UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Extensión de *xawtv* para la transmisión de flujos de video RTP

AUTOR

Francisco Javier Parrado García

DIRECTORES

Juan Ángel Pastor Franco

Felipe García Sánchez

Septiembre / 2003

Autor	Francisco Javier Parrado García
E-mail del Autor	fjparrado@teleline.es
Director(es)	Juan Ángel Pastor Franco Felipe García Sánchez
E-mail del Director	juanangel.pastor@upct.es
Codirector(es)	
Título del PFC	Extensión de *xawtv* para la transmisión de flujos de video RTP
Descriptores	

Resumen

Partiendo de la suite *xawtv* de 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 usando los protocolos de transmisión UDP y RTP. Además, pueden negociarse parámetros sobre la calidad de servicio, y todo ello con una filosofía de diseño basada en *plugins*.

Titulación	Ingeniería Técnica de Telecomuniación, especialidad Telemática
Intensificación	
Departamento	Departamento de Tecnologías de la Información y las 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 [2]. Un ORB que cumple estas características, es el que se encuentra en las librerías de ACE y se llama The ACE ORB (TAO) [3]. 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* [4], 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) [5] 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 pretende transmitir video a través de una LAN (Local Area Network). Para ello se desarrollará una aplicación partiendo de una suite de libre distribución (y *freesource*) llamada *xawtv*, y basándose en el modelo Cliente/Servidor

Para el extremo del cliente se usará el reproductor de video *pia*. En el extremo servidor se partirá de la aplicación *xawtv*. Como el código fuente de la *suite* está escrito en C y preparado para ser compilado bajo plataformas Unix-Linux, se trabajará sobre un SO de este tipo (Suse, Redhat, Mandrake…). En concreto se ha trabajado bajo *Redhat linux* 8.0.

El servidor de la aplicación debe aceptar peticiones de conexión procedentes de varios clientes y poder negociar ciertos parametros sobre el video y la transmisión. Por esto se considera de importancia llevar cabo un diseño que permita crear, añadir y negociar nuevos parámetros sin que los cambios afecten a la estructura global del programa. Entre los parámetros de transmisión que se negocian, se encuentra el tipo de protocolo de transmisión de video, que podrá ser UDP (con alguna funcionalidad extra) o RTP.

Para la implementación de RTP también se usarán unas librerías de libre distribución y *freesource* implementadas en lenguaje C.

2 La aplicación *xawtv*

2.1 Introducción

xawtv [6], 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* [7].

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

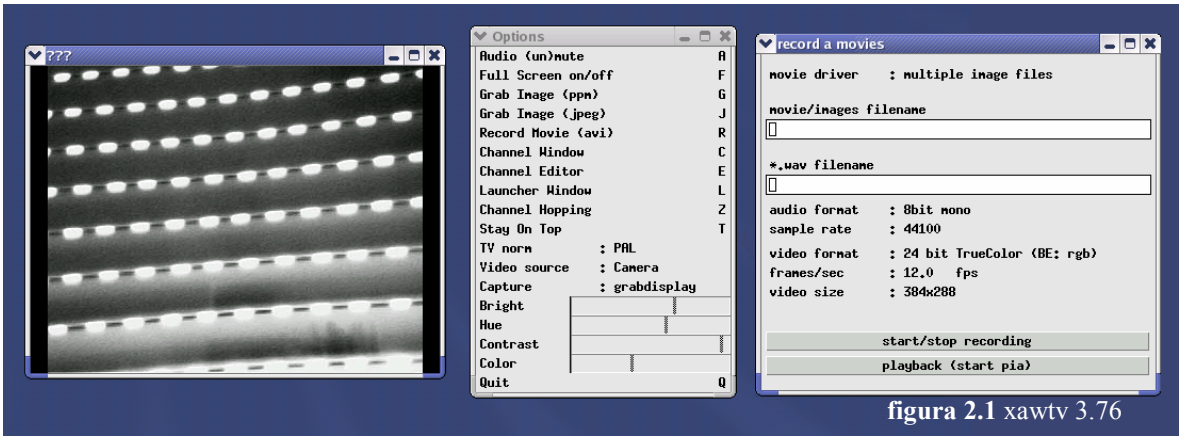


figura 2.1 xawtv 3.76

2.2 Características

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

- Está programado en lenguaje C [8] y es *freesource*, lo que permite su ampliación o modificación por otros programadores
- Es equivalente a un *framework* [9] en *POO*[3] a pesar de estar escrito en C
- Gran capacidad de ampliación mediante *plugins* [10]
- 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

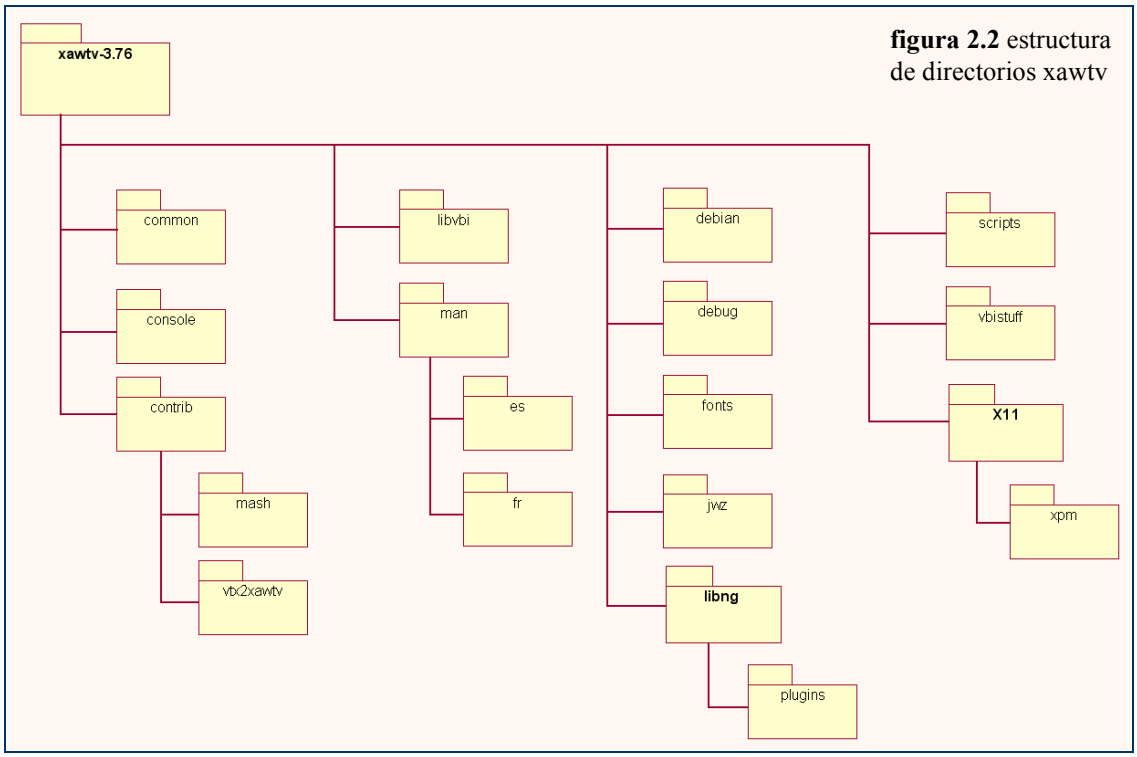


figura 2.2 estructura de directorios xawtv

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

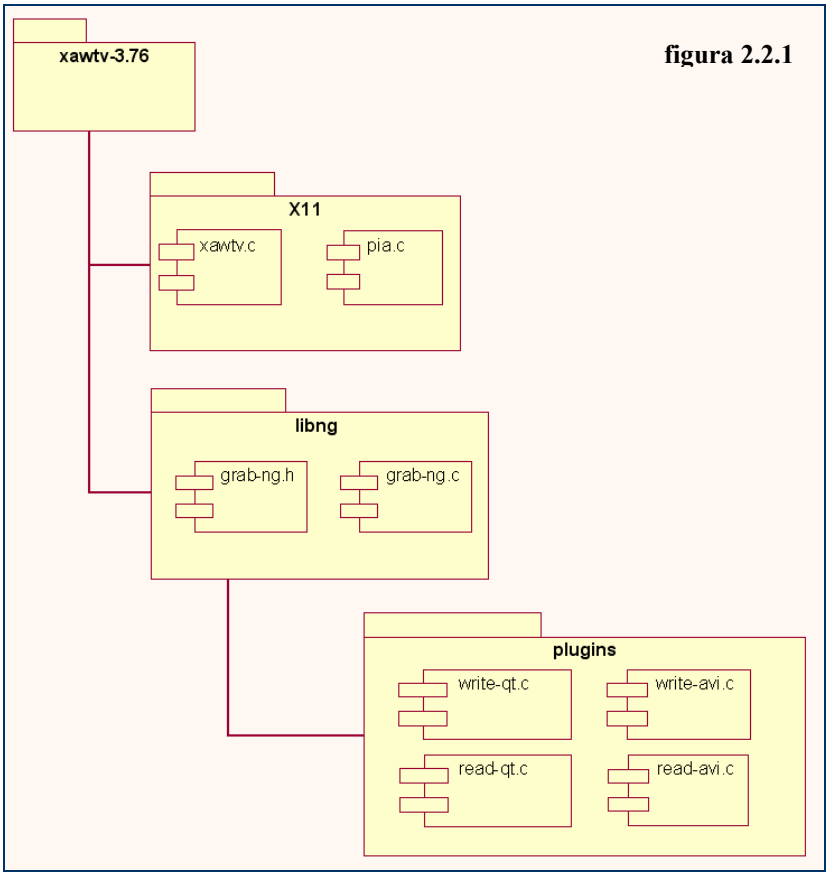


figura 2.2.1

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

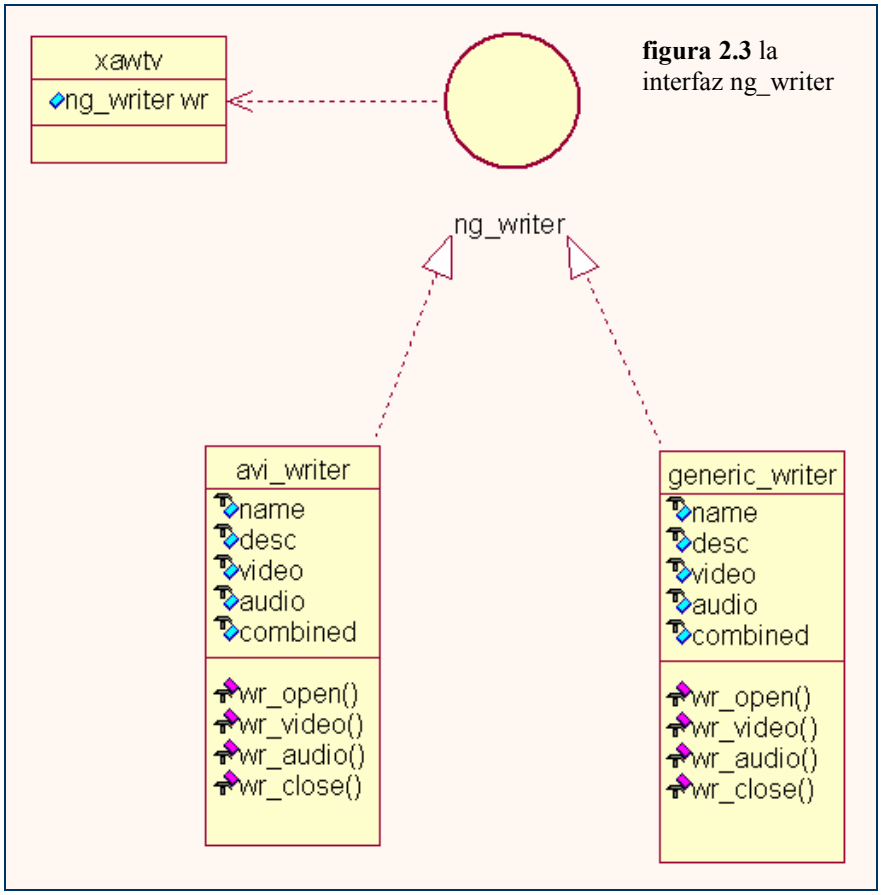


figura 2.3 la interfaz ng_writer

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

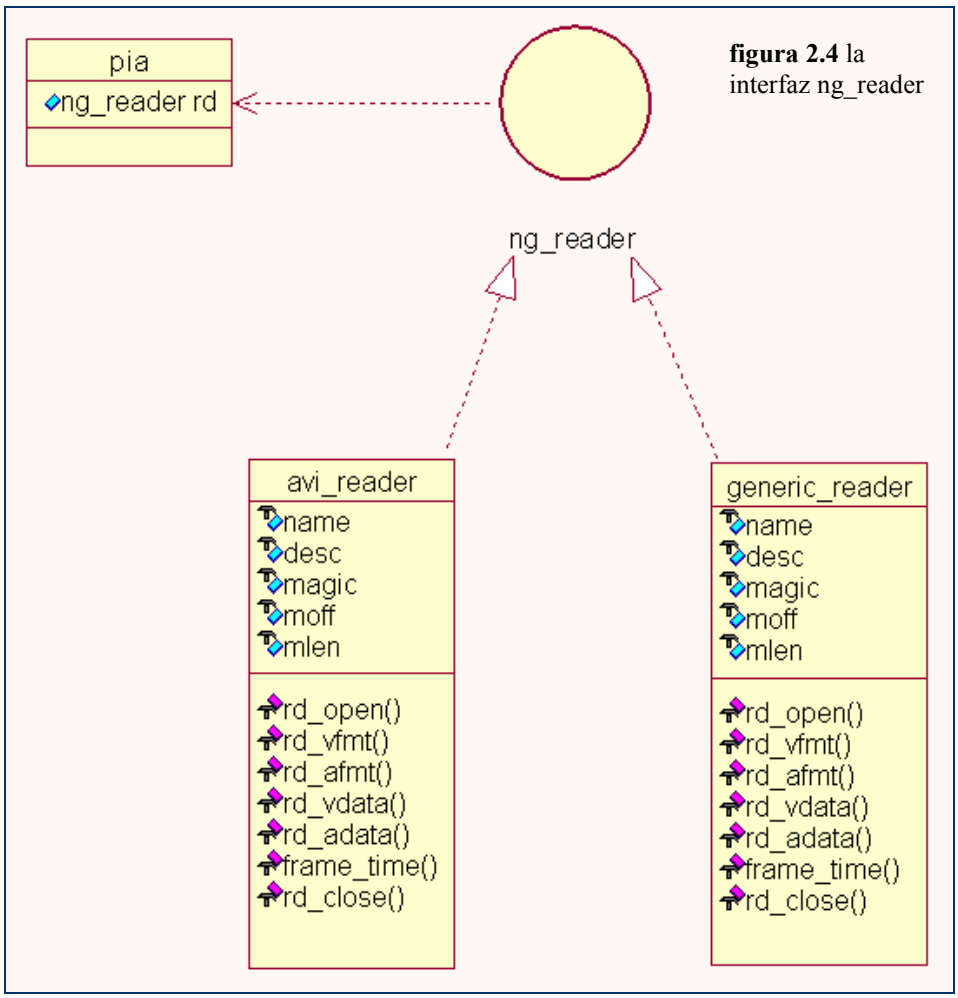


figura 2.4 la interfaz ng_reader

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.

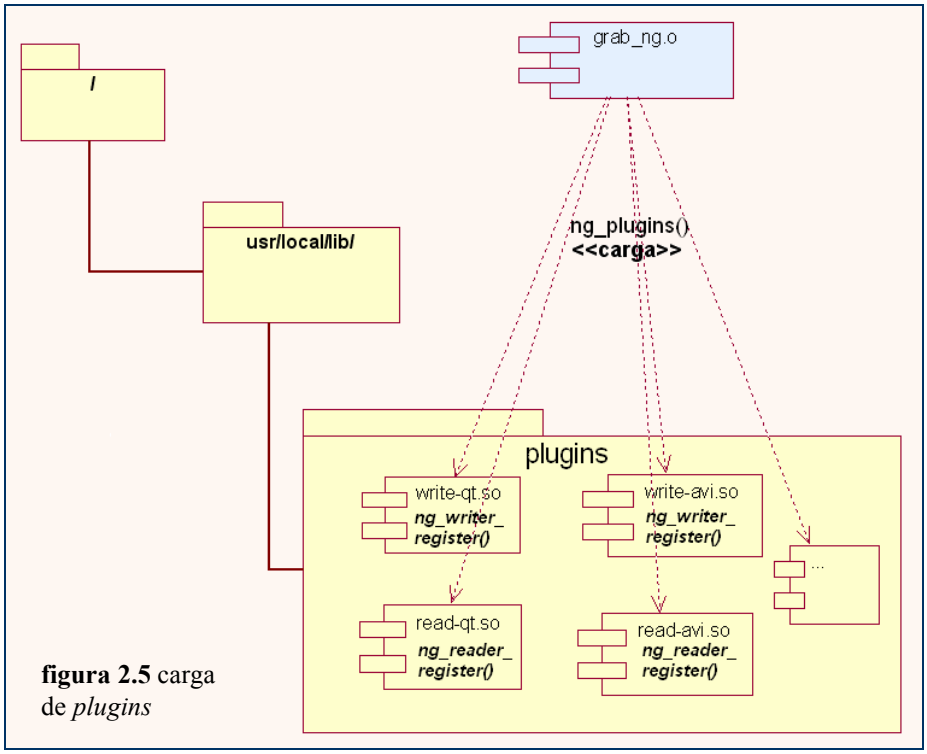


figura 2.5 carga de *plugins*

[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.

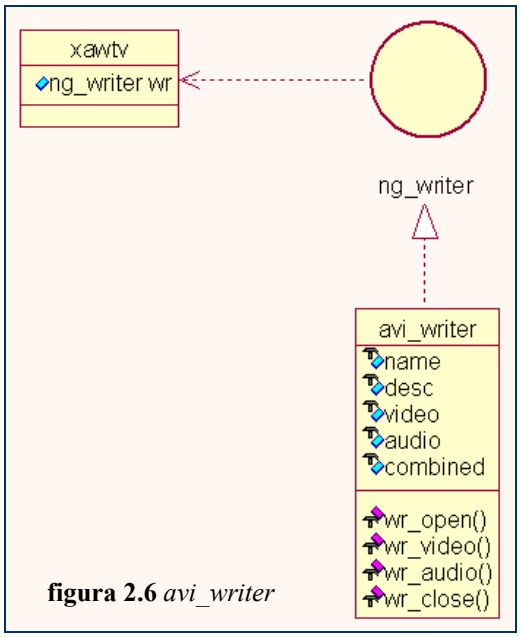


figura 2.6 *avi_writer*

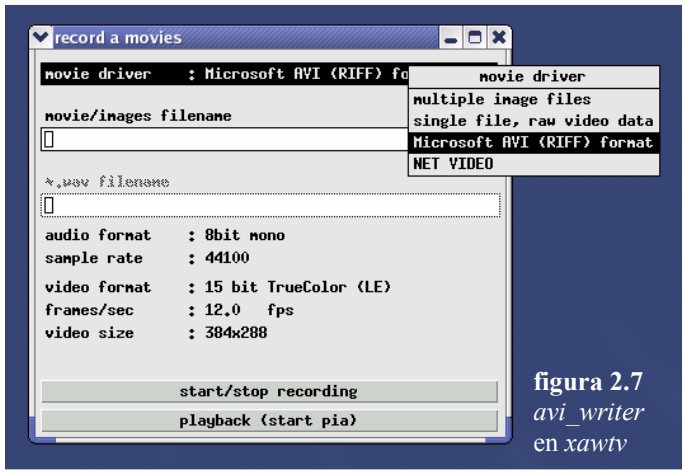


figura 2.7 *avi_writer* en *xawtv*

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.

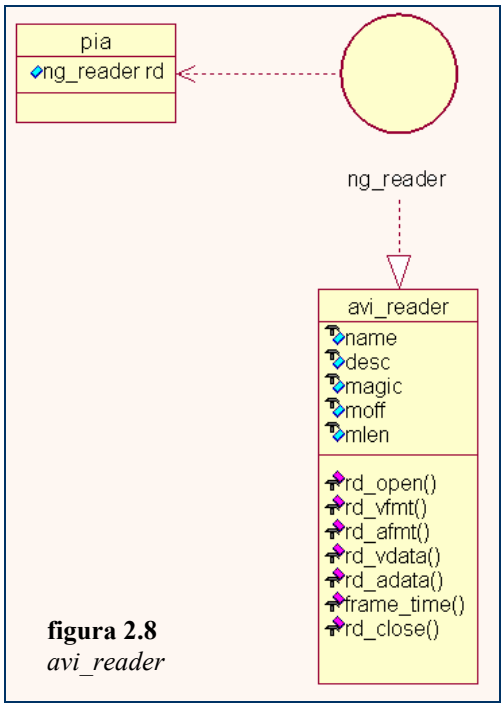


figura 2.8
avi_reader

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.

