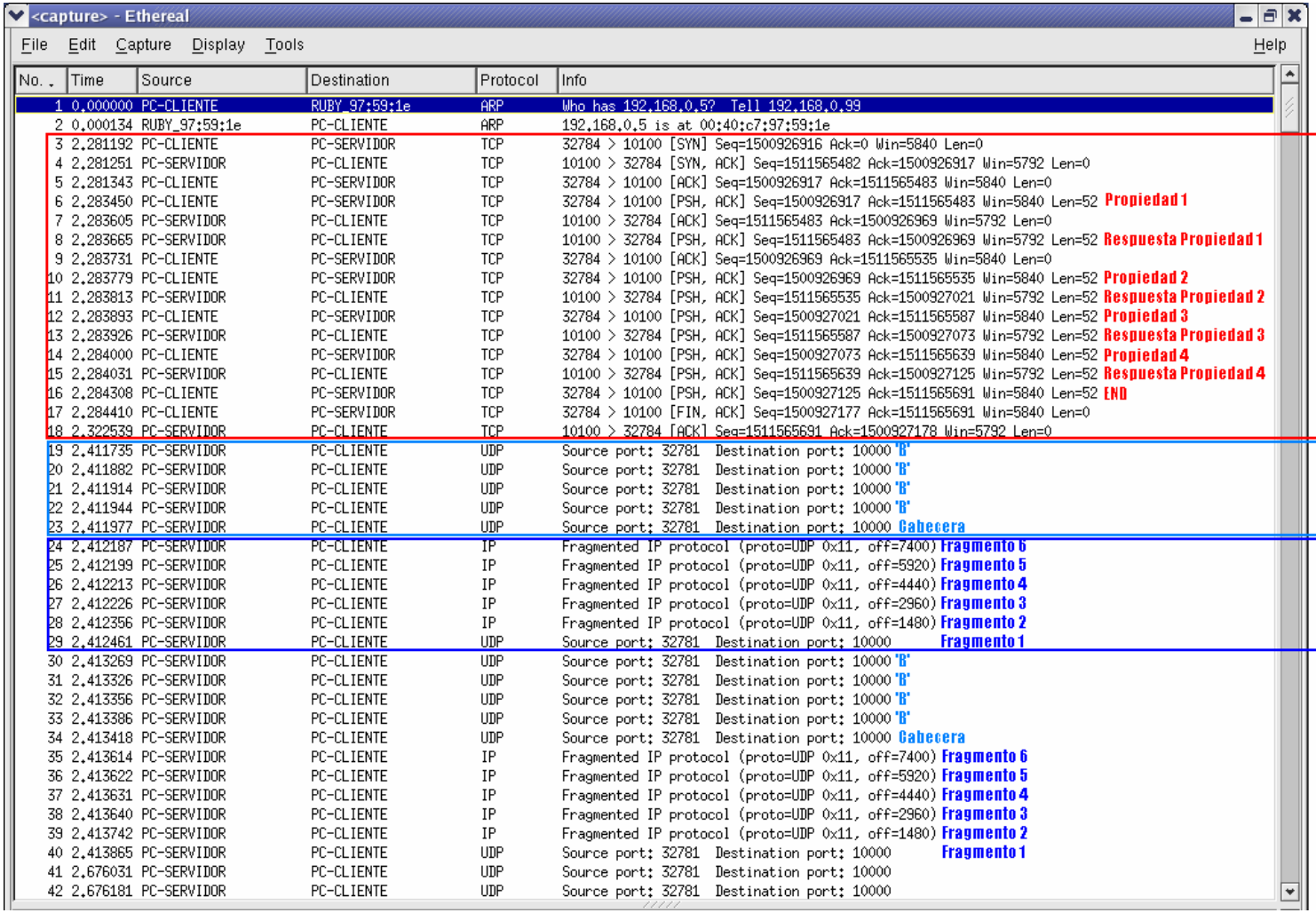
Para compilar los componentes importantes de TAO se debe ejecutar lo siguiente:

```
cd $TAO_ROOT/tao
make

cd $TAO_ROOT/TAO_IDL
make

cd $TAO_ROOT/orbsvcs
make

cd $TAO_ROOT/utils
make
```

Una recomendación es copiar estas líneas para compilar dentro de dos *scripts* y ejecutarlos. Es una forma cómoda y automática.