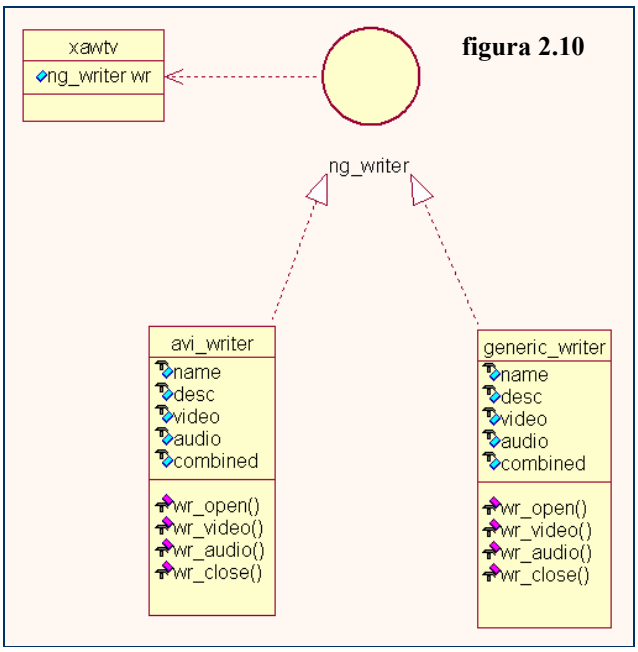


figura 2.10

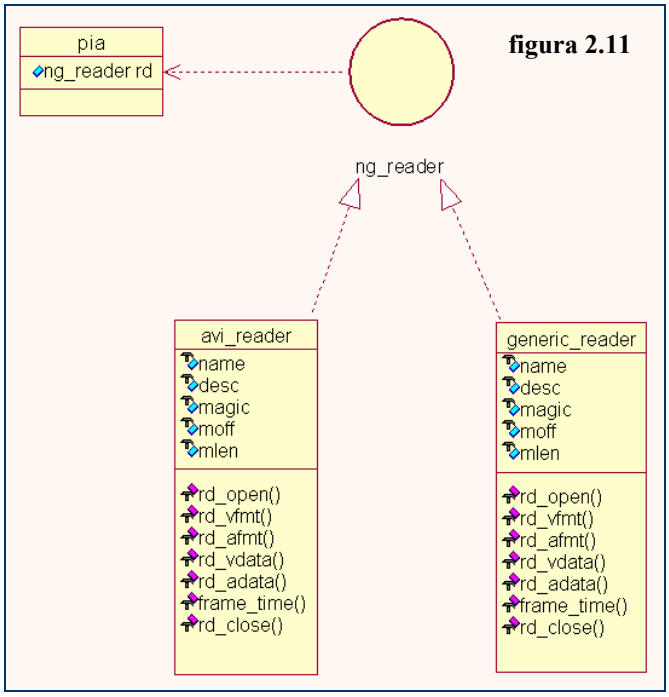


figura 2.11

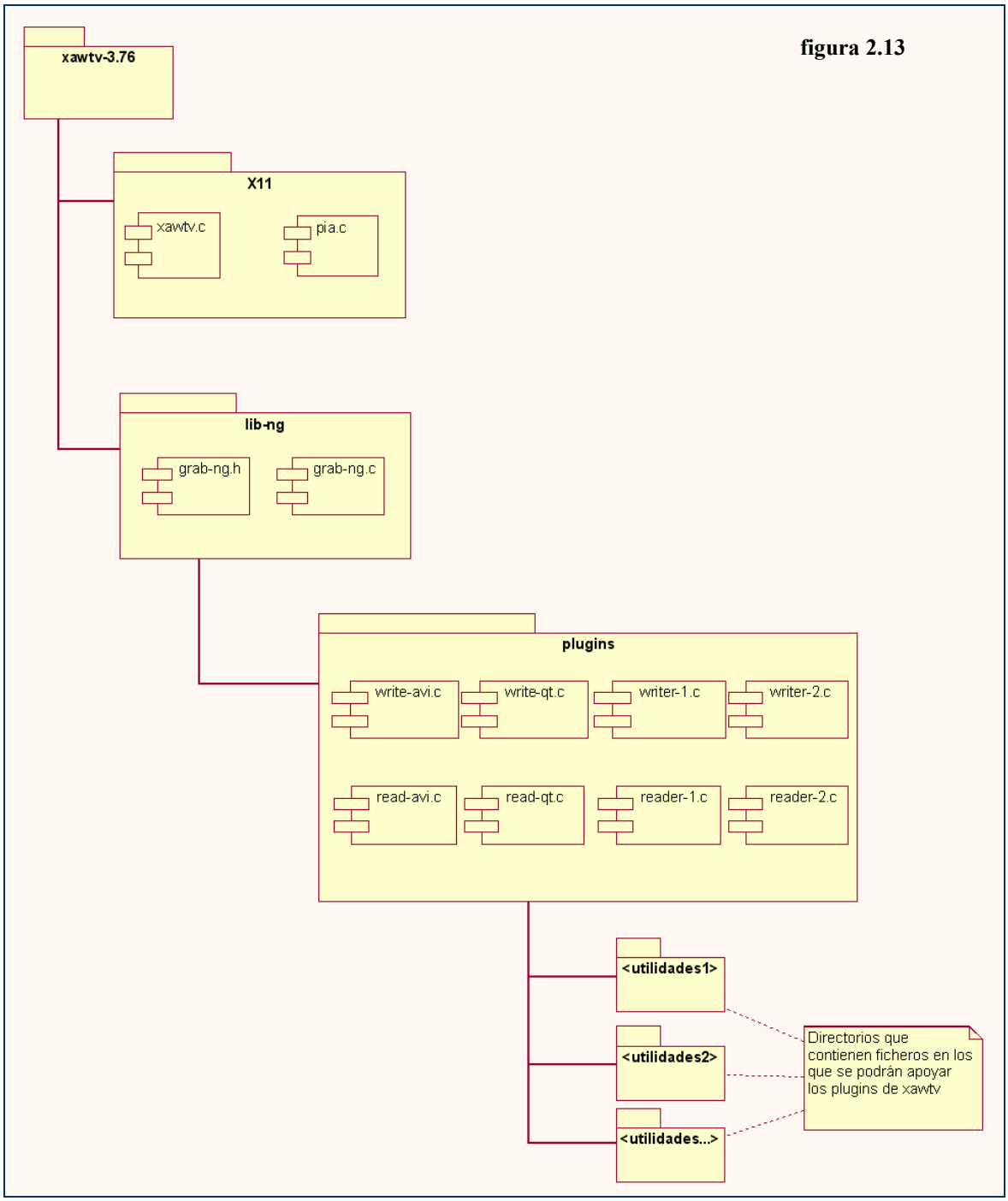

xawtv-3.76
X11
xawtv.c
pia.c
lib-ng
grab-ng.h
grab-ng.c
plugins
write-avi.c
write-qt.c
writer-1.c
writer-2.c
read-avi.c
read-qt.c
reader-1.c
reader-2.c
<utilidades1>
<utilidades2>
<utilidades...>
Directorios que contienen ficheros en los que se podrán apoyar los plugins de xawtv

figura 2.13

2.6 Compilación

La compilación [11] 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 *configure* [12]. 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.

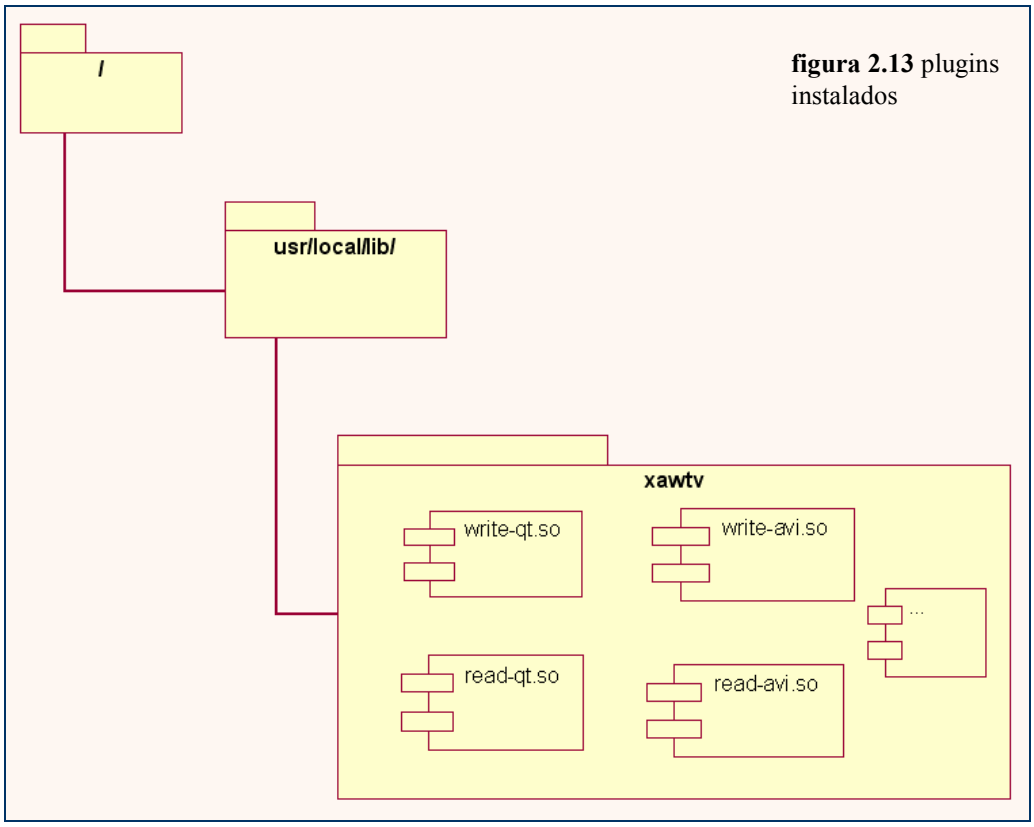


figura 2.13 plugins instalados

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.

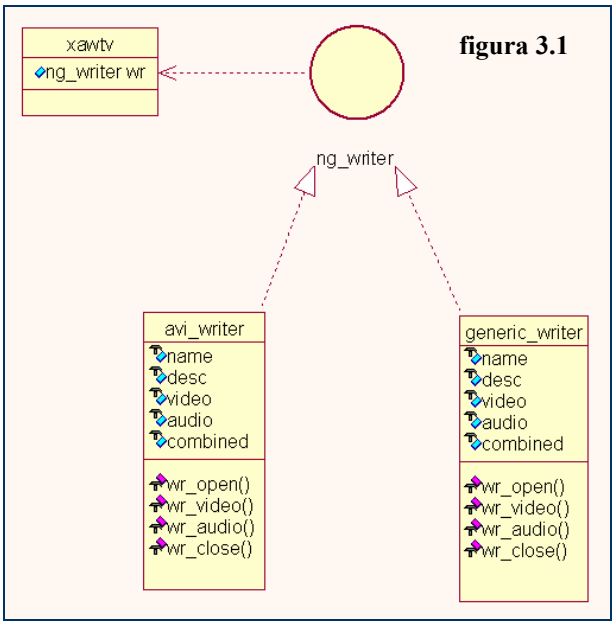


figura 3.1

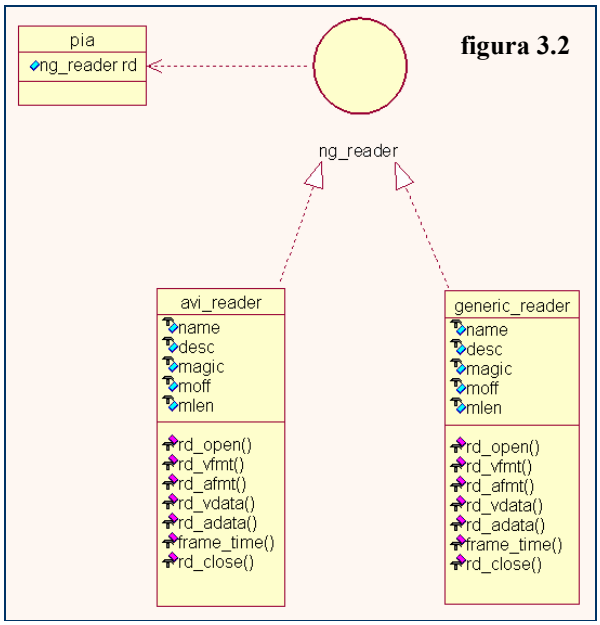


figura 3.2

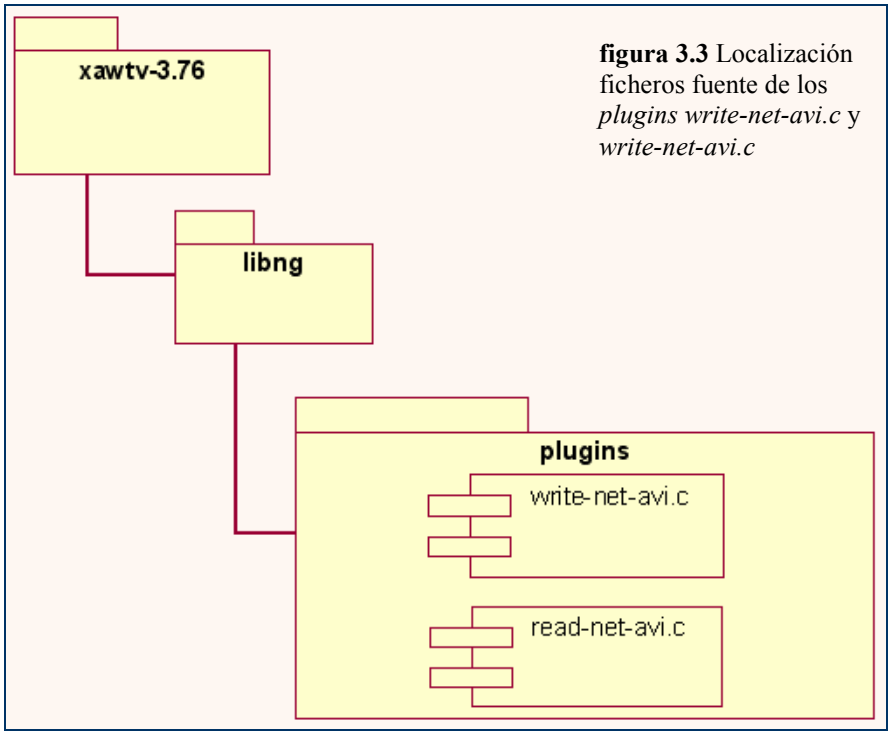


figura 3.3 Localización ficheros fuente de los *plugins write-net-avi.c* y *write-net-avi.c*

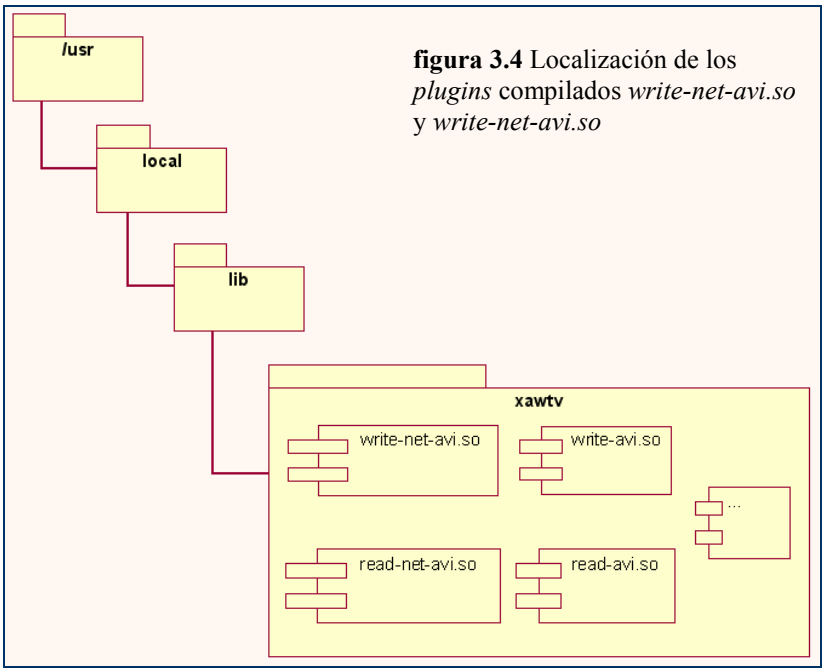


figura 3.4 Localización de los *plugins* compilados *write-net-avi.so* y *write-net-avi.so*

4 Protocolo de inicialización y negociación TCP

4.1 Introducción a TCP

TCP e IP [13] 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 [14], 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

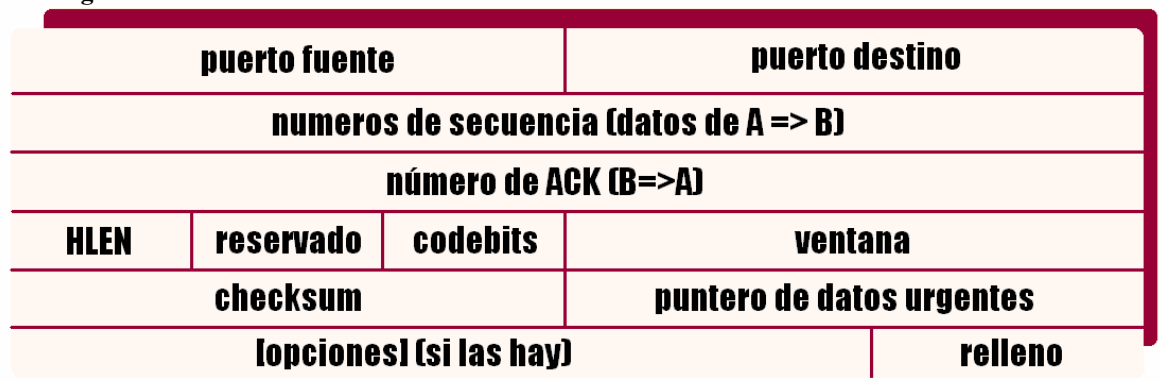
Pueden haber pérdidas de datagramas IP por diversos motivos: fallo en el hardware de red, *switchs* o *routers* congestionados que descartan datagramas… Existe así una necesidad de proporcionar a aplicaciones fiabilidad sobre el protocolo IP (no confiable).

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.

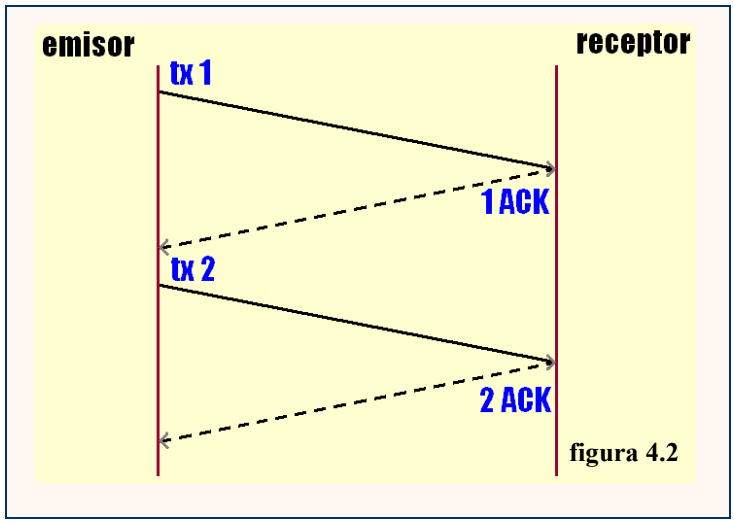


figura 4.2

- 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

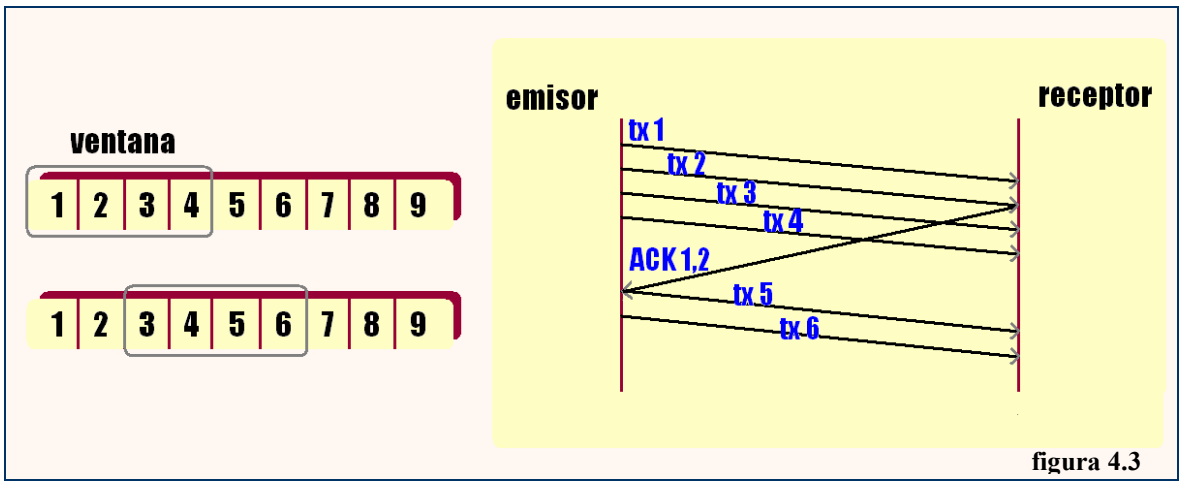


figura 4.3

- 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

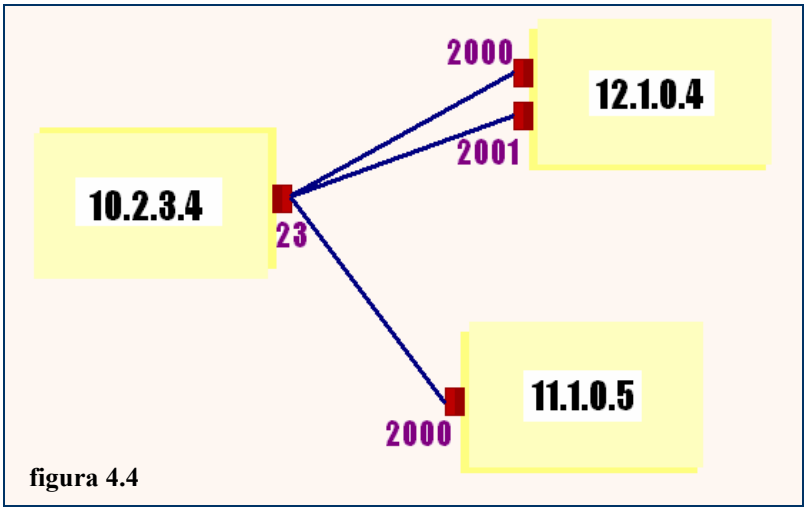


figura 4.4

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

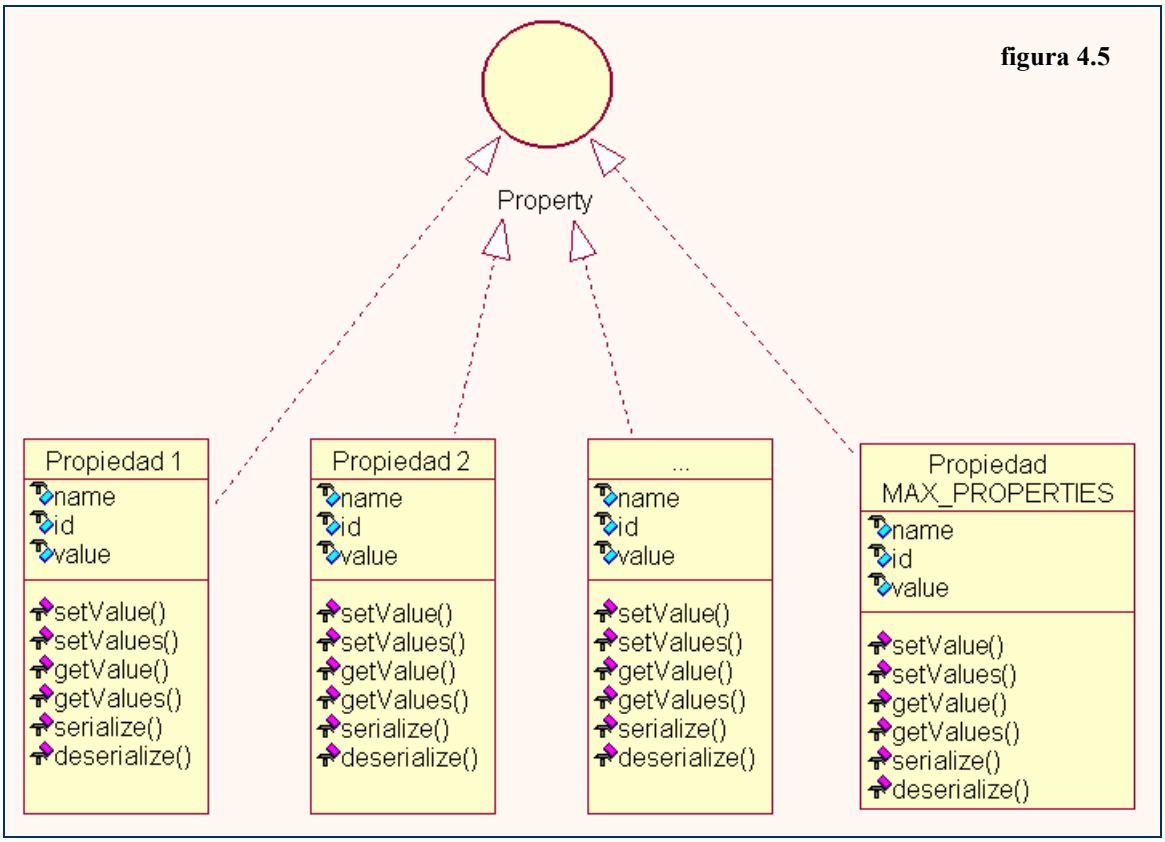


figura 4.5

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.

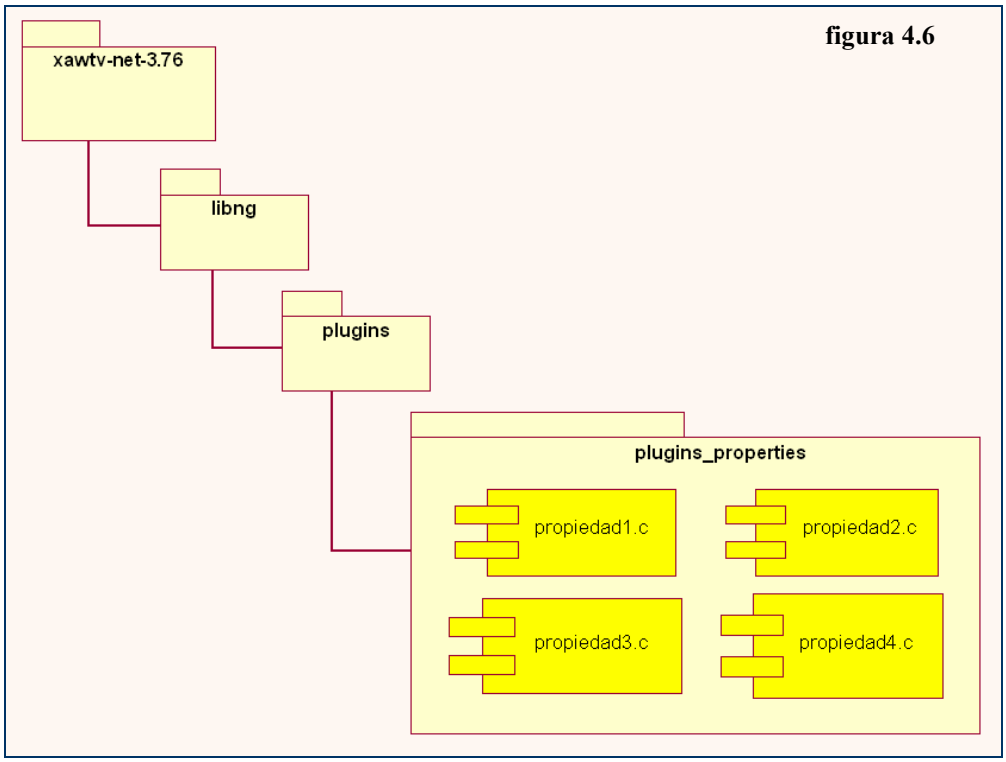
4.2.2.3 Integración de las propiedades en *xawtv*

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*.

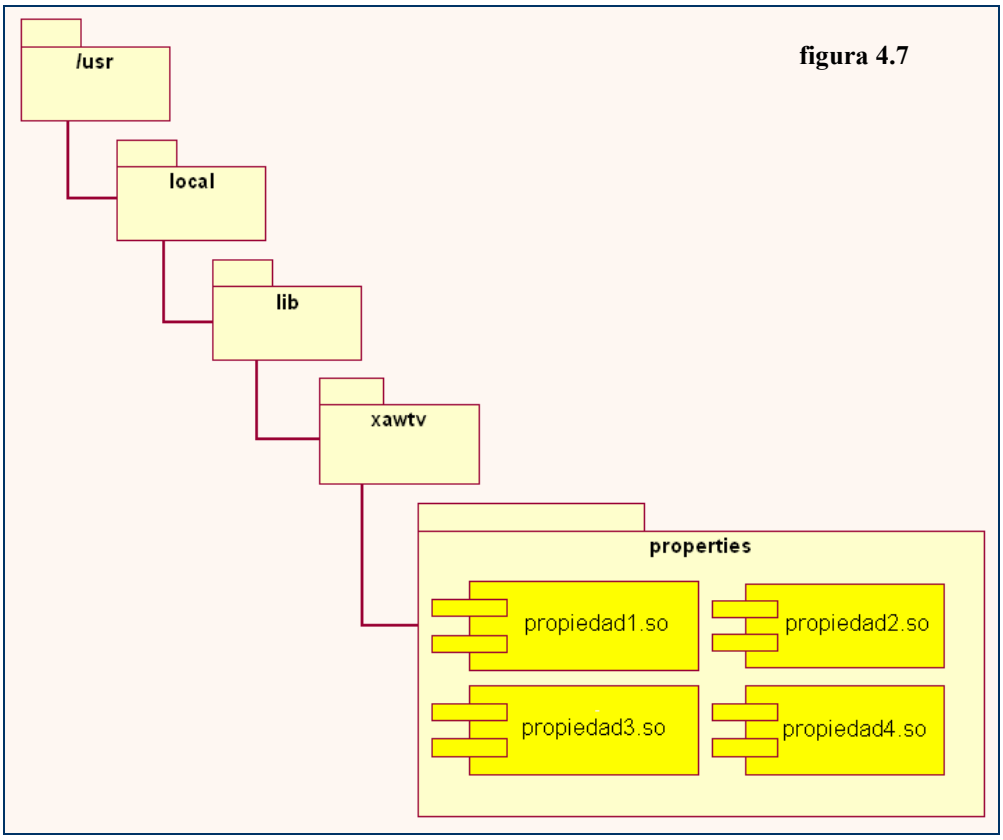
4.2.2.3.1 Ficheros de propiedades

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*.

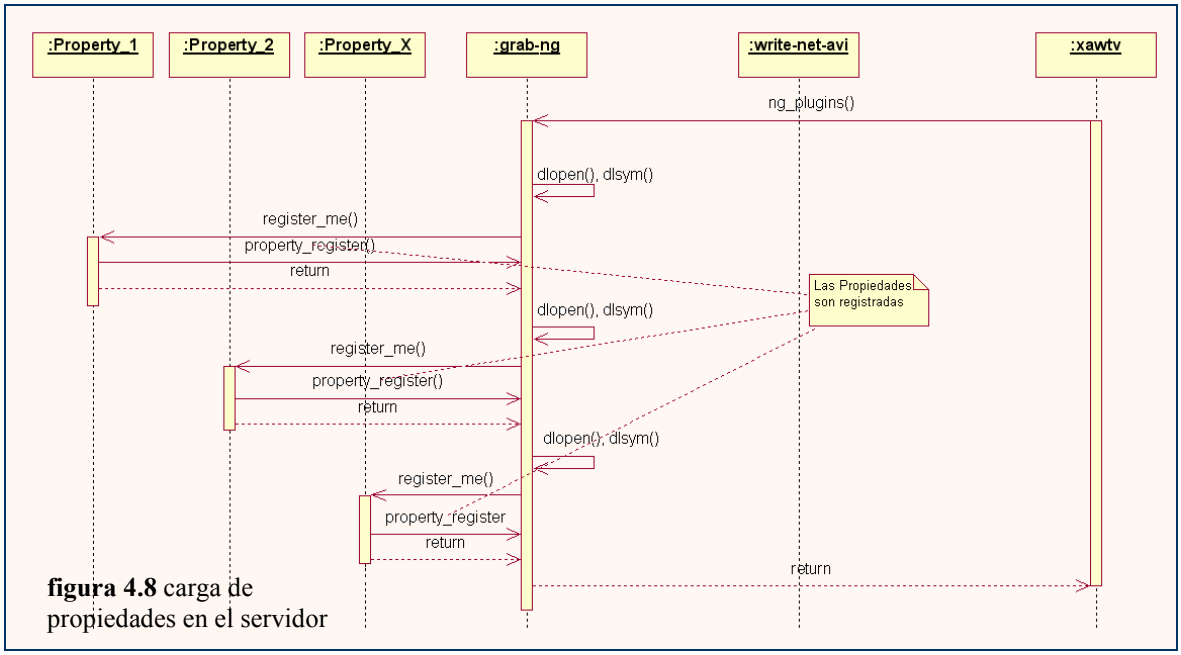


figura 4.8 carga de propiedades en el servidor

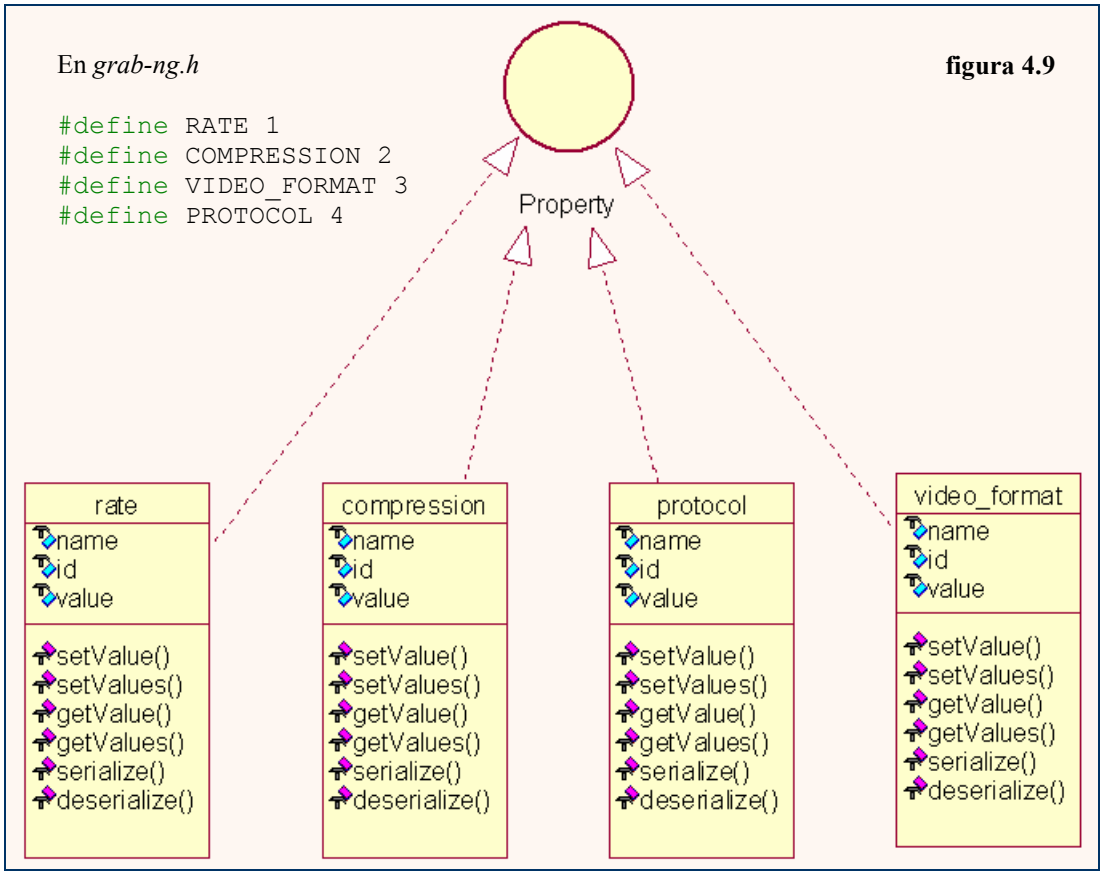
4.2.2.4 Propiedades implementadas

Se han implementado cuatro propiedades.

En *grab-ng.h*

figura 4.9

```
#define RATE 1
#define COMPRESSION 2
#define VIDEO_FORMAT 3
#define PROTOCOL 4
```



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.