ANEXO V

Capturas de Red

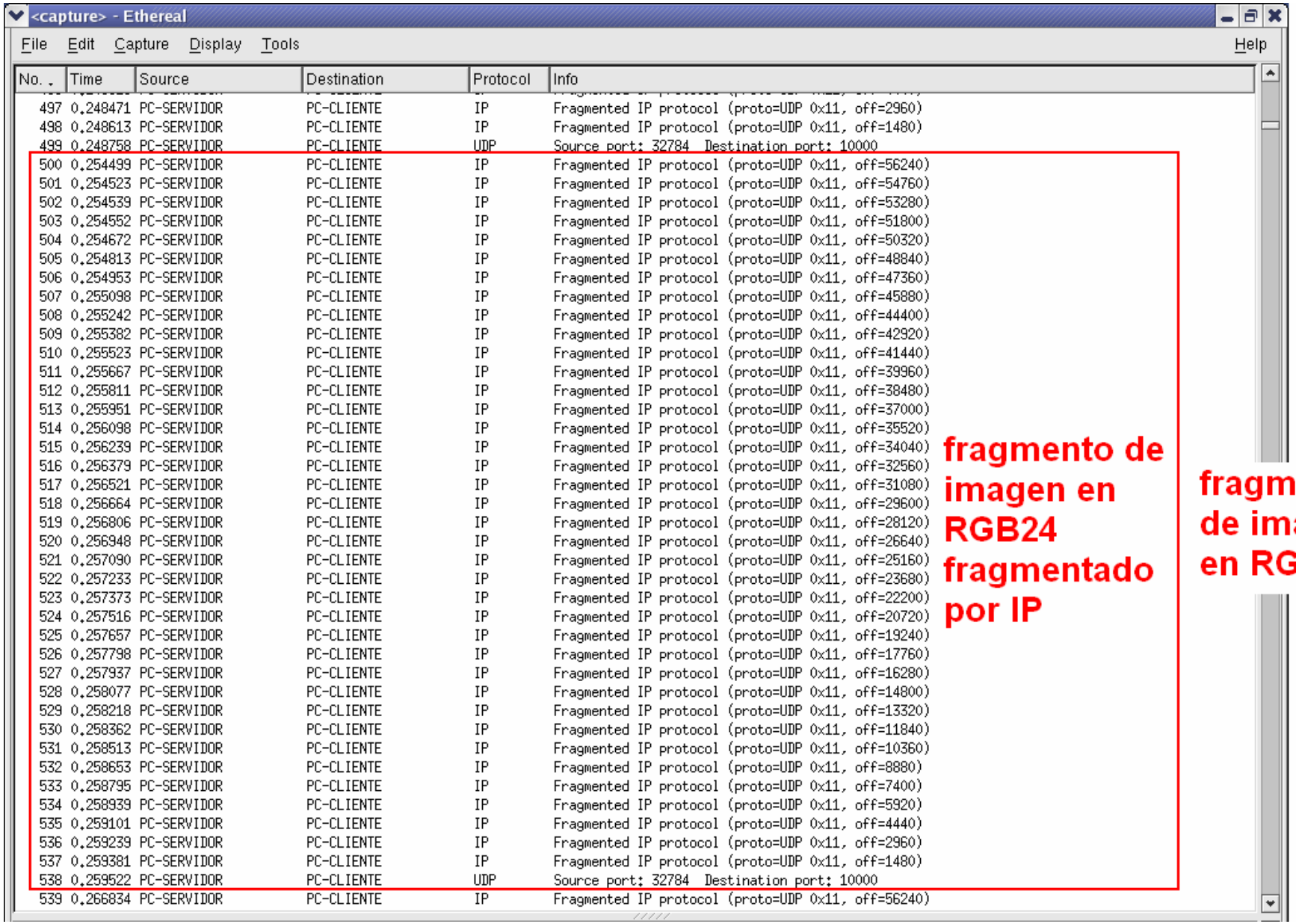


Negociación TCP y transmisión de video UDP

NEGOCIACIÓN TCP

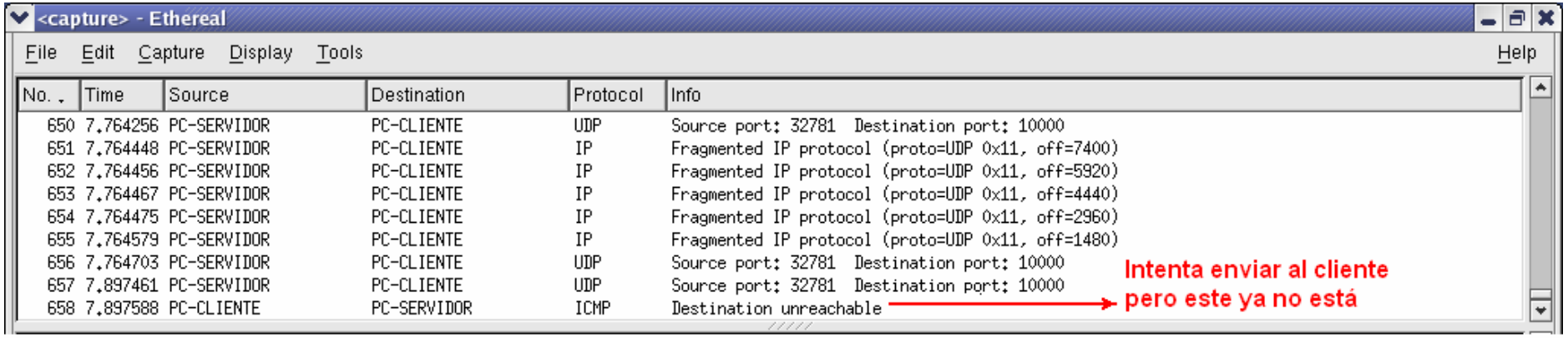
SINCRONIZACIÓN Y CABECERA

IMAGEN MJPEG FRAGMENTADA POR IP

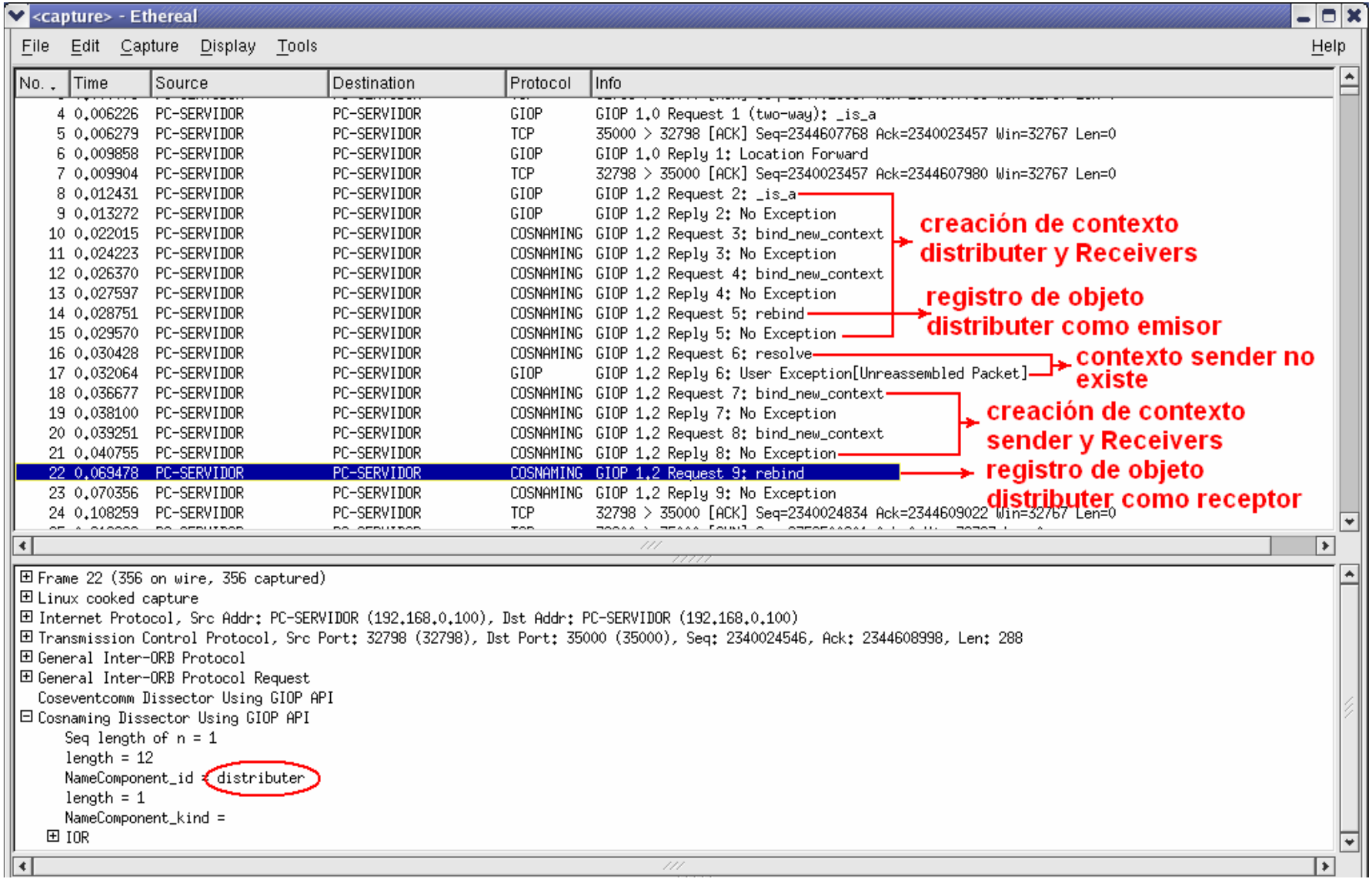


Transmisión de video usando compresión RGB 24

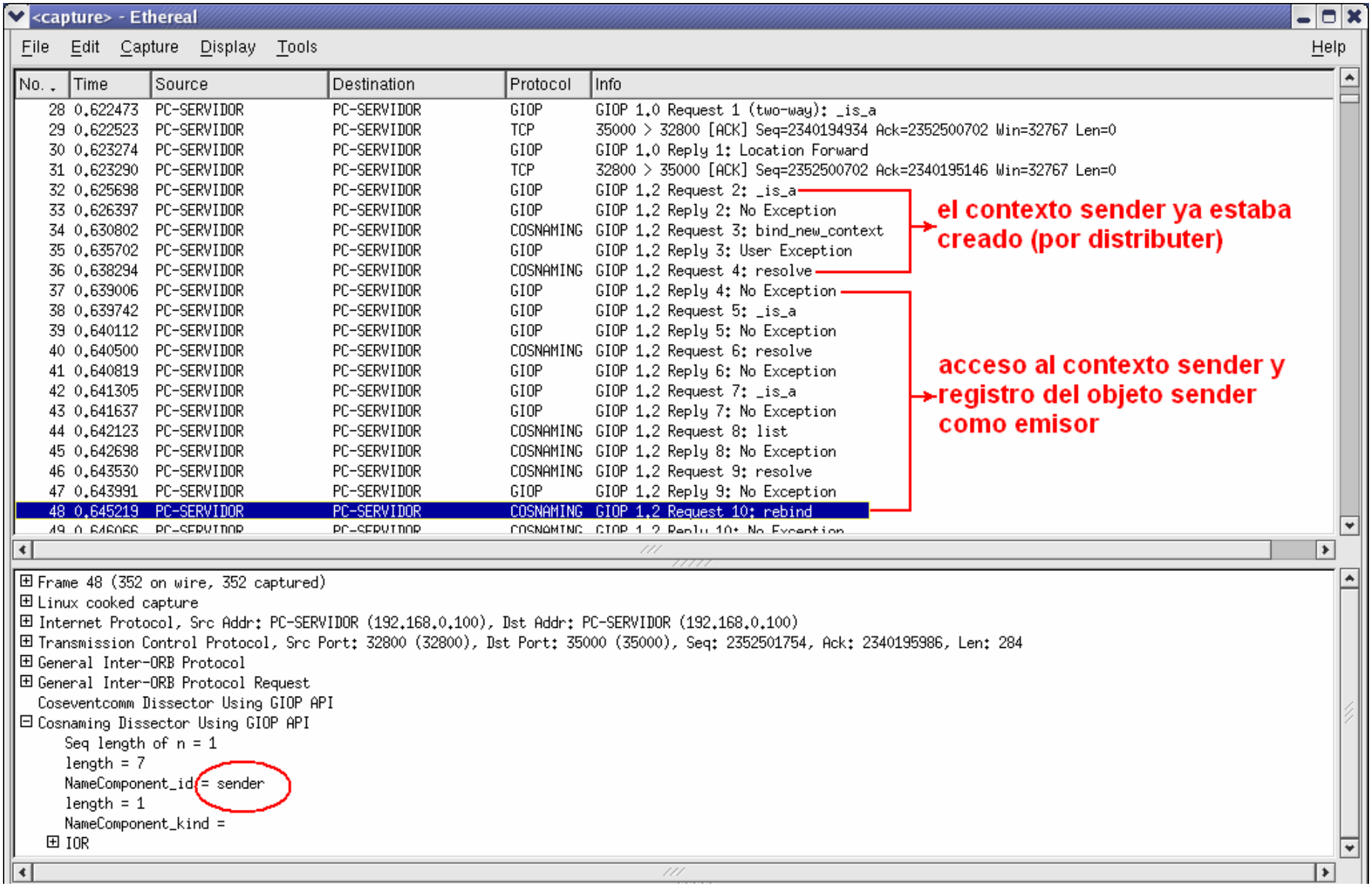