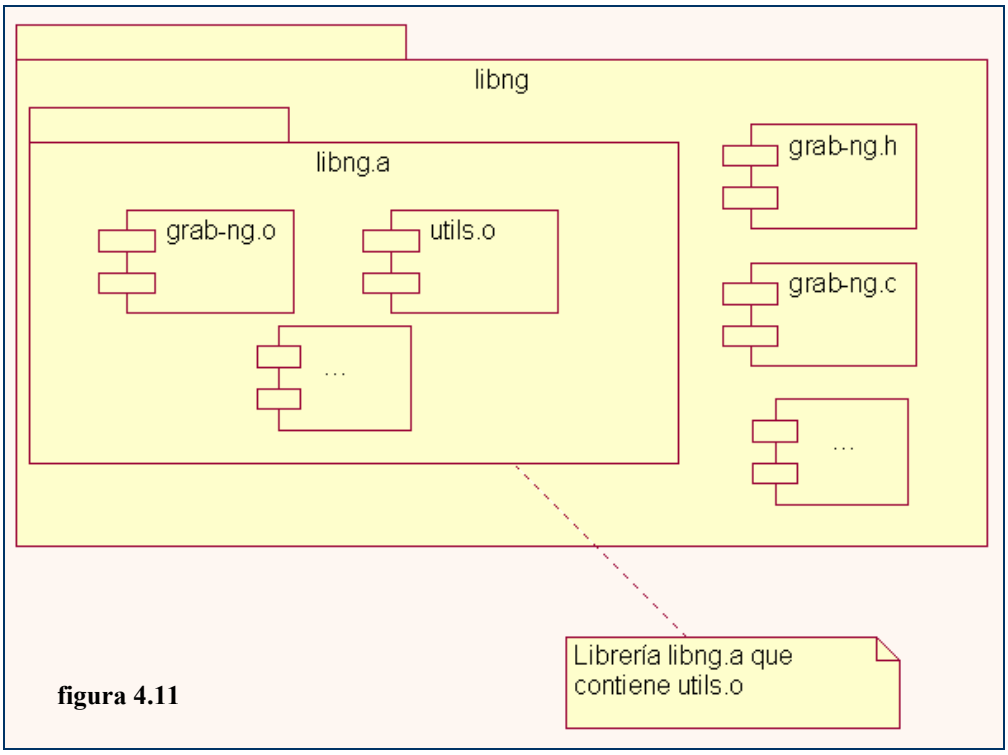


figura 4.11

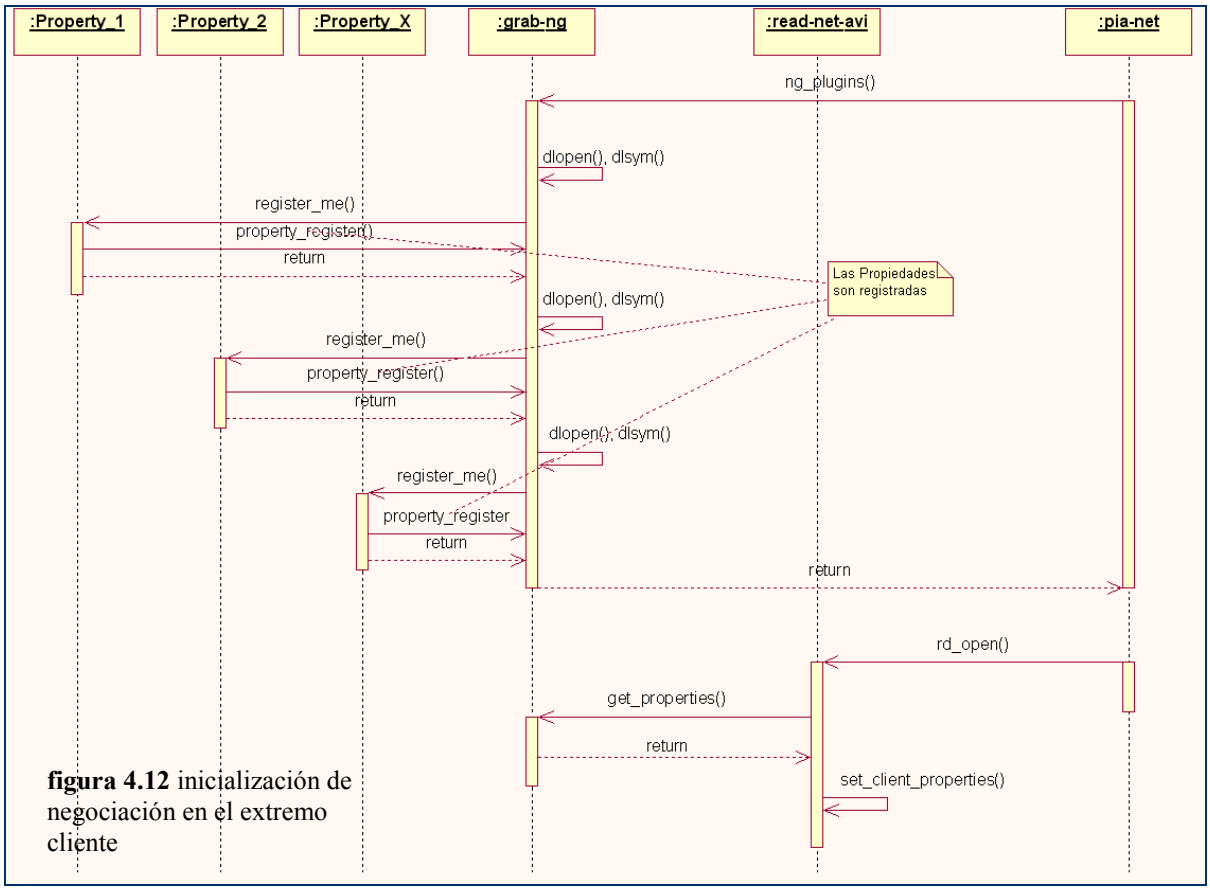


figura 4.12 inicialización de negociación en el extremo cliente

```
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

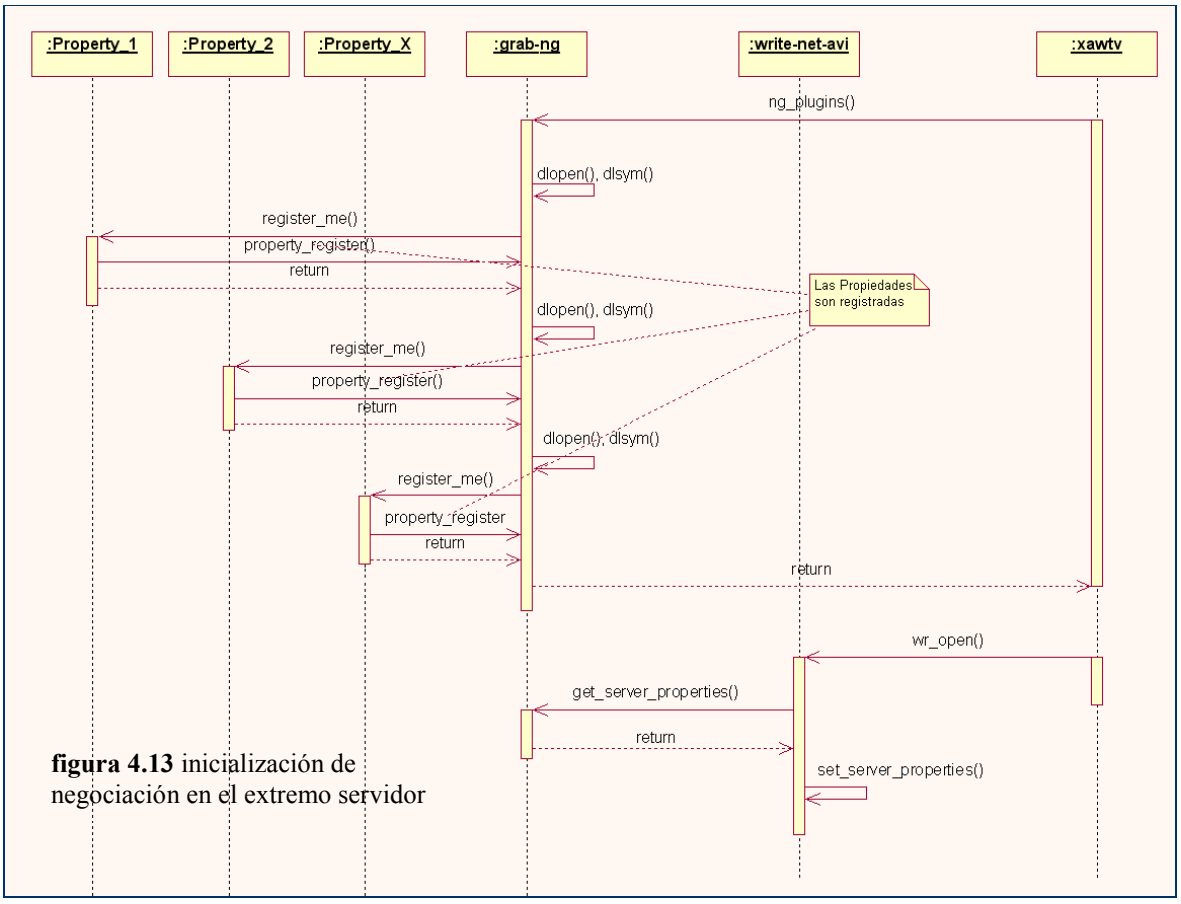


figura 4.13 inicialización de negociación en el extremo servidor

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).

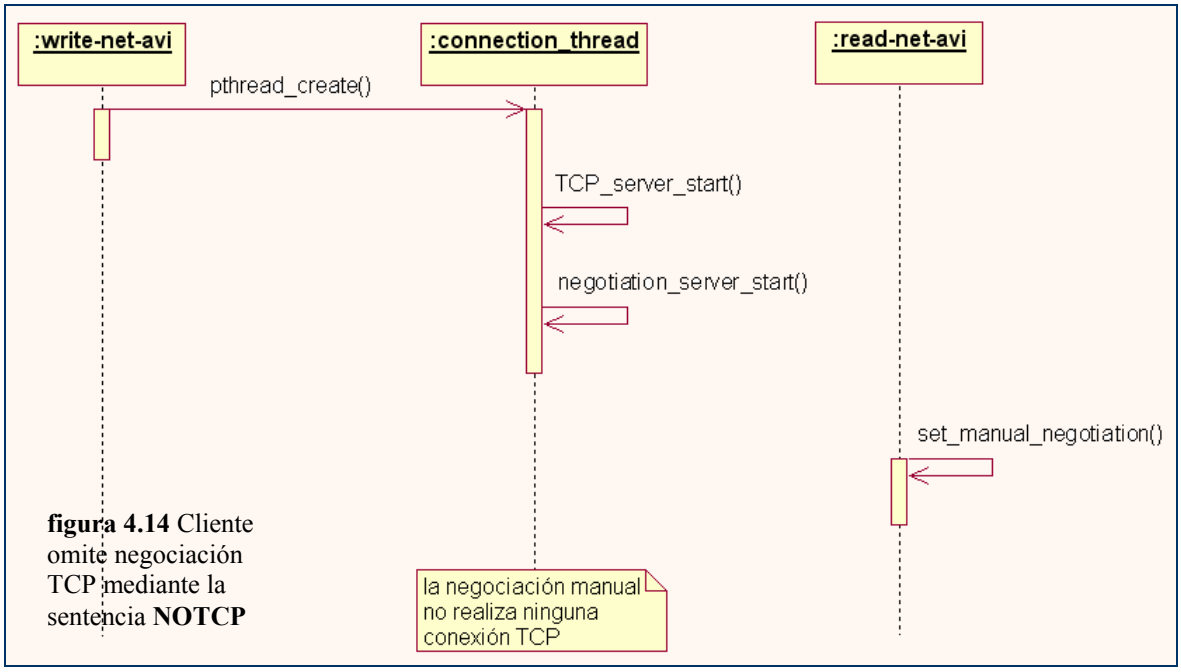


figura 4.14 Cliente omite negociación TCP mediante la sentencia **NOTCP**

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.

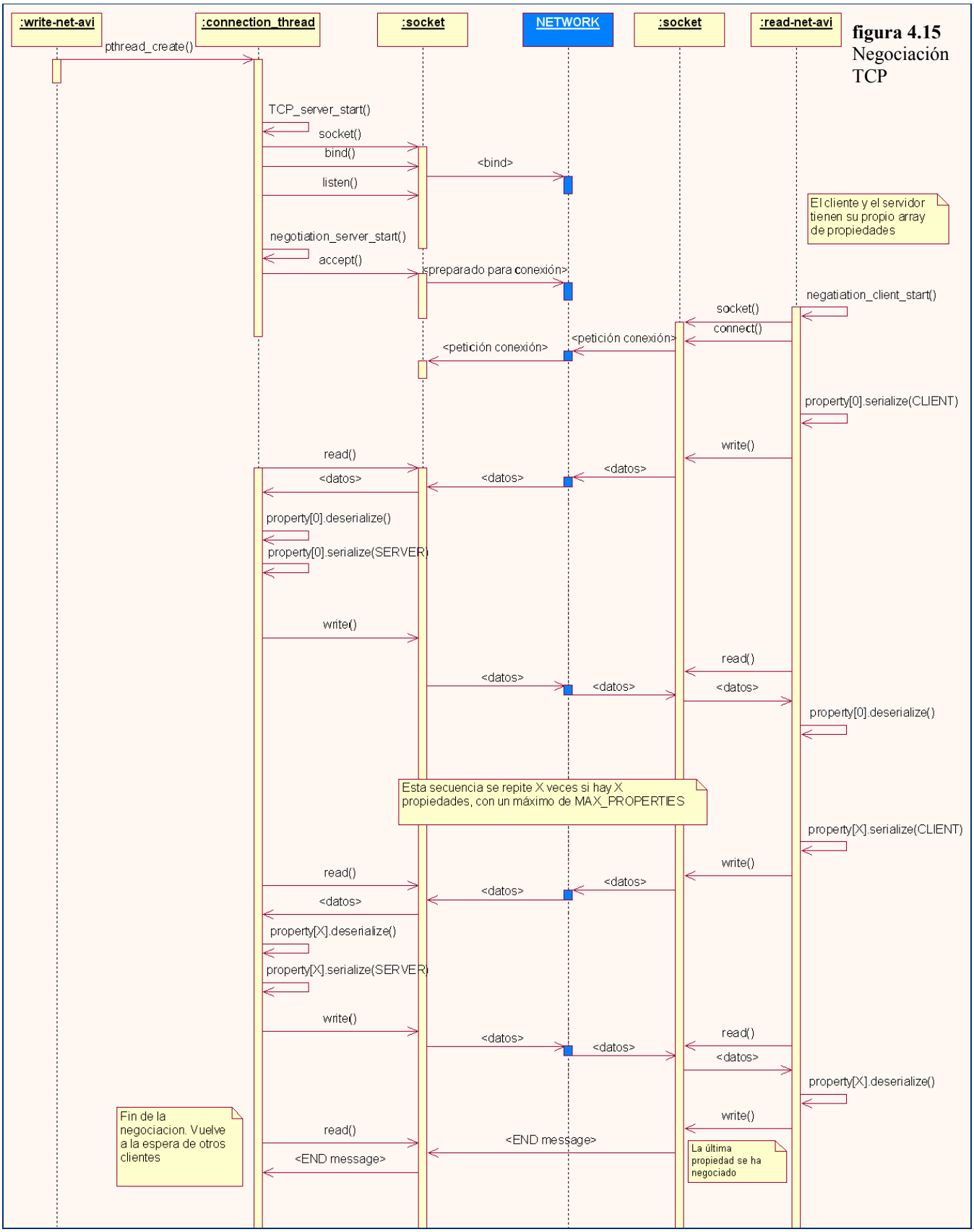
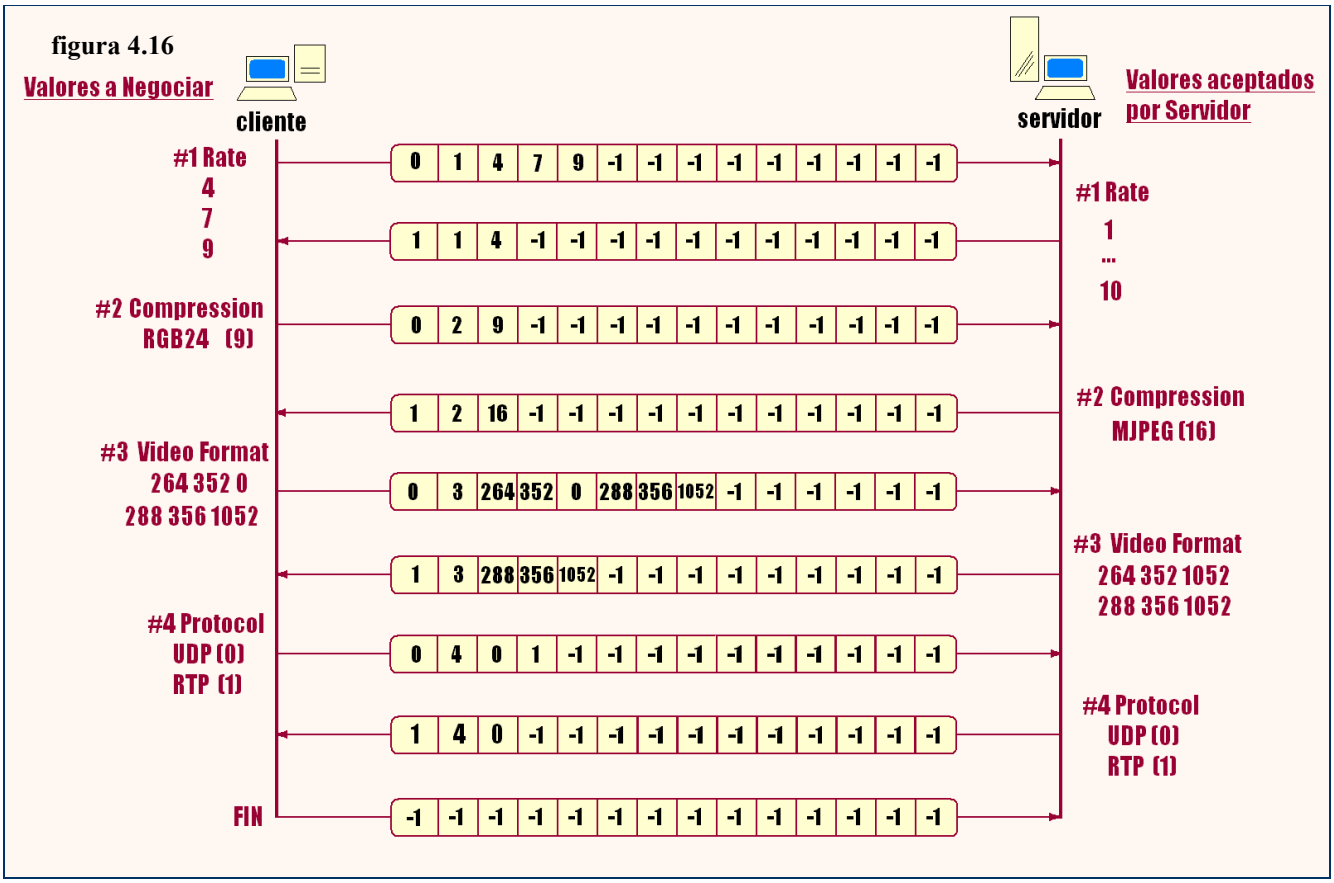


figura 4.15 Negociación TCP

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

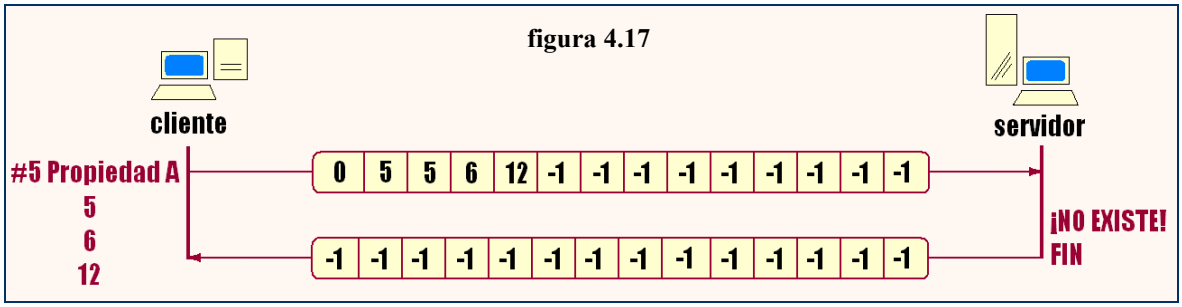


figura 4.17

5 Controladores

5.1 Introducción

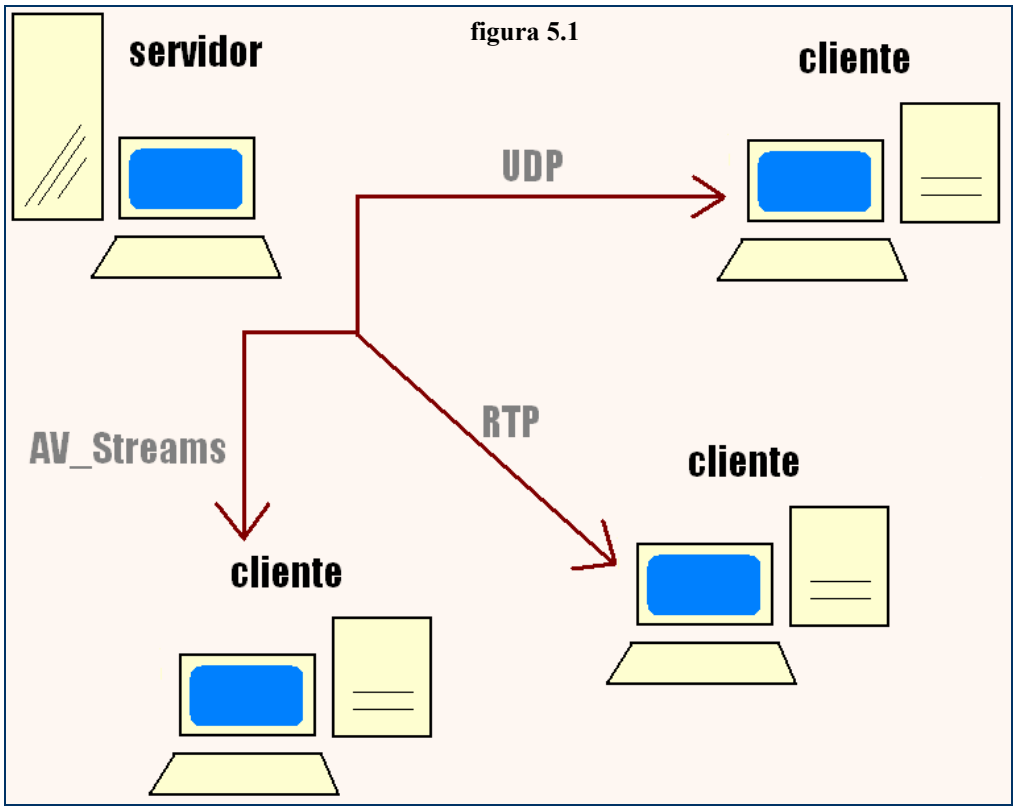
Existen distintos protocolos que pueden usarse para la transmisión de video, algunos específicos como son RTP, AVStreams de Corba… otros aunque no específicos sí que pueden ser adecuados por sus características, como por ejemplo UDP.

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.

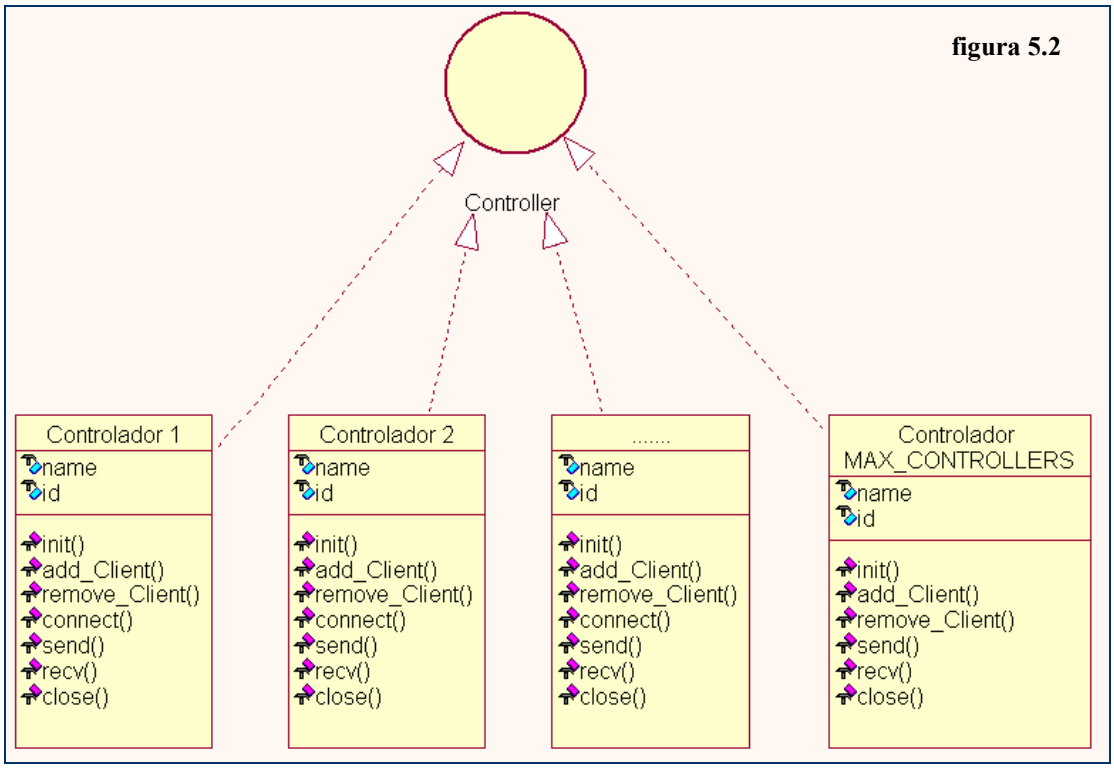


figura 5.2

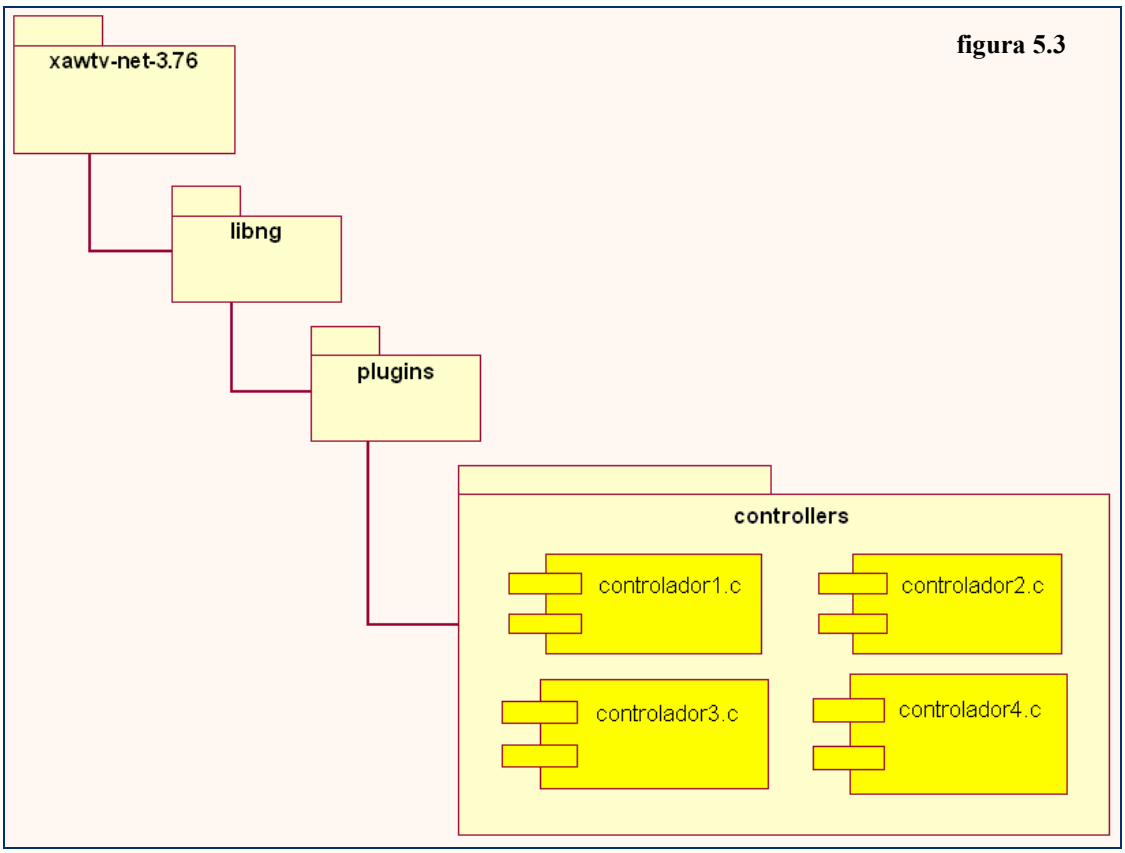
5.3 Controladores y *xawtv*

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).



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

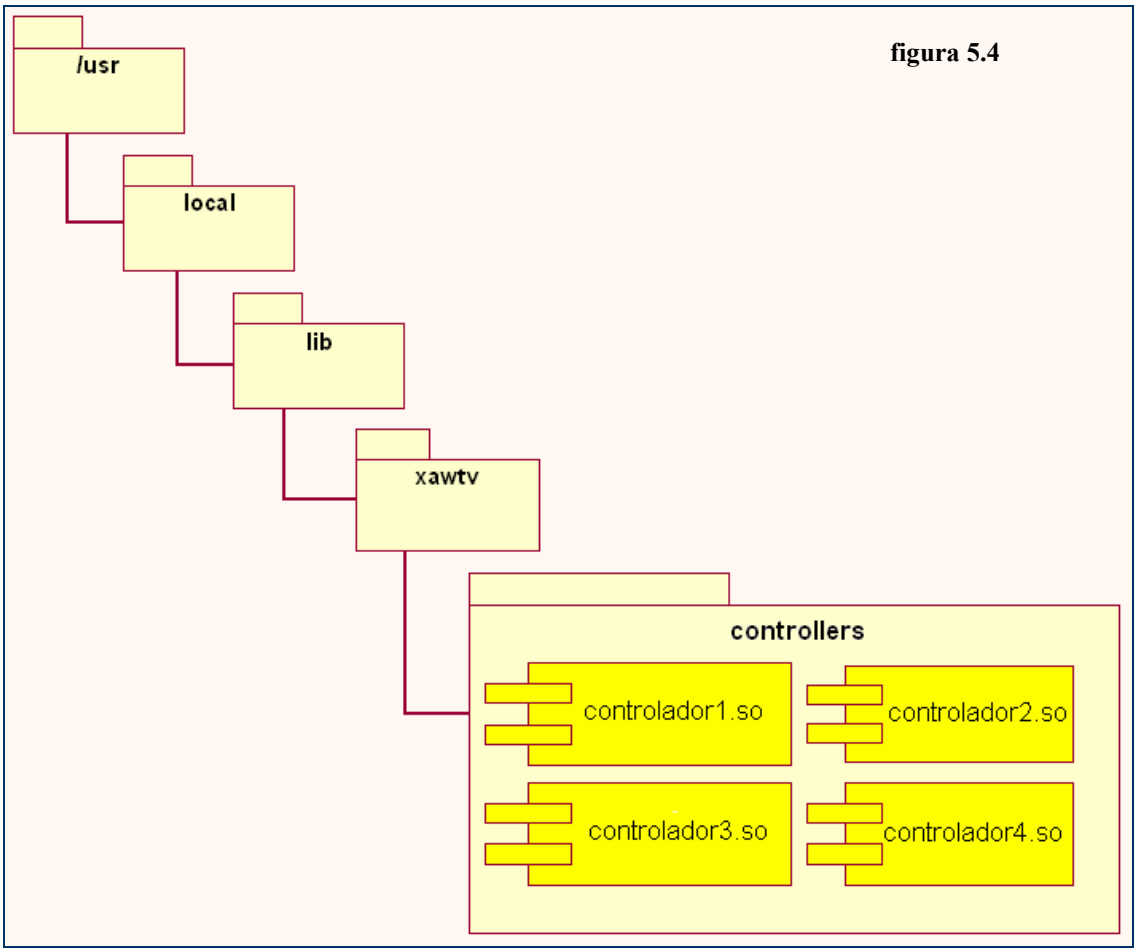


figura 5.4

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*.

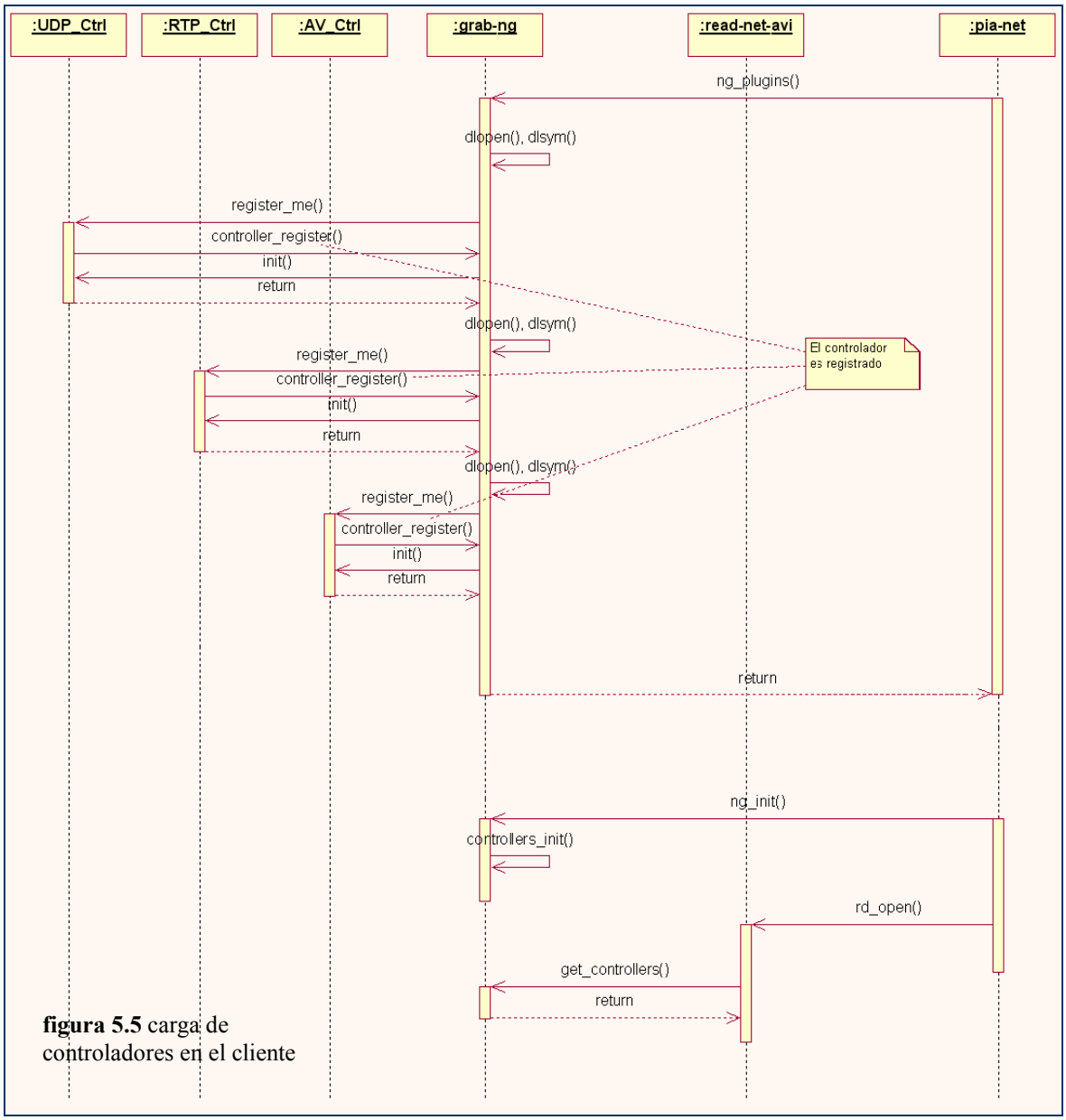


figura 5.5 carga de controladores en el cliente

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 [15] 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 8 *Kbytes* en la implementación de UDP en los sistemas operativos de Sun, o 64 *Kbytes* 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

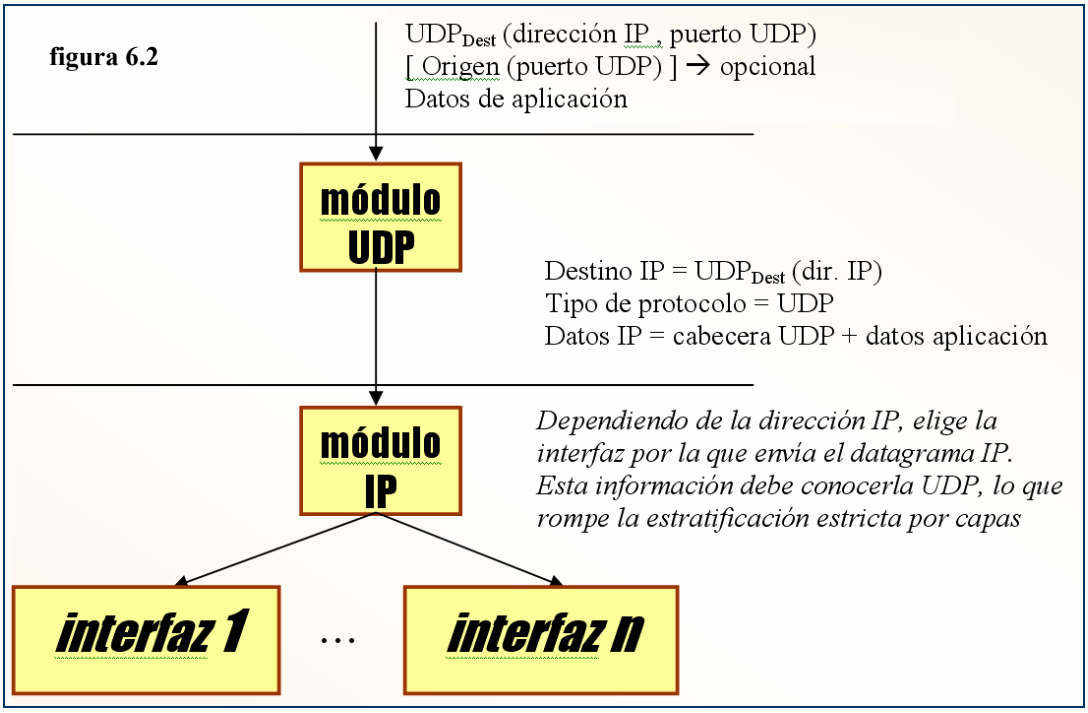
figura 6.1

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

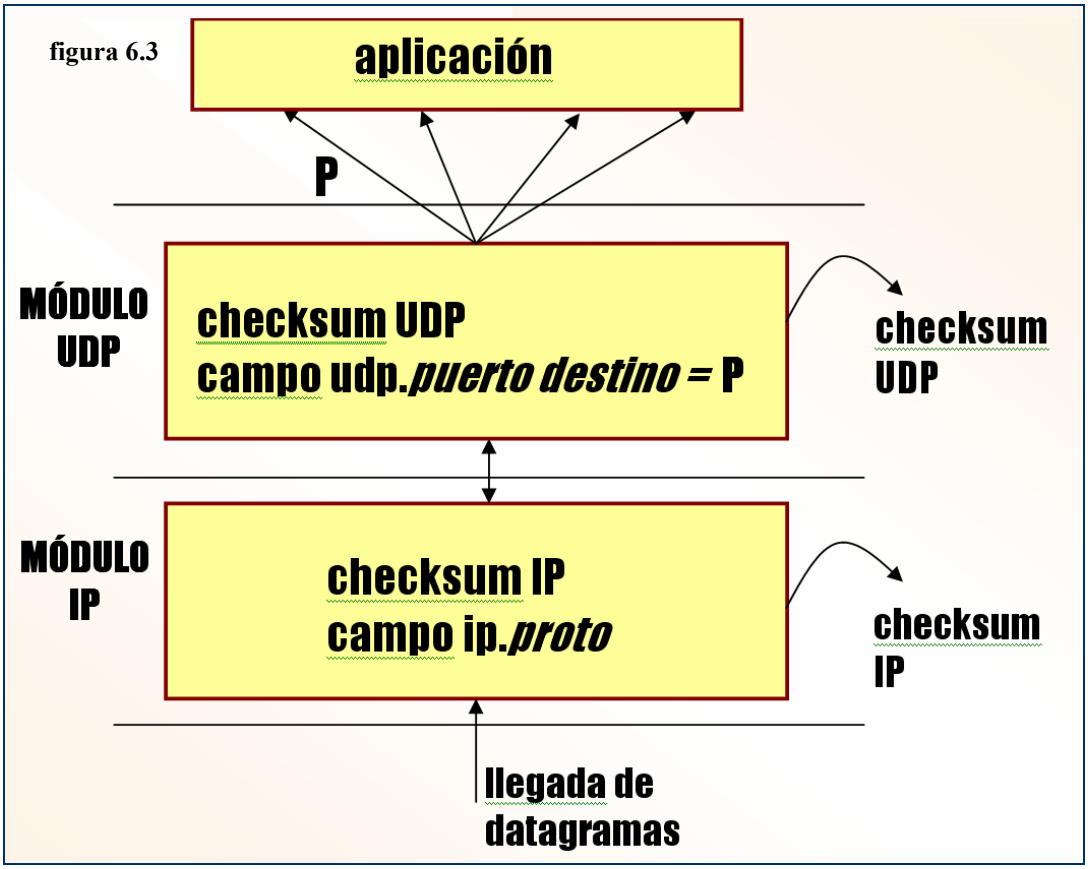
- **Puerto fuente**: En caso de que se pretenda que el destino responda a este datagrama UDP, se le puede indicar un puerto UDP de mi máquina a donde puede dirigirse. En caso contrario, lleva el valor 0.
- **Puerto destino**
- **Longitud**
- **Suma verificación**: Chequea datos UDP, cabecera UDP, y parte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

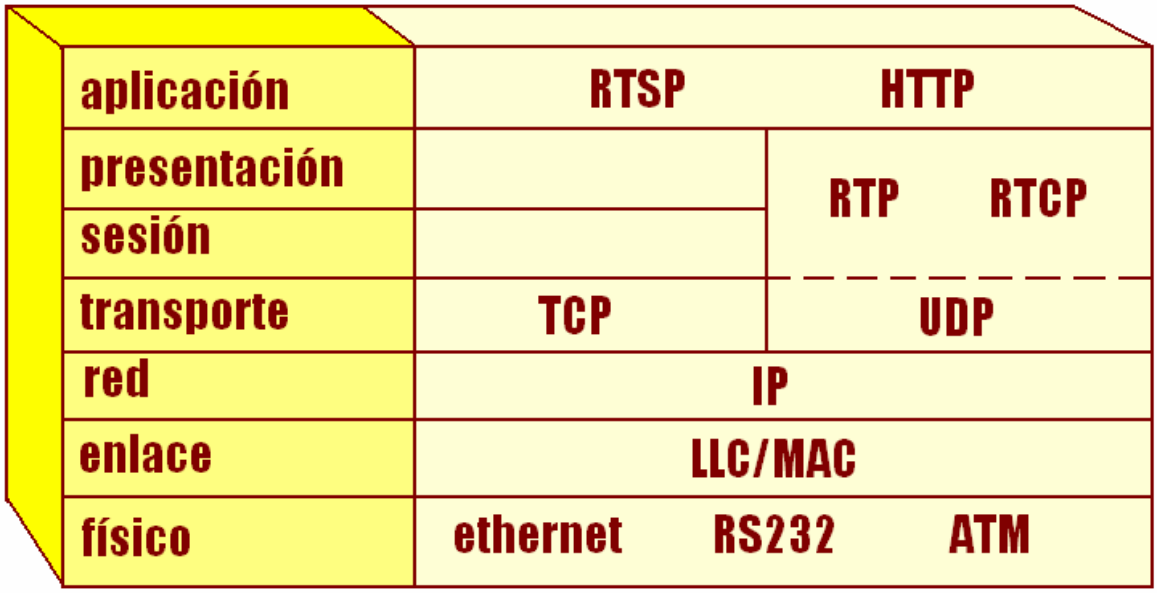
La alternativa a UDP es TCP, pero por su naturaleza no es el protocolo más indicado. Recordemos que TCP:

- 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



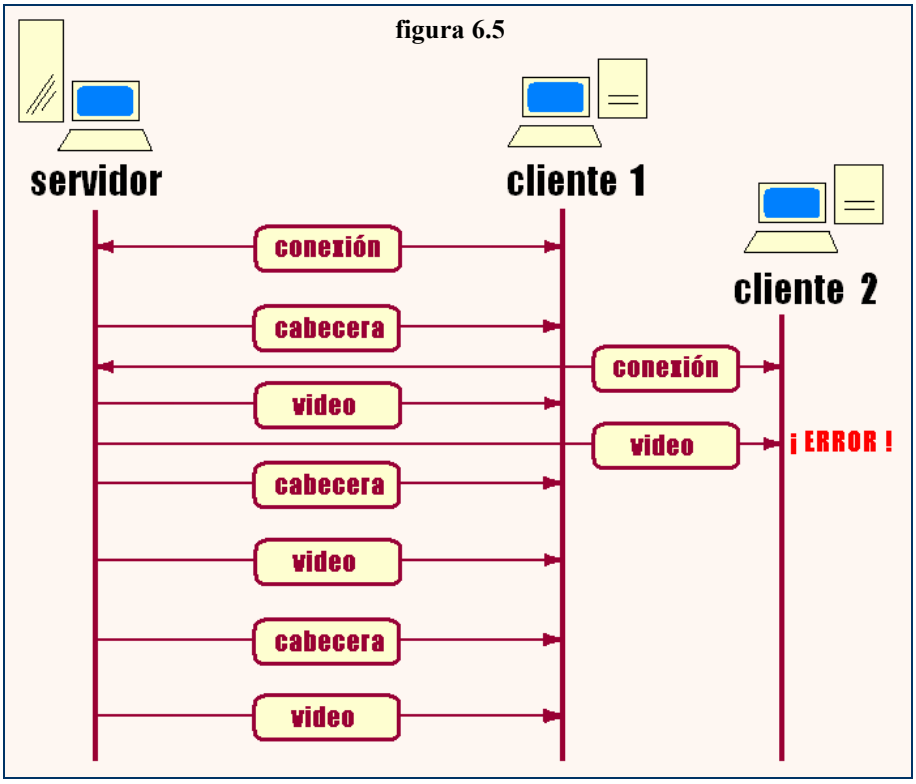
6.2.1 Sincronización. Fragmentación de imágenes.

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

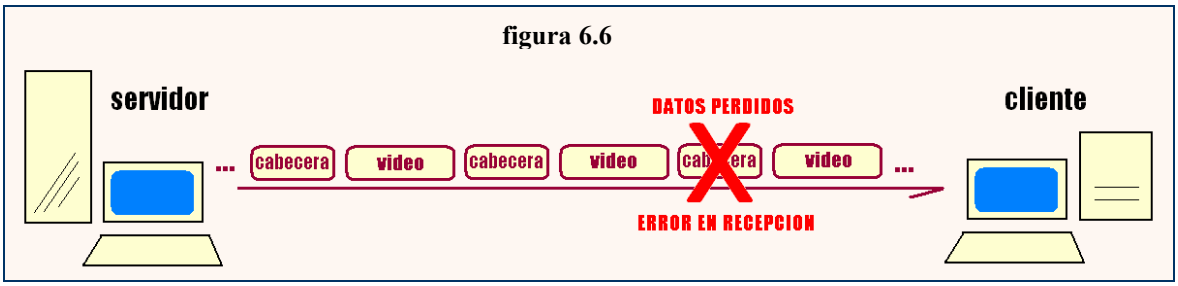
Una vez que conocemos qué tipo de información es interesante, debemos buscar la forma de enviarla. Parece claro que debe recibirse justo antes de cada *frame*, de forma que el servidor alternara el envío de información con el de imágenes.

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

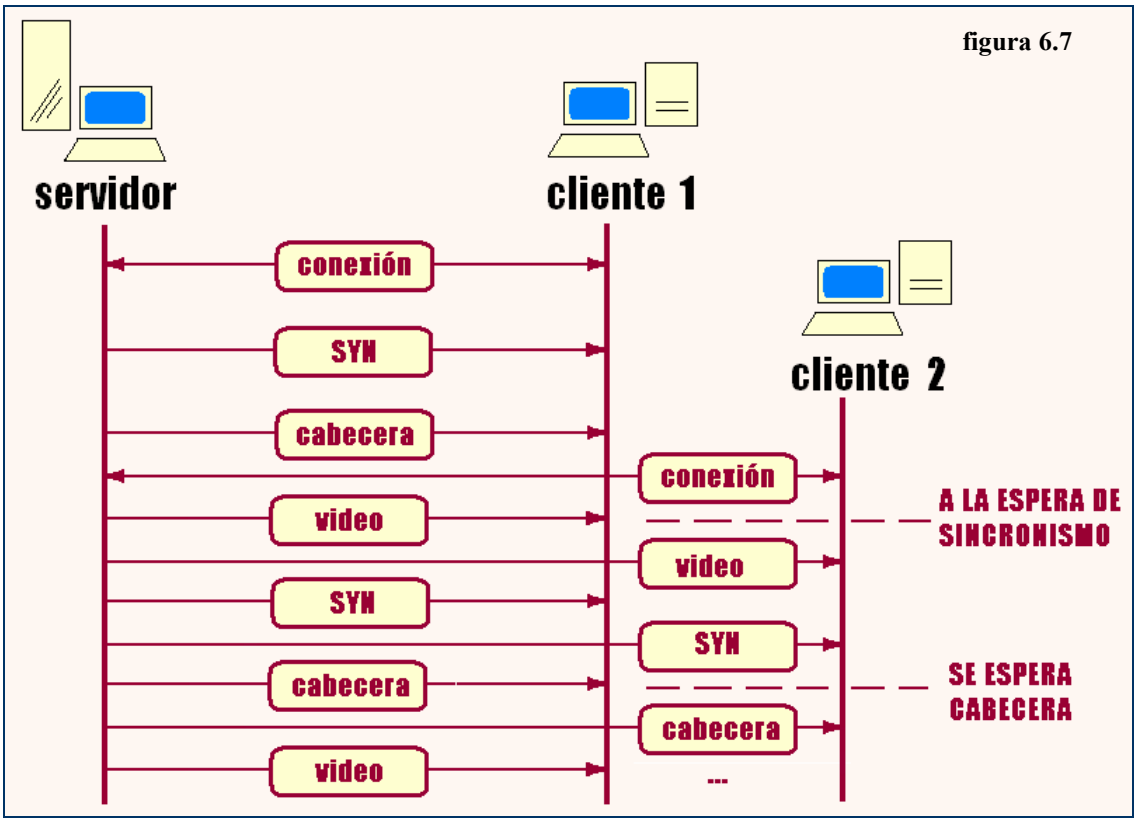
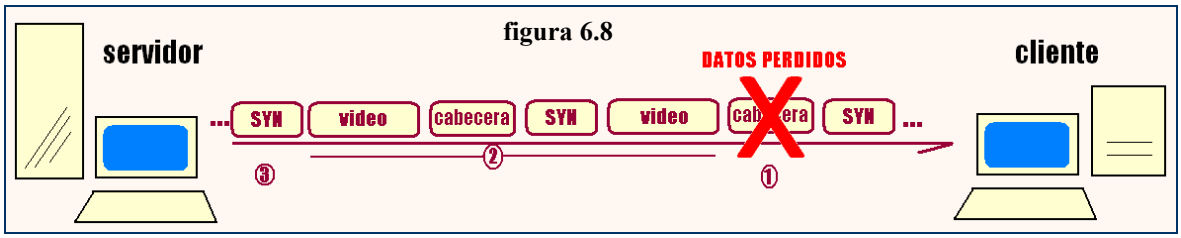


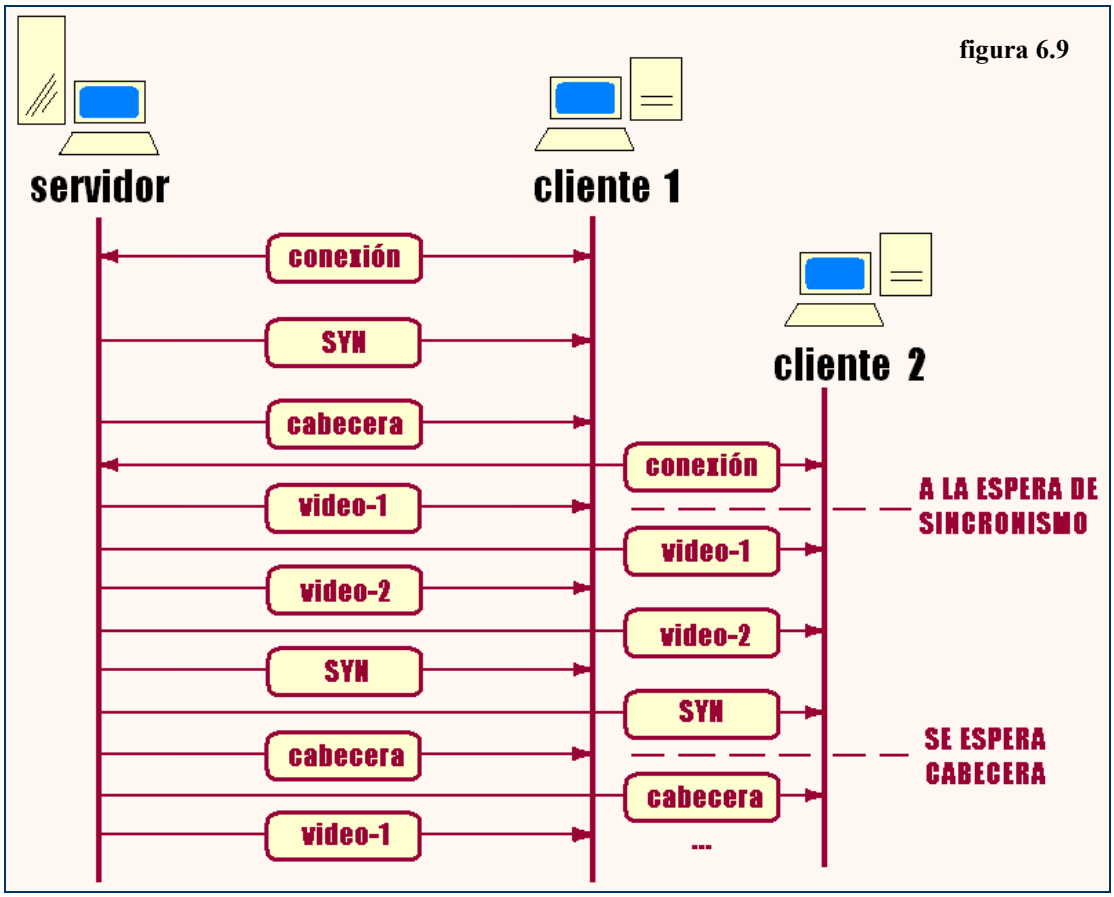
figura 6.8



(**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.

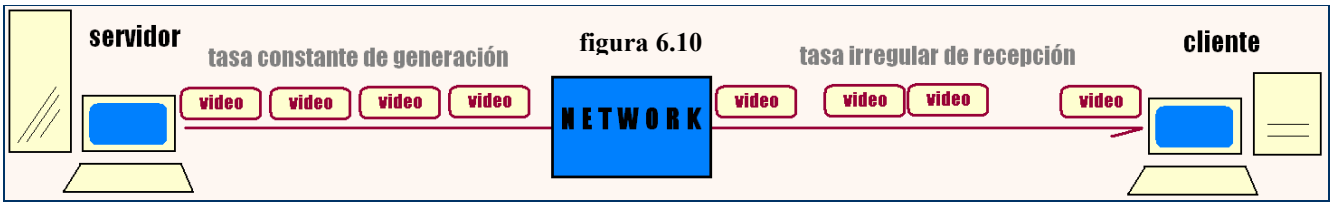


figura 6.10

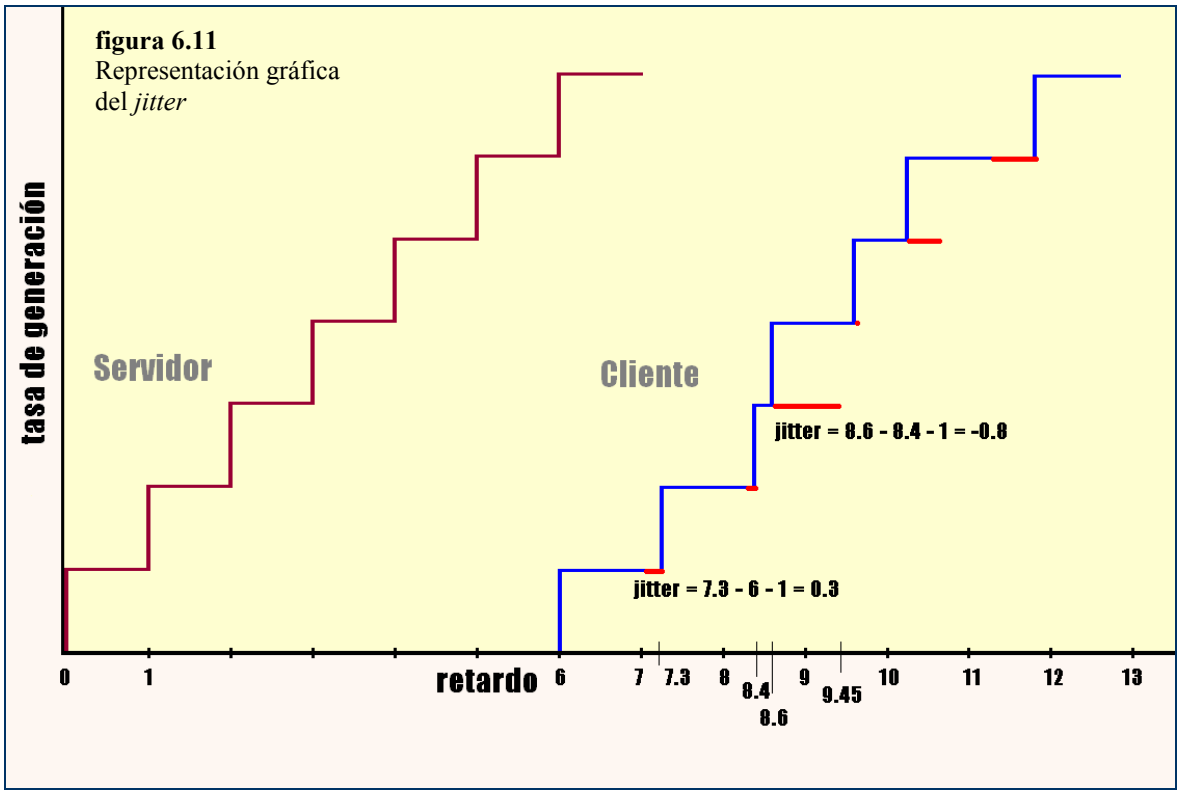
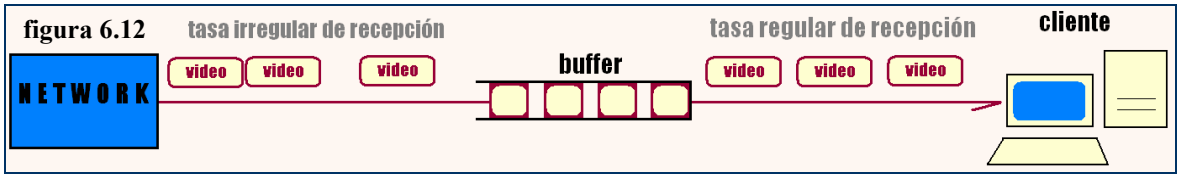


figura 6.11
Representación gráfica del *jitter*

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.

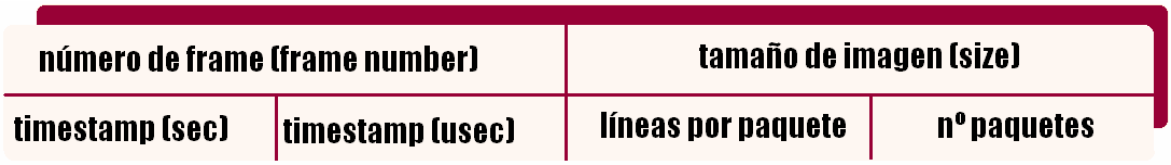
Para compresión MJPEG o JPEG**,** no es necesaria la fragmentación. En las pruebas realizadas en el laboratorio el tamaño de cada imagen estaba (como se ha dicho) entre 8 y 15 *Kbytes*. Es probable que con una cámara de mayor calidad, las imágenes capturadas sean de más tamaño y se necesite fragmentar. 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.

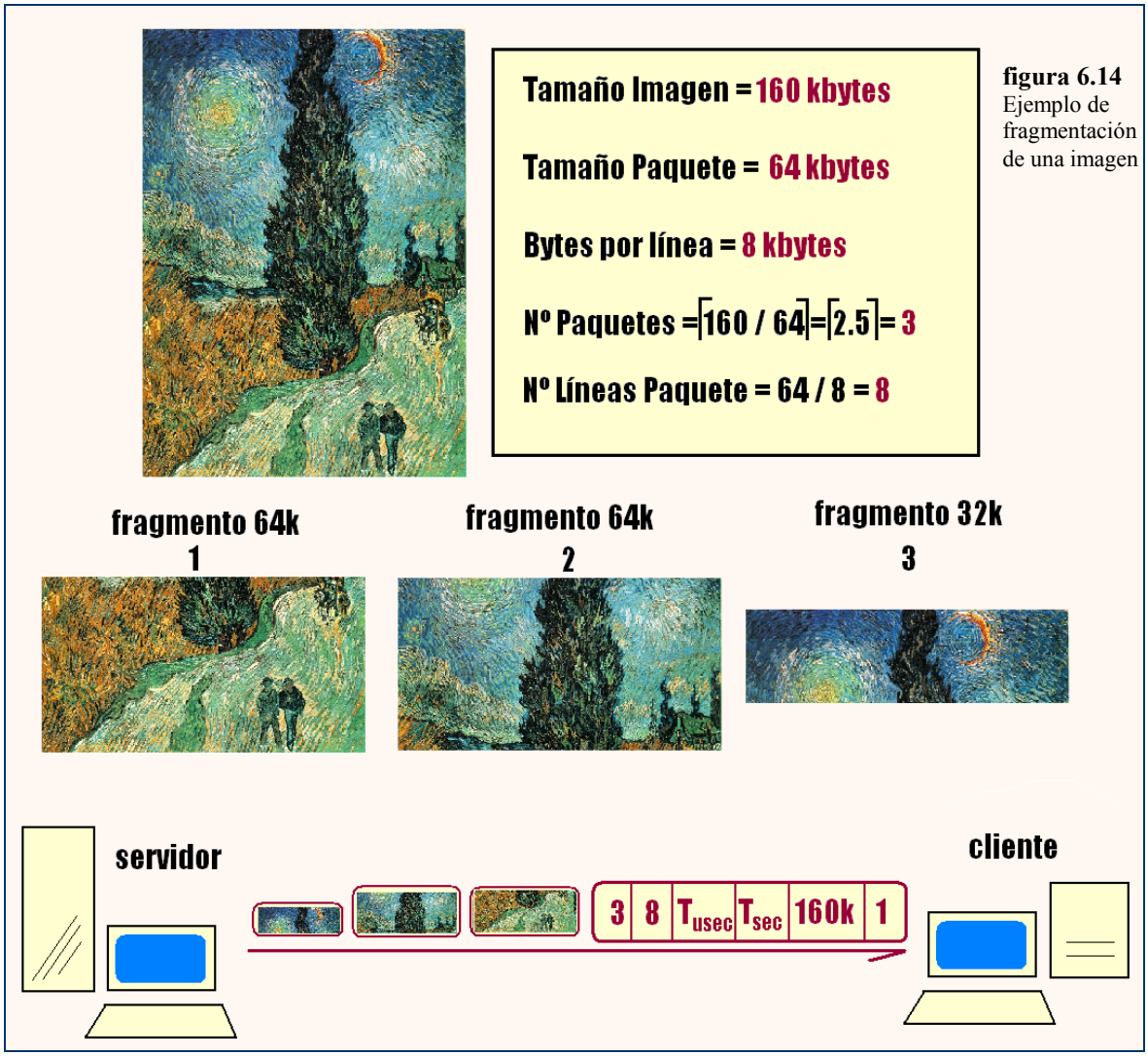


figura 6.14 Ejemplo de fragmentación de una imagen

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.

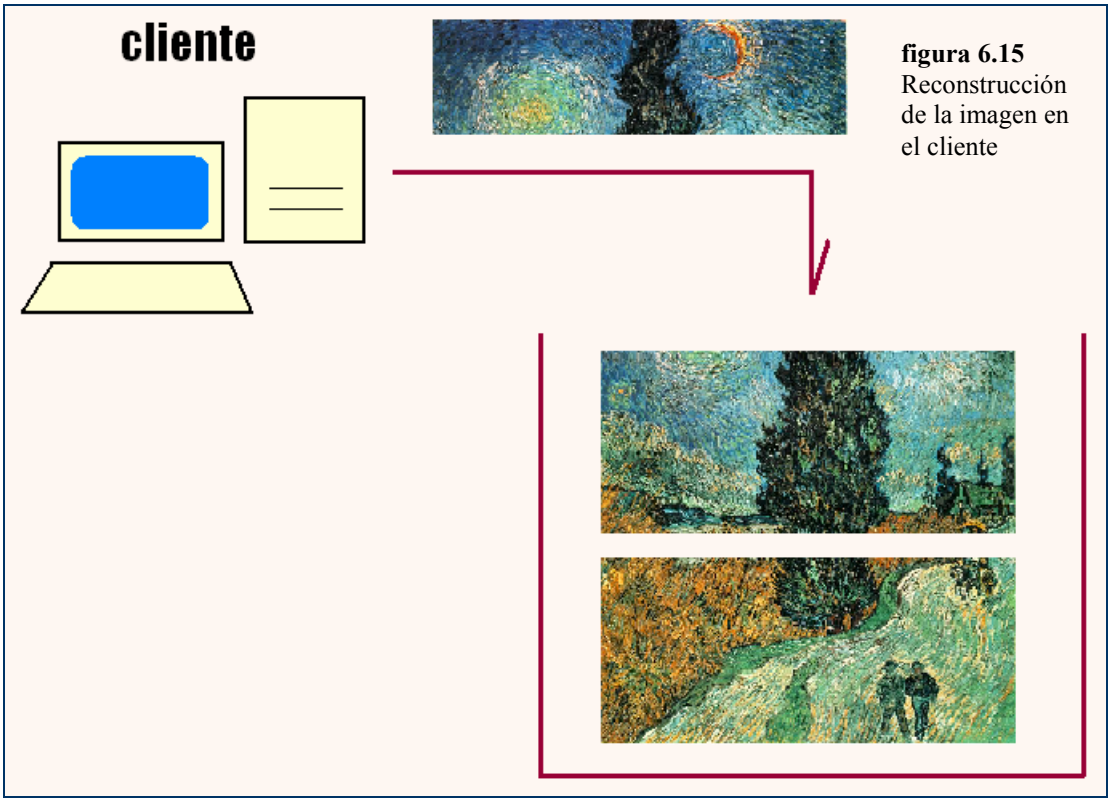


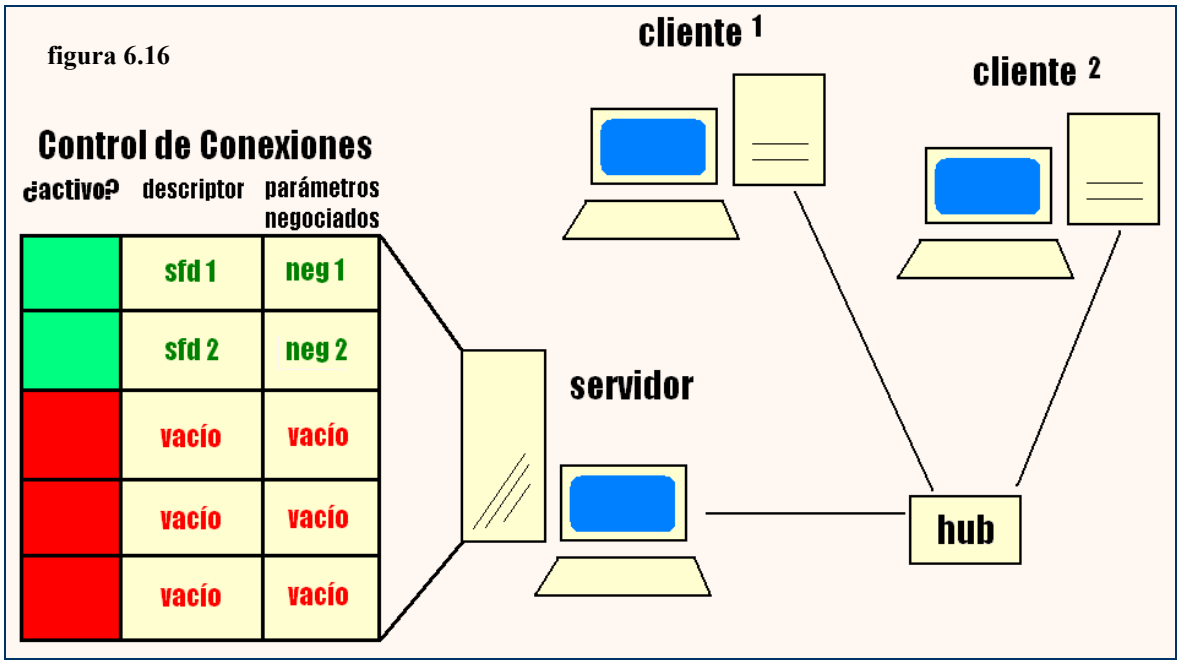
figura 6.15 Reconstrucción de la imagen en el cliente

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.

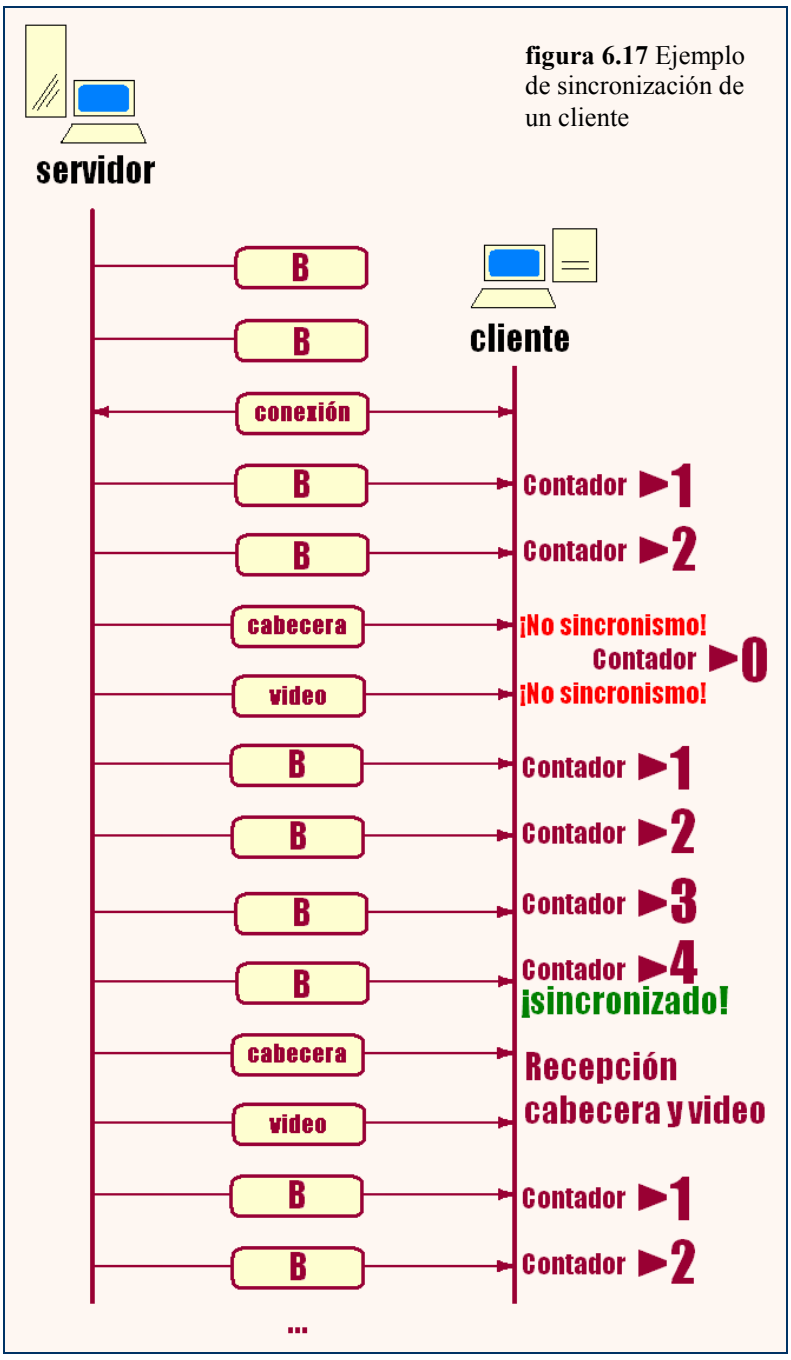


figura 6.17 Ejemplo de sincronización de un cliente

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

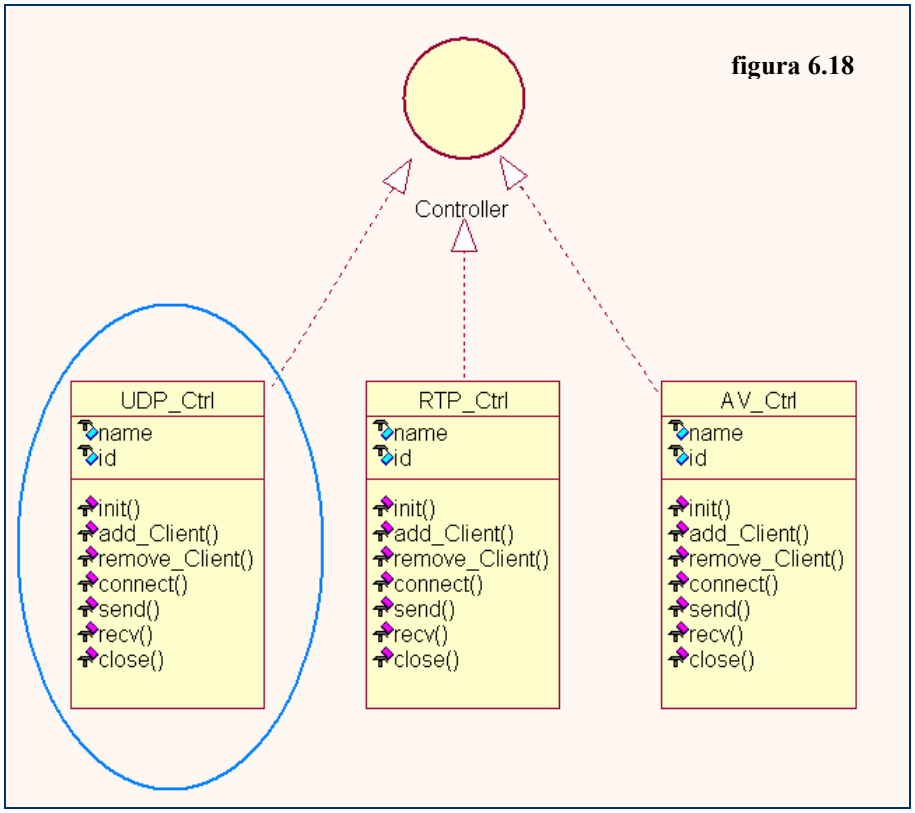


figura 6.18

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.

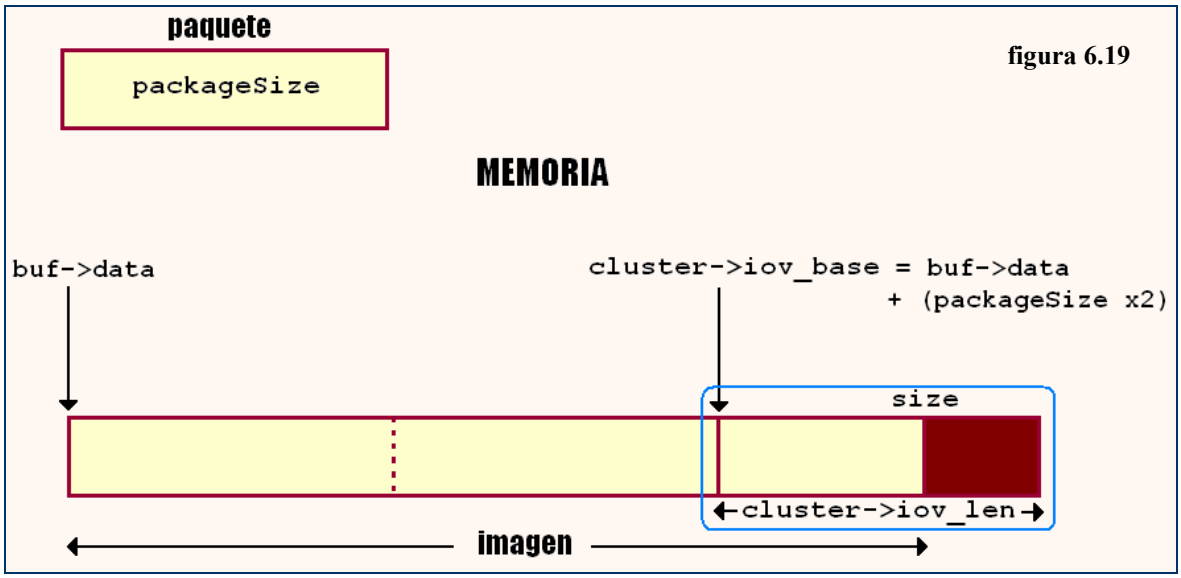


figura 6.19

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.

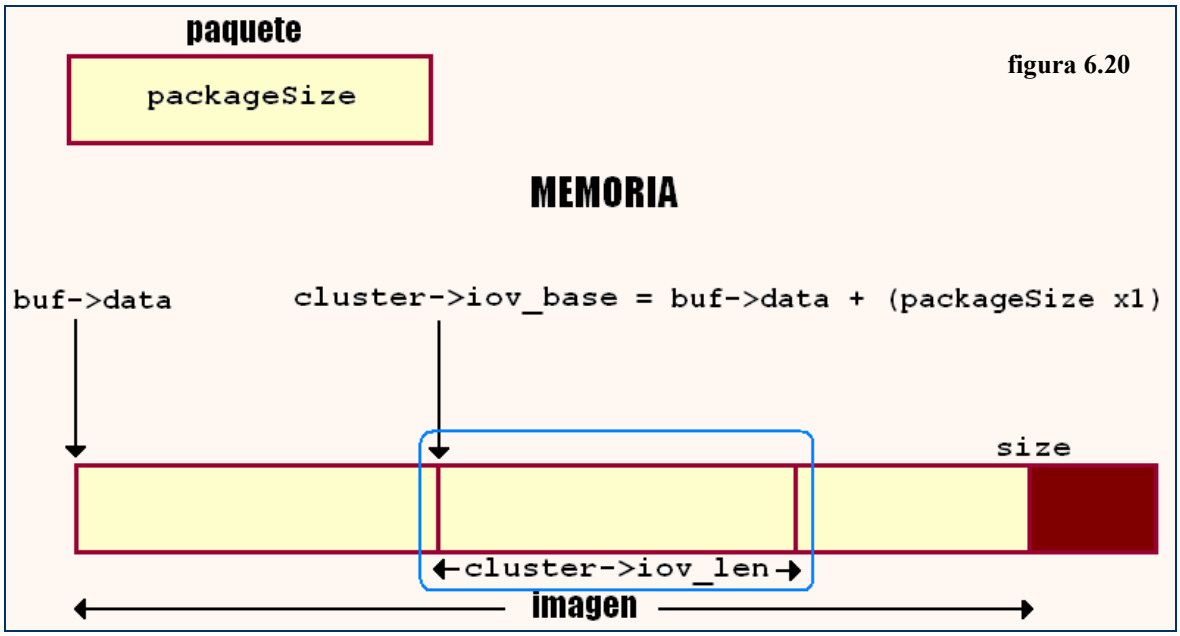


figura 6.20

En el siguiente bucle, j=0:

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

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

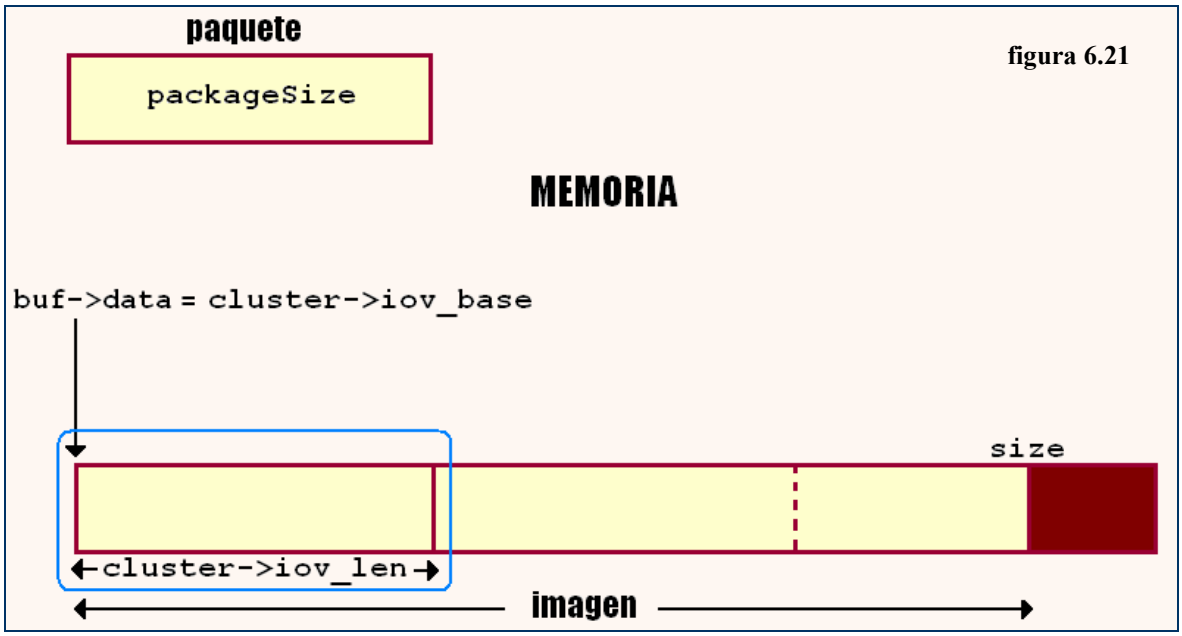


figura 6.21

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.

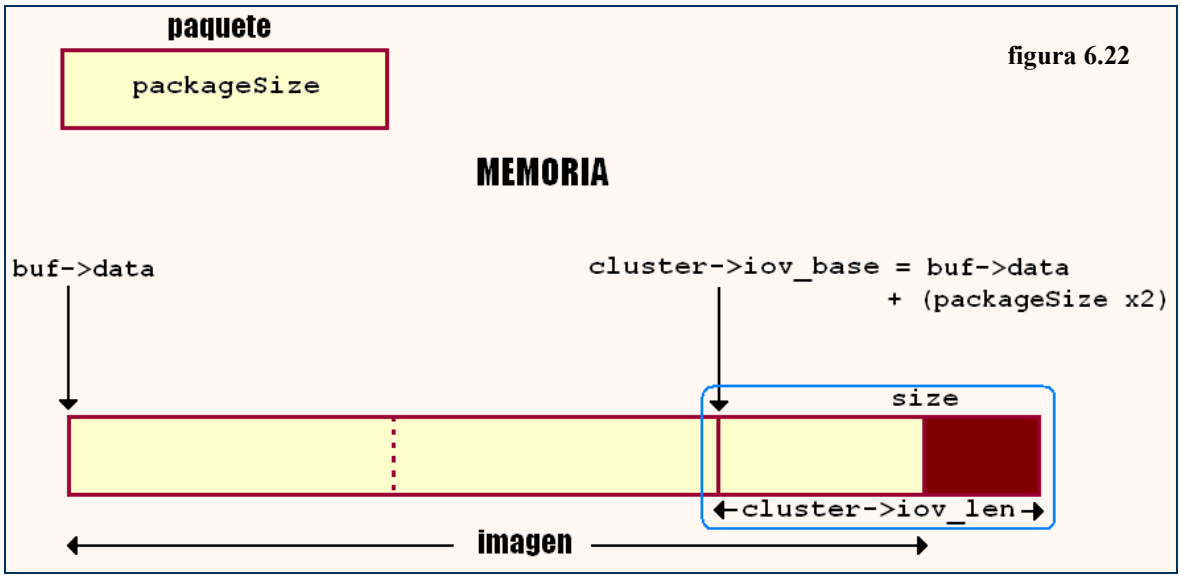


figura 6.22

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.

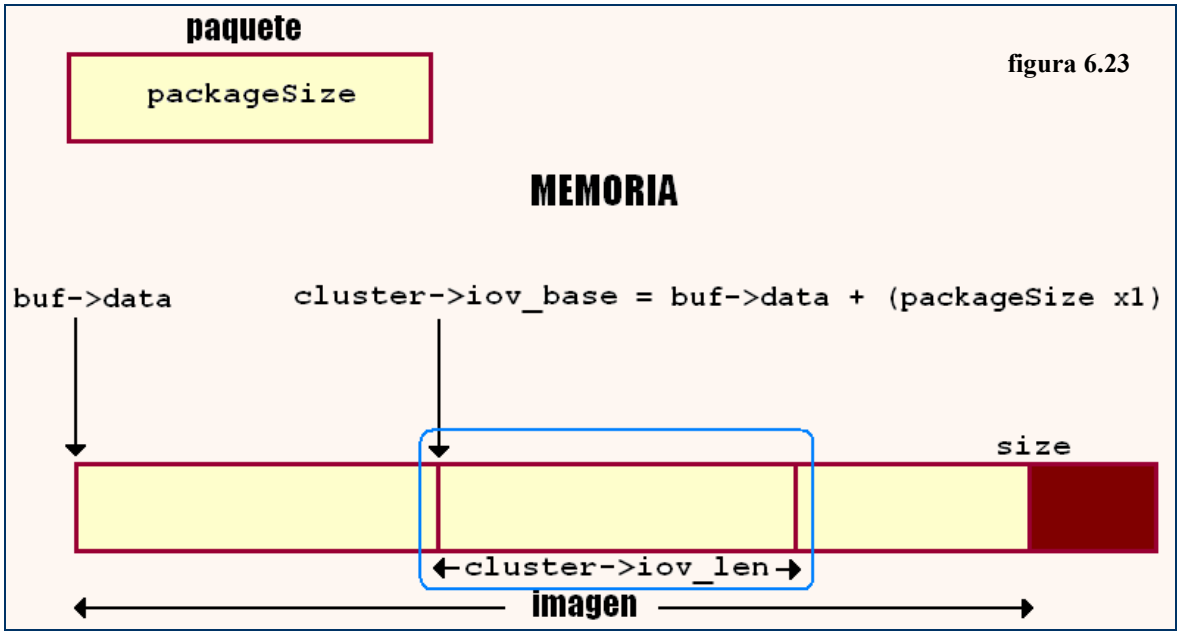


figura 6.23

En el siguiente bucle, j=0:

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

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

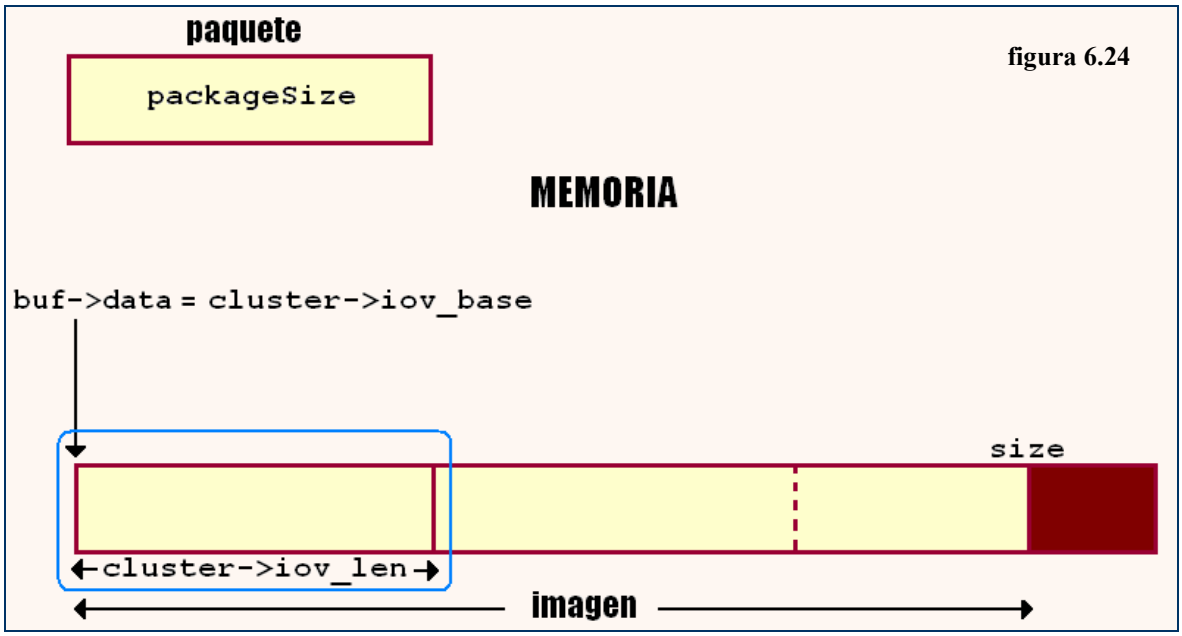


figura 6.24

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

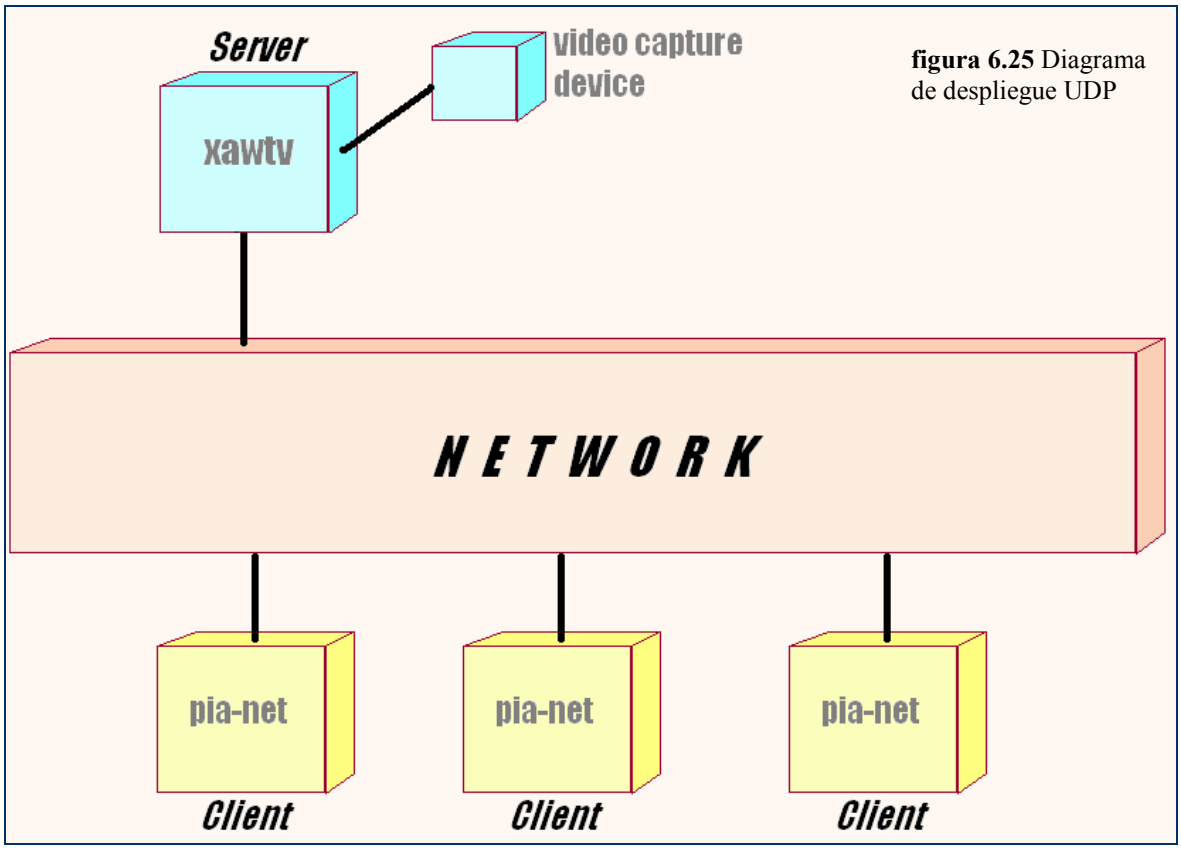


figura 6.25 Diagrama de despliegue UDP

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.

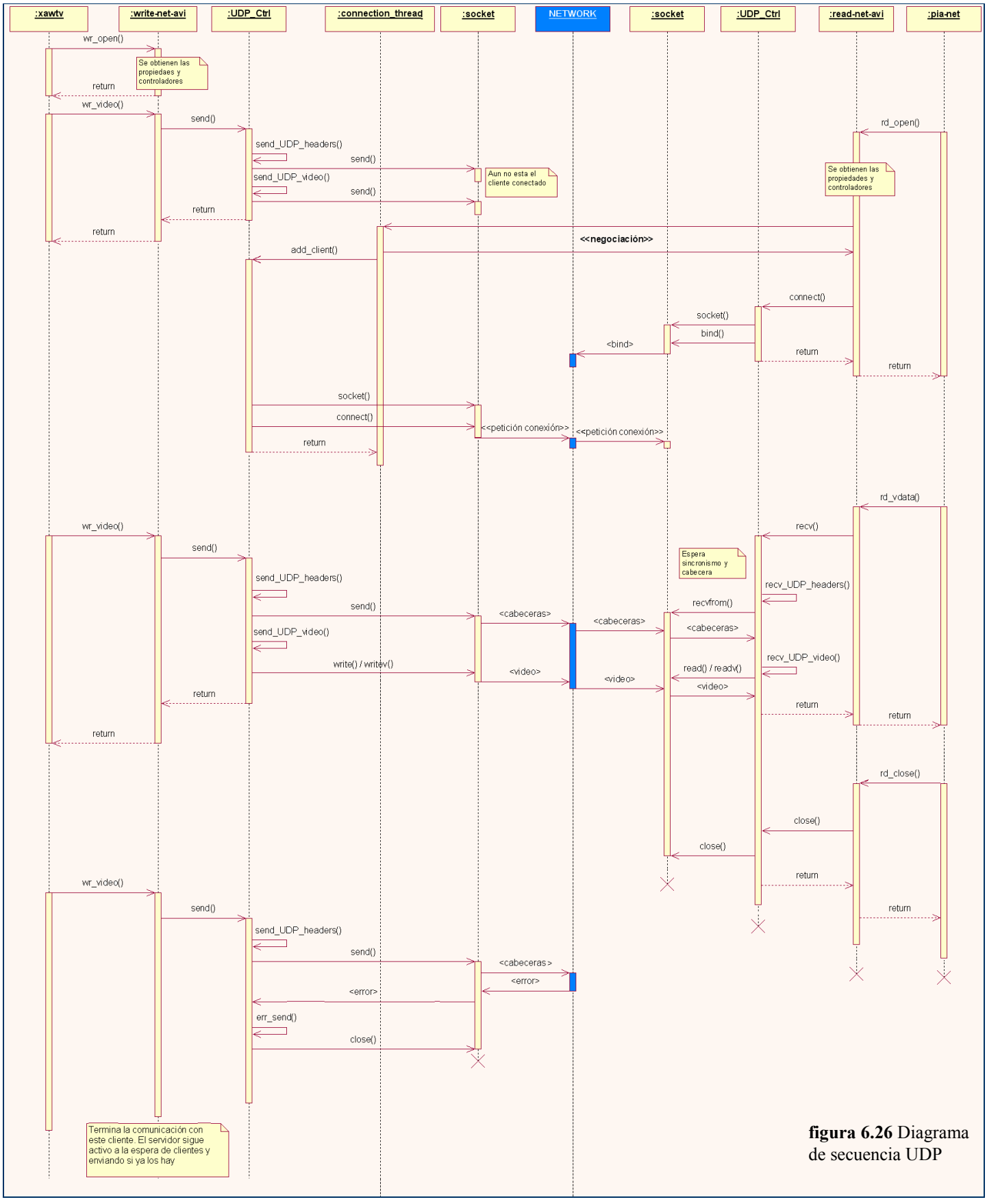


figura 6.26 Diagrama de secuencia UDP

7 Controlador para transmisión RTP

7.1 Introducción a RTP

El 22 de Noviembre de 1995, RTP [16] fue aprobado por el IESG (*Internet Engineering Steering Group*) como un estándar propuesto para internet. Sus publicaciones son:

- RFC 1889, *RTP: A Transport Protocol for Real-Time Applications*
- RFC 1890, *RTP Profile for Audio and Video Conferences with Minimal Control*

Hoy en día, empresas como Intel, Microsoft, y un consorcio de otras 100 empresas integran las especificaciones RTP en sus productos. Programas como *Microsoft NetMeeting*, o el *Netscape LiveMedia framework* están basados en el protocolo de transporte a tiempo real RTP (*Realtime Transport Protocol*). [17]

La IESG recibe continuamente nuevas propuestas acerca de actualizaciones y mejoras de RTP. [18]

RTP consta de una parte de datos y otra de control, esta última es RTCP. La parte RTP de datos es un protocolo que provee soporte para aplicaciones en tiempo real tales como flujos continuos multimedia (audio, video…) incluyendo reconstrucción de los tiempos, pérdida de detección, seguridad e identificación del contenido.

RTCP (*RTP control protocol*) da soporte a conferencias en tiempo real para grupos de cualquier tamaño a través de internet. Este servicio incluye parámetros de calidad de servicio desde receptores a un grupo multicast, así como la sincronización de distintos flujos multimedia e identificación de extremos. RTCP se encarga de “controlar los datos”.

Algunas aplicaciones como la simulación distribuida son hoy en día objetivos a cumplir.

7.1.1 Servicio

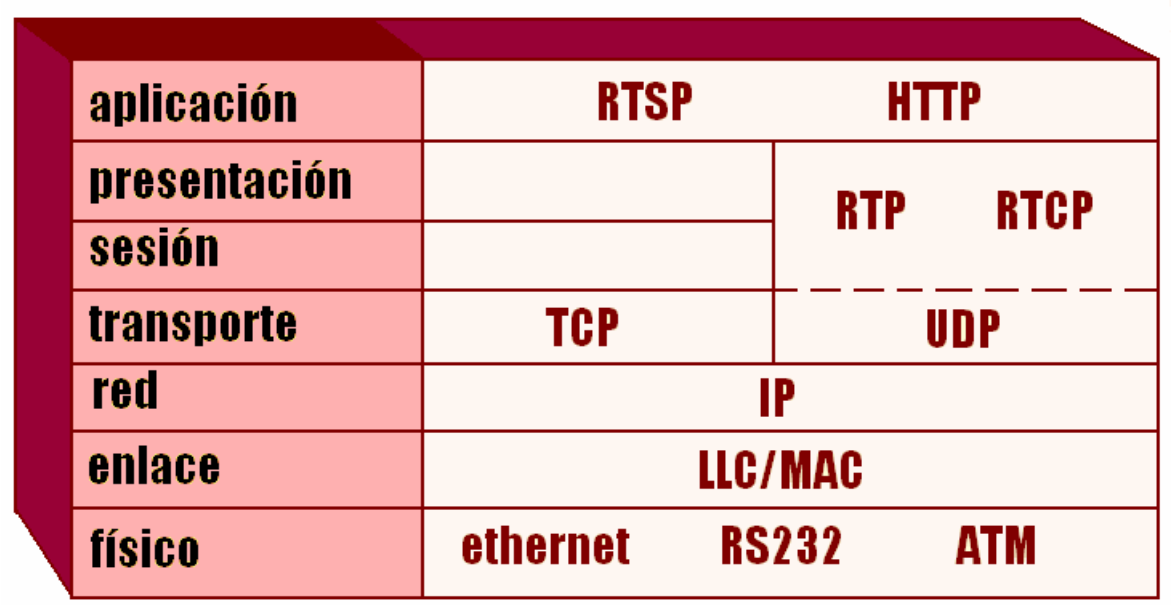
RTP proporciona servicio de entrega extremo a extremo para la transmisión de datos de tiempo real. Es independiente de la red y del protocolo de nivel de transporte subyacente, aunque por lo general se usa UDP. RTP:

- Puede ser usado en transmisiones *unicast* o *multicast*, siendo esta última mucho más eficiente
- Identifica el tipo de datos que transporta (*payload*)
- Indica el orden en el que deben ser reproducidos los datos (*sequence number*)
- Sincroniza los paquetes de diversas fuentes (*SSRC*)
- No garantiza el orden de llegada de los paquetes, ni siquiera garantiza que lleguen los paquetes

RTCP se basa en la transmisión periódica de paquetes de control a todos los participantes en una sesión, usando el mismo mecanismo de distribución, así como los mismos paquetes de datos. El protocolo subyacente debe proveer multiplexación de los datos y paquetes de control, por ejemplo usar puertos diferentes con UDP. El servicio que ofrece RTCP se puede expresar en cuatro funciones:

- La función principal es monitorizar la calidad de la transmisión.
- RTCP mantiene un identificador a nivel de transporte para una fuente RTP, este identificador es el nombre canónico *CNAME*. Debido a que el identificador *SSRC* puede cambiar si se produce un problema o el programa reinicia, los receptores pueden hacer uso del *CNAME* (que no cambia) para reconocer a la fuente RTP
- Las dos funciones anteriores requieren que todos los participantes envíen paquetes RTCP, la frecuencia con la que se envían debe estar controlada porque puede darse el caso de que exista gran número de participantes. Cada participante puede observar el número de participantes. Este número se usa para calcular la frecuencia con la que los paquetes de control son enviados.
- Una función opcional es transportar la mínima información de control sobre la sesión (por ejemplo identificación de un participante para que pueda ser mostrada en una interfaz de usuario). Es útil en sesiones en las que los participantes se conectan y desconectan sin llevar un control de miembros.

figura 7.1 Arquitectura RTP



7.1.2 Definiciones

Antes de estudiar el formato de las tramas RTP y RTCP, y debido a la sintaxis a utilizar a partir de ahora, es imprescindible comprender una serie de conceptos. Son los siguientes:

RTP payload

Son los datos transportados por un paquete RTP (datos de video comprimidos o audio…). Existe un **Payload Type** que informa sobre el tipo de datos, por ejemplo *Payload Type = 26*, indica formato de compresión JPEG para video. En la web http://www.iana.org/assignments/rtp-parameters se encuentra una lista con todos los tipos de *payload* registrados y aceptados.

RTP packet

Un paquete RTP consta de una cabecera fija, una posible lista de *CSRC* (ver definición) y de datos (*payload*). Algunos protocolos subyacentes pueden requerir una encapsulación del paquete RTP. Normalmente un paquete perteneciente al protocolo subyacente contiene un solo paquete RTP (por ejemplo UDP), pero también puede contener varios paquetes RTP si el método de encapsulación lo permite, lo que sería ventajoso pues simplificaría la sincronización entre distintas fuentes y reduciría el *overhead* de cabeceras. Normalmente se usa UDP, por lo que esto último no es habitual.

RTCP packet

Un paquete de control consta de una cabecera fija similar a la de los paquetes RTP, seguida por unos elementos estructurados que varían dependiendo del tipo de paquete RTCP. Los distintos tipos de paquetes RTCP se verán en el apartado 7.1.4.

Normalmente se envían juntos múltiples paquetes RTCP como una “mezcla” de paquete RTCP en un único paquete del protocolo subyacente (UDP).

RTP session

Se puede definir como la asociación entre una serie de participantes comunicándose mediante RTP. Para cada participante, la sesión se define mediante una dirección IP mas un puerto para RTP y otro para RTCP. Una sesión, puede distinguirse por el par de puertos y/o por la dirección IP.

Synchronization source (SSRC)

La fuente de un flujo de paquetes RTP es identificada mediante un número de 32 bits llamado identificador SSRC. Lo contiene la cabecera RTP para no ser dependiente de la dirección de red. Todos los paquetes con un mismo SSRC mantienen sus números de secuencia y valores de *timing*. Así, un receptor o un grupo de receptores pueden sincronizar la fuente para la reproducción de video/audio. Un ejemplo de la sincronización, sería que una fuente podría enviar un mismo video pero con distintos formatos de compresión (o video procedente de distintas cámaras), en ese caso aunque la fuente tiene una misma dirección IP, los clientes podrían diferenciar los paquetes RTP que corresponden a un video o a otro a partir de su SSRC. Es más, en el envío de audio y video simultáneamente, la información de audio debe estar identificada con un SSRC distinto al del video. El SSRC puede cambiar, pero no afecta a los clientes, pues pueden detectarlo a través de otro identificador (*CNAME*).