Desconexión de cliente usando UDP



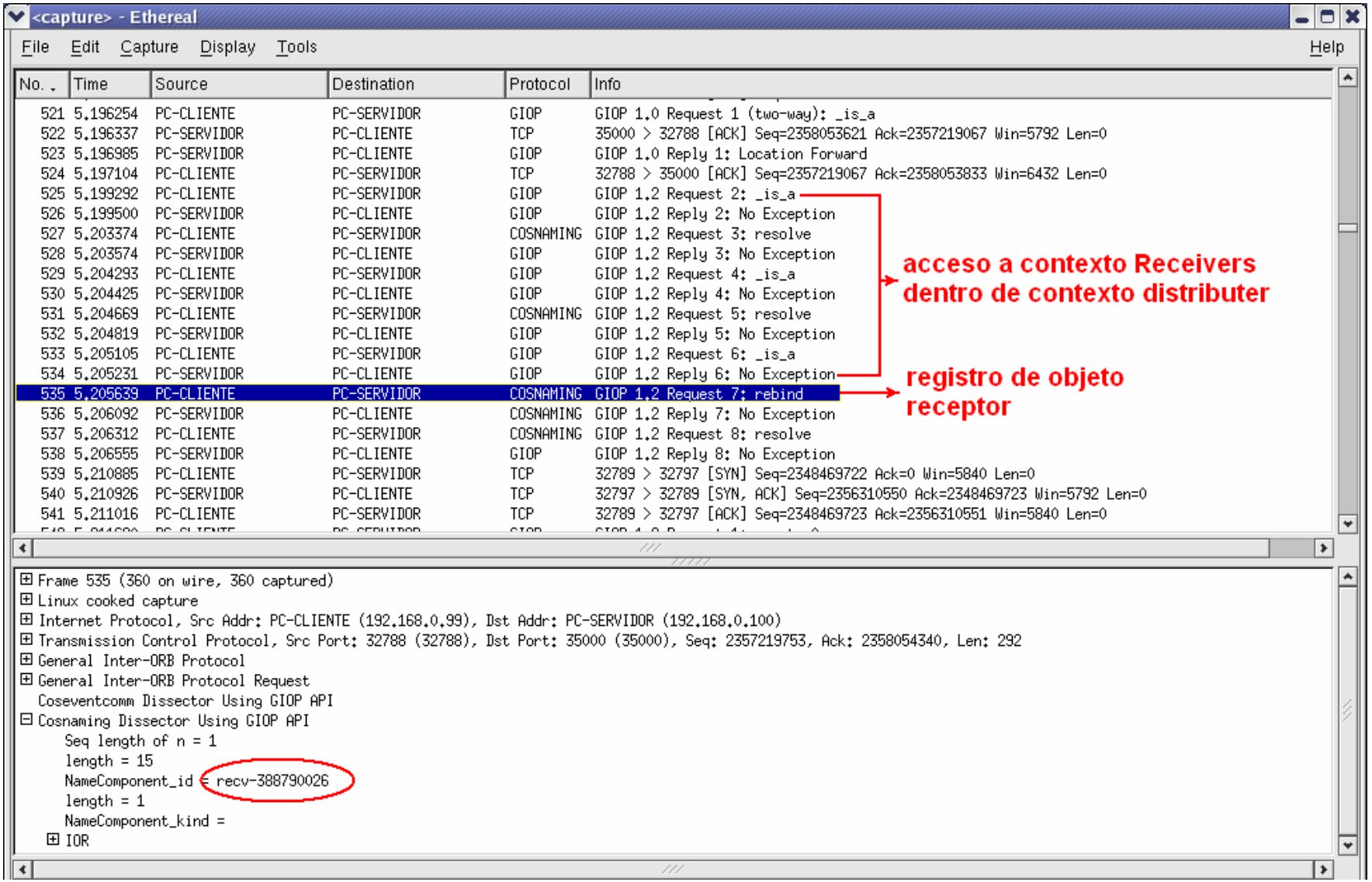
Registro del distribuidor en el Servicio de Nombrado



Registro del objeto `sender` en el Servicio de Nombrado



Registro de un cliente en el Servicio de Nombrado



Transmisión de una imagen mediante AVStreams

No.	Time	Source	Destination	Protocol	Info
606	5.255226	PC-SERVIDOR	PC-SERVIDOR	COSNAMING	GIOP 1.2 Reply 179: No Exception
607	5.282652	PC-SERVIDOR	PC-SERVIDOR	UDP	Source port: 32786 Destination port: 32785
608	5.284062	PC-SERVIDOR	PC-SERVIDOR	UDP	Source port: 32786 Destination port: 32785
609	5.285378	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=7400)
610	5.285402	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=5920)
611	5.285412	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=4440)
612	5.285428	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=2960)
613	5.285556	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=1480)
614	5.285677	PC-SERVIDOR	PC-CLIENTE	UDP	Source port: 32787 Destination port: 32773
615	5.286139	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=7400)
616	5.286166	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=5920)
617	5.286179	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=4440)
618	5.286310	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=2960)
619	5.286390	PC-SERVIDOR	PC-CLIENTE	IP	Fragmented IP protocol (proto=UDP 0x11, off=1480)
620	5.286517	PC-SERVIDOR	PC-CLIENTE	UDP	Source port: 32787 Destination port: 32773