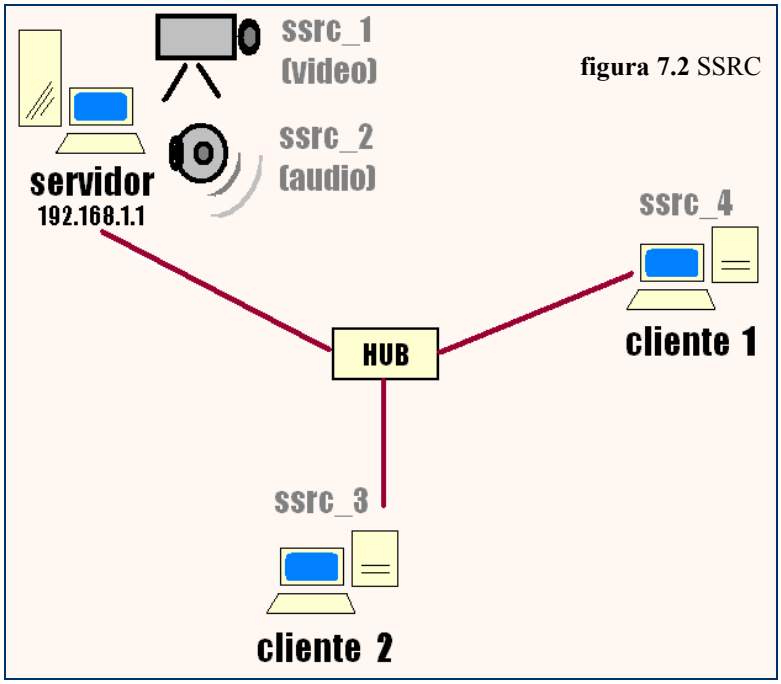


figura 7.2 SSRC

Contributing source (CSRC)

Es una fuente de un flujo de paquetes RTP que ha contribuido al flujo combinado que procede de un **RTP mixer** (ver definición). El **mixer** inserta una lista de identificadores SSRC de las fuentes que han contribuido a la generación de un paquete a la cabecera RTP de ese paquete. Esta lista es llamada lista CSRC. Por ejemplo, en una conferencia de audio donde un **mixer** señala todos los participantes que han hablado, cuya locución se mezcló para producir un paquete de salida, permitiendo al receptor indicar la persona que está hablando, incluso si todos los paquetes de audio contienen el mismo SSRC (el del **mixer**).

End system

Consiste en una aplicación que genera el contenido que va a ser enviado en paquetes RTP o bien que consume el contenido de un paquete RTP recibido. Puede actuar como una o más fuentes de sincronización en una sesión RTP, pero normalmente como una.

Mixer

Es un sistema que actúa de intermediario y recibe paquetes RTP de una o más fuentes. Es capaz de cambiar el formato de los datos, combinar paquetes RTP y así crear y enviar a la red paquetes nuevos. Como el *timing* de los paquetes entre distintas fuentes no coinciden, el **mixer** realiza los ajustes de *timing* entre los flujos y genera el suyo propio para el flujo combinado. Así, todos los paquetes originados por un **mixer**, lo tienen como su fuente de sincronización.

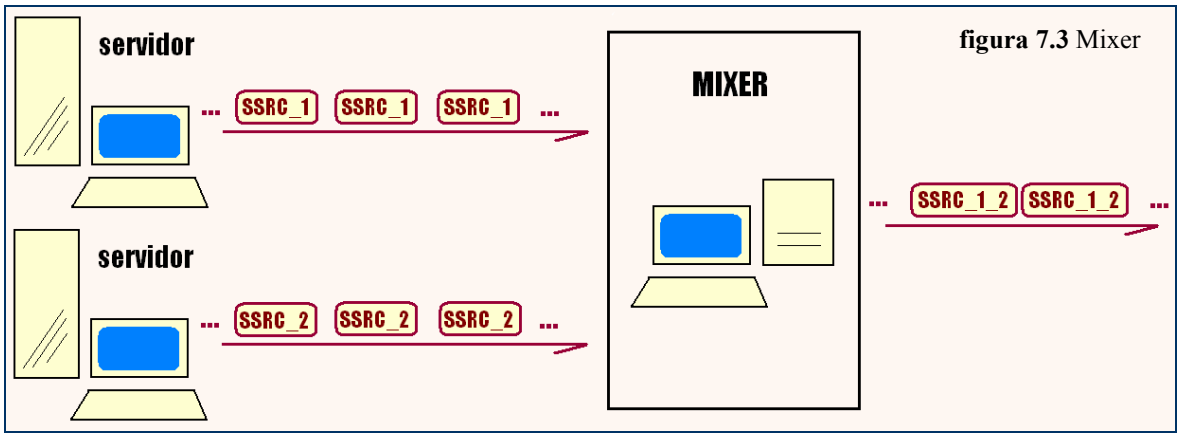


figura 7.3 Mixer

Translator

Es un sistema intermediario que redirige paquetes RTP manteniendo su SSRC intacto. Ejemplos de **translator** pueden ser un dispositivo que convierte la información de un sistema de compresión a otro, **translator** que pasa de multicast a unicast…

Monitor

Una aplicación que recibe paquetes RTCP enviados por los participantes en una sesión RTP, en particular la información sobre la recepción de paquetes, y estima la calidad de servicio para la distribución monitorizada. Un **monitor**, probablemente esté incluido en una aplicación que participa en la sesión, pero también puede ser una aplicación independiente que no participe, y que no envíe ni reciba paquetes de datos RTP. Esos son llamados monitores *third party*.

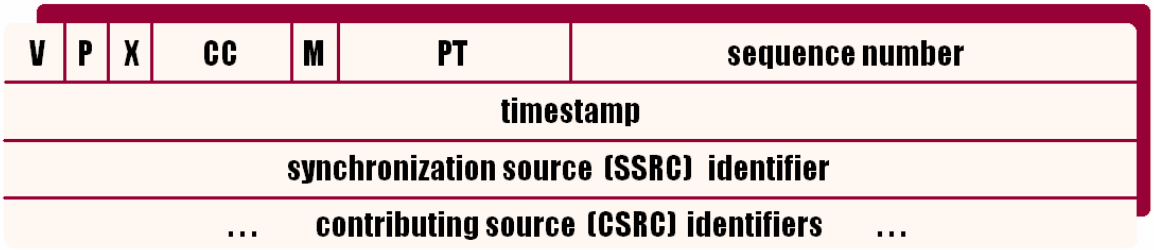
Non-RTP means

Son protocolos y mecanismos que pueden ser necesarios en RTP para proveer un servicio “aprovechable”. En particular, para conferencias multimedia, una aplicación de control para la conferencia puede distribuir direcciones multicast y claves para la encriptación, negociar el algoritmo de encriptación que va a ser usado y definir los formatos de *payload* que no han sido previamente predefinidos. Puede usarse también para aplicaciones simples, como correo electrónico o bases de datos.

7.1.3 RTP Protocolo de Transferencia de Datos

7.1.3.1 Cabecera RTP.

figura 7.4 Cabecera paquete RTP



La lista de los identificadores **CSRC** sólo se encuentra en los paquetes que han pasado por un **mixer**.

version (V): 2 bits. Identifica la versión de RTP. La última versión es la 2.

padding (P): 1 bit. Si se rellena el bit de *padding*, el paquete contiene uno o más octetos al final que no son parte del *payload*. El último octeto de *padding* contiene el número de octetos que deben ser ignorados, incluyéndose a sí mismo. Se usa para paquetes de longitud fija (útil para algunos algoritmos de encriptación).

extension (X): 1 bit. Si se fija este bit, la cabecera debe estar seguida de una cabecera de "extensión". Es una cabecera que contiene información adicional sobre un formato de *payload* que requiera información extra para llevar a cabo sus funciones.

CSRC count (CC): 4 bits. Contiene el número de identificadores CSRC que siguen a la cabecera.

marker (M): 1bit. La interpretación de este bit es definida por un *profile* (perfil). Una cabecera está diseñada para llevar a cabo las funciones más comunes de las aplicaciones RTP, pero la cabecera puede tolerar modificaciones o mejoras definidas en un *profile*. Esto está relacionado también con el campo **extension**, pues la cabecera de "extensión" es la que contiene información sobre el "perfil"

payload type (PT): 7 bits. Este campo identifica el formato del *payload* RTP, y determina su interpretación por la aplicación.

sequence number: 16 bits. El número de secuencia se incrementa en uno por cada paquete RTP enviado y se usa por parte del cliente para detectar pérdida de paquetes y reordenarlos. El valor inicial debe ser aleatorio.

timestamp: 32 bits. Representa el instante en el que se muestrea el primer octeto del paquete RTP de datos. La frecuencia de reloj es dependiente del formato de los datos *payload*. Por ejemplo JPEG, que corresponde a **PT** = 7, según su especificación, su *clock rate* = 90.000 Hz. Es decir, si la aplicación lee bloques que cubren 90.000 periodos de muestreo a partir del dispositivo de entrada (una cámara…), el **timestamp** debe ser incrementado de 90.000 en 90.000. El valor inicial del **timestamp** debería ser aleatorio.

SSRC: 32 bits. Identifica la fuente de sincronización. Este identificador debería de ser elegido de forma aleatoria, con la intención de que dos fuentes de sincronización diferentes con la misma sesión tengan el mismo SSRC. Todas las implementaciones de RTP deben estar preparadas para detectar y arreglar posibles colisiones, una colisión se

produce cuando dos fuentes de sincronización tienen un mismo SSRC. En ese caso, se cambia el SSRC.