Imagen

primer fragmento

Obtención de lista de clientes actuales del distribuidor

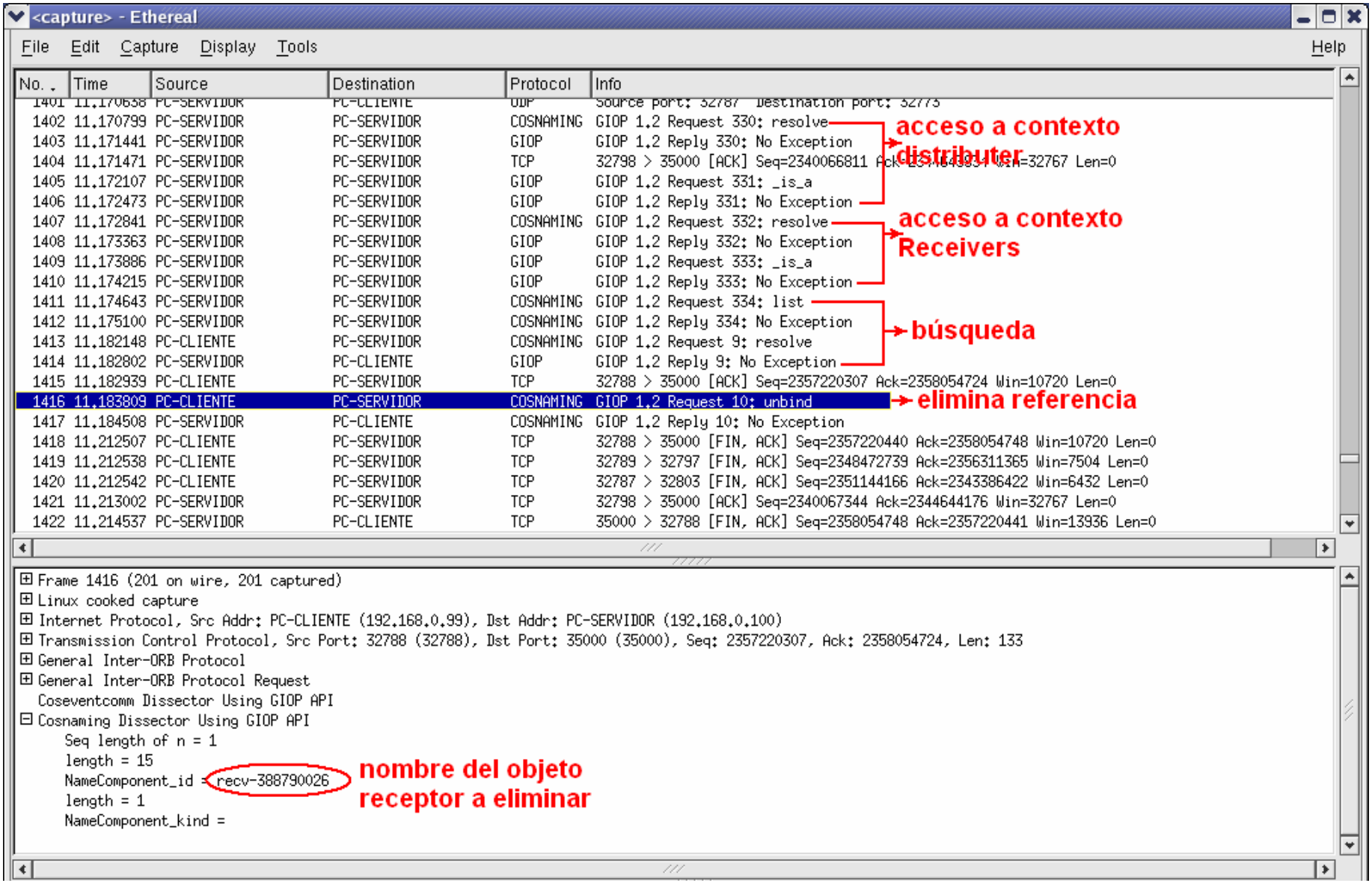
No.	Time	Source	Destination	Protocol	Info
506	5.018953	PC-SERVIDOR	PC-SERVIDOR	COSNAMING	GIOP 1.2 Request 160: resolve
507	5.019581	PC-SERVIDOR	PC-SERVIDOR	GIOP	GIOP 1.2 Reply 160: No Exception
508	5.019607	PC-SERVIDOR	PC-SERVIDOR	TCP	32798 > 35000 [ACK] Seq=2340044575 Ack=2344624838 Win=32767 Len=0
509	5.020228	PC-SERVIDOR	PC-SERVIDOR	GIOP	GIOP 1.2 Request 161: _is_a
510	5.020582	PC-SERVIDOR	PC-SERVIDOR	GIOP	GIOP 1.2 Reply 161: No Exception
511	5.020932	PC-SERVIDOR	PC-SERVIDOR	COSNAMING	GIOP 1.2 Request 162: resolve
512	5.021283	PC-SERVIDOR	PC-SERVIDOR	GIOP	GIOP 1.2 Reply 162: No Exception
513	5.021735	PC-SERVIDOR	PC-SERVIDOR	GIOP	GIOP 1.2 Request 163: _is_a
514	5.022100	PC-SERVIDOR	PC-SERVIDOR	GIOP	GIOP 1.2 Reply 163: No Exception
515	5.022441	PC-SERVIDOR	PC-SERVIDOR	COSNAMING	GIOP 1.2 Request 164: list
516	5.022830	PC-SERVIDOR	PC-SERVIDOR	COSNAMING	GIOP 1.2 Reply 164: No Exception

acceso a contexto distributer

acceso a contexto Receivers

listado de objetos registrados

Cierre de la conexión de un cliente



ANEXO VI

Condiciones técnicas

Los requisitos de hardware y software (entre paréntesis los recomendados) que garantizan un funcionamiento correcto son:

HARDWARE

- Pentium 3 1.0 Ghz (Pentium 4 2.0 Ghz)
- 256 *Mbytes* Ram (512 *Mbytes*)
- Disco duro 10 Gb (20 Gb)
- Webcam compatible con SO linux (capturadora tv *miro PCTV* y cámara)
- Tarjeta de red *Fast Ethernet* 100 Mbps (Tarjeta *Gigabit Ethernet* 1000 *Mbps*)
- Hub/Switch Fast Ethernet 100 Mbps (Switch *GigaBit Ethernet* 1000 *Mbps*)

SOFTWARE

- Redhat 8.0 (o superior)
- ACE+TAO 5.2.6
- JRE 1.4

REFERENCIAS

[1] *XAnim*
http://smurfland.cit.buffalo.edu/xanim/home.html

[2] Directshow
http://msdn.microsoft.com/library/en-us/directx9_c/directx/htm/directshow.asp

[3] The Java Media Framework (JMF)
http://java.sun.com/products/java-media/jmf/

[4] Web oficial de xawtv
http://www.bytesex.org/xawtv

[5] Red Hat Linux
http://www.redhat.com

[6] C Programming Language , The. Kerninghan, Ritchie

[7] "Object-Oriented Application Frameworks", Communications of the ACM, Vol. 40, No 10, October 1997. M. Fayad, D.C. Schmidt

[8] "Designs Patterns: Elements of Reusable Object Oriented Software", Addison Wesley, Reading Mass, 1995. E. Gamma, R. Helm, R. Johnson, J. Vlissides

[9] Manual GNU Make
http://www.gnu.org/manual/make-3.80/make.html

[10] Linux Complete Command Reference, Compiled By J. Purcell

[11] Computer Networks 4ª Edición, 2003. Tanenbaum, Andrew S

[12] Transmission Control Protocol
http://www.ietf.org/rfc/rfc0793.txt

[13] User Datagram Protocol
http://www.ietf.org/rfc/rfc0768.txt

[14] Web oficial del OMG
http://www.omg.org/

[15] Web de ACE TAO en Universidad de Washington
http://www.cs.wustl.edu/~schmidt/TAO.html

[16] OMG, "Control and Management of A/V Streams Specification". OMG Document telecom/97-05-07 ed, October 97

[17] Obtención de librerías ACE-TAO
http://deuce.doc.wustl.edu/Download.html

[18] El Lenguaje de Programación C++ Edición especial. Bjarne Stroupstrup

[19] Java 2 Platform, Standard Edition
http://java.sun.com/j2se

[20] Advanced Streaming Format (asf)
http://www.microsoft.com/windows/windowsmedia/format/asfspec.aspx