CSRC list: de 0 a 15 CSRCs, 32 bits cada uno. Identifica las fuentes que han contribuido al *payload* contenido en el paquete RTP. El número de identificadores lo da el campo CC. Si han contribuido más de 15 fuentes, tan solo son identificadas 15. Recordemos que los identificadores CSRC son insertados por los **mixers** (ver apartado 7.1.2).

7.1.4 RTCP Protocolo de control RTP.

7.1.4.1 Formato de paquete RTCP

La especificación define varios tipos de paquetes RTCP que permiten gran variedad de información de control:

SR: *Sender report*, para estadísticas de transmisión y recepción desde participantes que son emisores activos.

RR: *Receiver report*, para estadísticas de recepción desde participantes que no son emisores activos.

SDES: *Source description*, son valores que describen a una fuente, incluye *CNAME*.

BYE: Indica el fin de la participación.

APP: Funciones específicas de la aplicación.

Cada paquete RTCP comienza con una parte fija similar a los paquetes RTP de datos, seguida por una serie de elementos estructurados que pueden ser de longitud variable según el tipo de paquete pero que nunca deben sobrepasar los 32 bits.

Recordemos que varios paquetes RTCP pueden ser concatenados y ser enviados como un único paquete del protocolo de la capa inferior (UDP). Aún así, en destino todos los paquetes RTCP son procesados de forma individual sin requerir ningún orden a la hora de combinar los paquetes. Para potenciar las funciones del protocolo, se imponen las siguientes restricciones:

- Las estadísticas de recepción (en **SR** o **RR**), deberían ser enviadas a menudo según lo permita el ancho de banda de la red, para optimizar los valores de las estadísticas. Por lo tanto, cada paquete RTCP compuesto debe incluir un paquete SR o RR.
- Los nuevos receptores necesitan recibir el *CNAME* de una fuente tan pronto como sea posible para identificar la fuente y empezar a asociar los flujos multimedia. Cada paquete RTCP debe contener el *SDES CNAME*.

Así, todos los paquetes RTCP deben ser enviados en un paquete compuesto de por lo menos dos paquetes individuales, con el siguiente formato recomendado:

- **Prefijo de encriptación:** solamente si el paquete compuesto ha sido encriptado
- **SR o RR:** El primer paquete RTCP en el paquete compuesto debe ser siempre un paquete de *report* (informe) para facilitar la validación de cabeceras (existe un código que al ejecutarse indica si una cabecera es válida o no). Esto es así

incluso si no se han enviado o recibido datos, en cuyo caso debe ser enviado un RR vacío, e incluso si el otro paquete RTP en el paquete compuesto es *BYE*.

- **Additional RRs:** Si el número de fuentes cuyas estadísticas de recepción excede de 31 (el máximo que permite un paquete RTCP SR o RR), entonces los paquetes RR deberían ser apilados tras el paquete de *report* inicial.
- **SDES:** Un paquete SDES contiene un CNAME que debe ser incluido en cada paquete RTCP compuesto (una fuente se diferencia de otra por el CNAME, pero una fuente puede establecer distintas sesiones). Otros valores sobre la descripción de una fuente (como NAME, PHONE, EMAIL, LOC, TOOL, NOTE y PRIV) pueden añadirse también si son requeridos por la aplicación, pero esto consume ancho de banda.
- **BYE o APP:** Otros tipos de paquetes, incluyendo algunos aún no definidos. Pueden estar en cualquier orden, excepto que *BYE* debería ser el último paquete enviado.

Una implementación debería ignorar paquetes RTCP de tipo desconocido.

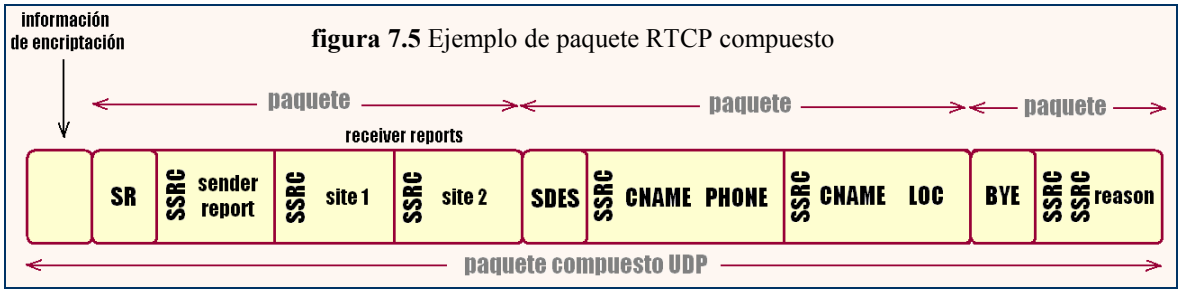


figura 7.5 Ejemplo de paquete RTCP compuesto

La frecuencia con que se envían los paquetes RTCP depende del ancho de banda de la red, es el usuario el que especifica el ancho de banda.

7.1.4.2 Sender y Receiver Reports

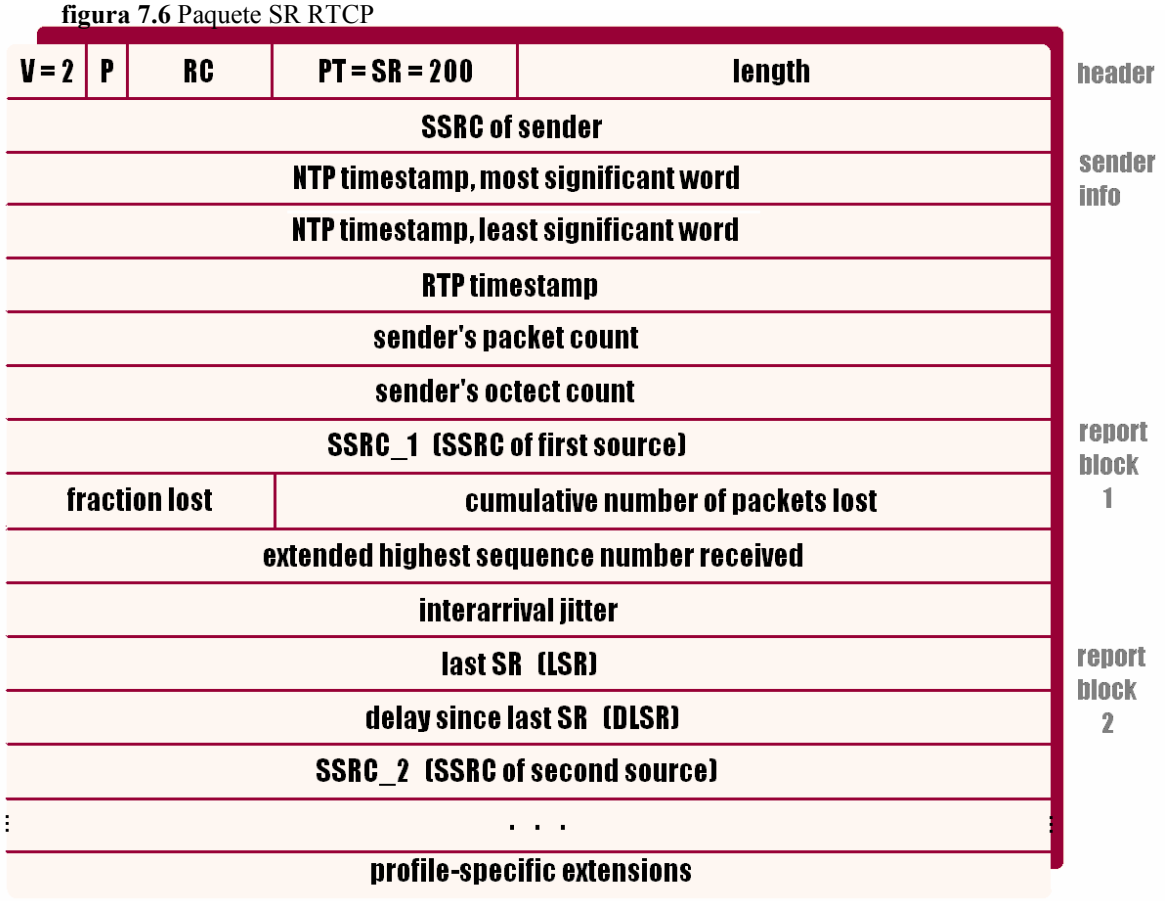
Sender y Receiver Reports se encargan de proveer “calidad” a la comunicación, informando mediante estadísticas. La diferencia entre el SR y RR, se encuentra en el campo de tipo de paquete y poco más.

Tanto SR como RR, incluyen cero o más bloques de recepción de *reports* (informes), uno para cada SSRC. Cada bloque provee estadísticas sobre los datos recibidos de una fuente en particular que se indica en ese bloque. Caben hasta un total de 31 bloques de recepción de *reports* en un paquete SR o RR.

En los próximos apartados se definen los formatos de los dos *reports*.

7.1.4.2.1 SR: paquete Sender Report RTCP

figura 7.6 Paquete SR RTCP



El paquete SR consta de tres secciones. La primera sección, la cabecera ocupa 8 octetos. Los campos que tiene son los siguientes:

version (V): 2 bits. Identifica la versión de RTP, que es la misma en los paquetes RTCP y RTP. La última versión es la 2.

padding (P): 1 bit. Si se rellena el bit de *padding*, este paquete RTCP contiene algunos octetos adicionales de *padding* que no son parte de la información de control, pero se incluyen en el campo **length**. El último octeto de *padding* es un número que indica cuantos octetos deberían ser ignorados, incluyéndose él mismo. En un paquete RTCP compuesto, sólo se requiere un paquete con bits de *padding*, que debe ser el último.

reception report count (RC): 5 bits. Indica el número de bloques de recepción de *report* contenidos en este paquete. El valor cero es válido.

packet type (PT): 8 bits. Contiene la constante 200 que identifica a este paquete como un paquete RTCP SR.

length: 16 bits. Indica la longitud del paquete RTCP incluyendo la cabecera y el *padding*.

SSRC: 32 bits. El identificador de la fuente de sincronización que ha originado este paquete SR.

La segunda sección, la información sobre el emisor, tiene una longitud de 20 octetos y se encuentra presente en cada paquete SR. Da información sobre la transmisión de datos del emisor. Los campos son los siguientes:

NTP timestamp: 64 bits. Indica el *Wallclock time*, que es la hora y la fecha absoluta. Se representa usando el formato NTP (*Network Time Protocol*), que es en segundos desde las 0 horas del 1 de Enero del año 1900. Un servidor que no tenga noción de tiempo puede fijar el valor de este campo a 0. Cuando se envía este *report* puede usarse junto a los *timestamps* devueltos en los *reports* de recepción de otros receptores para medir el tiempo de propagación a esos receptores.

RTP timestamp: 32 bits. Corresponde al mismo tiempo que el anterior *NTP timestamp*, pero en las mismas unidades y con el mismo *offset* aleatorio que en los *RTP timestamps* de los paquetes de datos. Esta correspondencia puede usarse para una sincronización intermedia para fuentes cuyos *NTP timestamps* están sincronizados, y pueden ser usados por receptores para estimar la frecuencia del reloj RTP.

sender's packet count: 32 bits. El número total de paquetes RTP de datos transmitidos por el emisor desde que se inició la transmisión hasta el momento en el que se genera el paquete SR. El valor debería ser inicializado si el emisor cambia su identificador SSRC.

sender's octect count: 32 bits. Indica el número total de octetos de *payload* transmitidos en paquetes RTP de datos por el emisor desde que empezó la transmisión hasta que el paquete SR se genera. No incluye cabecera ni *padding*. El valor debería ser inicializado si el emisor cambia su identificador SSRC.

La tercera sección contiene de ninguno a varios bloques de recepción de *reports* dependiendo del número de fuentes escuchadas por este emisor desde el último *report*. Cada bloque de recepción transporta estadísticas sobre la recepción de paquetes RTP procedentes de una fuente de sincronización. Esas estadísticas son:

SSRC_n (source identifier): 32 bits. El identificador SSRC de la fuente a la que pertenece la información contenida en este bloque de recepción de *report*.

fraction lost: 8 bits. La fracción de paquetes RTP de datos perdidos, procedentes de la fuente SSRC_n desde que se envió el paquete SR o RR previo. Esta fracción es el número de paquetes perdidos dividido entre el número de paquetes esperados.

cumulative number of packets lost: 24 bits. Es el número total de paquetes RTP de datos procedentes de la fuente SSRC_n que se han perdido desde el inicio de la recepción. Se define como el número de paquetes esperados, menos el número de paquetes recibidos actuales, donde este último incluye paquetes que llegan tarde y duplicados. El número de paquetes esperados se define como el último número de secuencia recibido menos el número de secuencia inicial que se recibió.

extended highest sequence number received: 32 bits. Los primeros 16 bits (*big endian*) incluyen el número de secuencia más alto recibido en un paquete de datos RTP desde la fuente SSRC_n, y los 16 bits más significantes amplían ese número de secuencia con la cuenta correspondiente al número de ciclos de la secuencia.

interarrival jitter: 32 bits. Es una estimación estadística de la varianza del tiempo de llegada de paquetes RTP de datos, medido en unidades de *timestamp* y expresado como un entero sin signo.

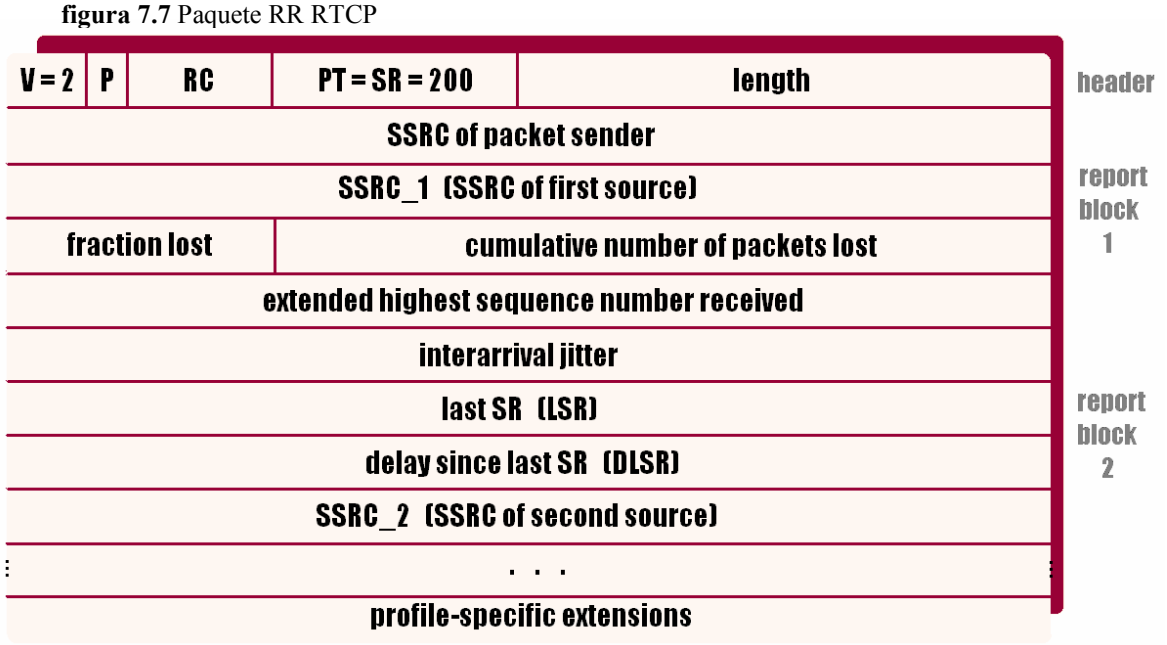
last SR timestamp (LSR): 32 bits. *Timestamp* que forma parte del paquete SR más reciente de la fuente SSRC_n (se obtienen de los 32 bits "centrales" de la NTP *timestamp*, para conseguir así una representación más sencilla). Si SR no se ha recibido aún, este campo vale cero.

delay since last SR (DSLR): 32 bits. Es el retardo expresado en unidades de 1/65536 segundos, entre la recepción del último paquete SR procedente de la fuente SSRC_n y el envío de este bloque de recepción de *report*.

profile-specific extensions: Información relacionada con la "cabecera de extensión" y perfiles, sobre los que se habló en el campo **extension** y **marker** de la cabecera del paquete RTP.

7.1.4.2.2 RR: paquete Receiver report RTCP

figura 7.7 Paquete RR RTCP



El formato del paquete RR es el mismo que el del paquete SR, exceptuando que el campo de tipo **packet type** contiene el valor constante 201, y la información del emisor se omite (**NTP** y **RTP timestamps** y **sender's packet** y **octect count**).

7.1.4.2.3 Utilidad de sender y receiver reports

Se utilizan para dar una "calidad" a la comunicación. El servidor puede modificar sus transmisiones según los *reports*. Los receptores pueden determinar si los problemas son locales, regionales o globales. Es posible utilizar en la red monitores que solamente reciban los paquetes RTCP para evaluar el rendimiento de la red, por ejemplo en una distribución multicast.

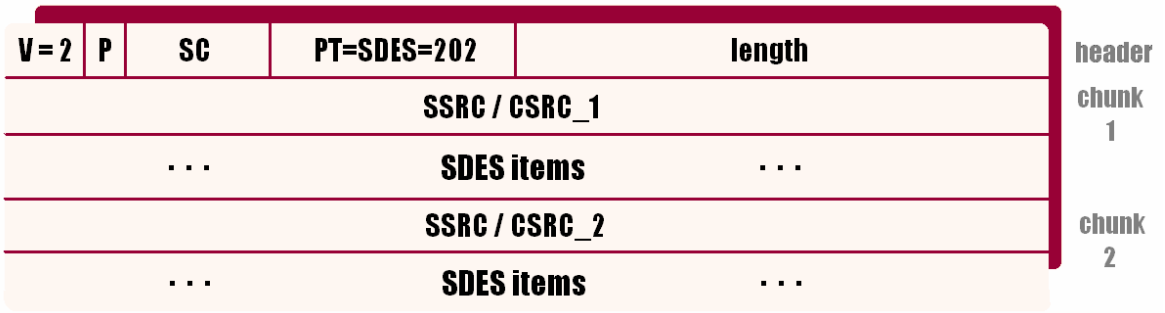
El uso acumulativo de tiempos permite que entre dos *reports* consecutivos podamos estimar constantemente la calidad de la distribución. Valores como el número de

paquetes procedentes del servidor nos permiten conocer cuántas pérdidas se han dado en la red.

El campo que nos informa sobre el *jitter*, provee una medida acerca de la congestión de la red. Mientras la pérdida de paquetes informa sobre una congestión persistente, el *jitter* informa de la congestión transitoria en la red. Además, a partir de este valor, puede implementarse un *buffer* de recepción que consiga reestablecer la tasa de generación de frames original del emisor.

7.1.4.3 SDES: Source Description RTCP

figura 7.8 Paquete SDES RTCP



El paquete SDES es una estructura de tres niveles compuesta por una cabecera y ninguno o más *chunks* (fragmentos). Los cuales están compuestos por *ítems* que describen a la fuente (ofrecen información adicional sobre ella) que identifica ese *chunk*. Los *ítems* se describen (dentro del paquete) de forma individual en secciones consecutivas.

versión (V), padding(P), lenght: Son como los descritos en el paquete SR (apartado 7.1.4.2.1).

packet type (PT): 8 bits. Contiene el valor de la constante 202 que identifica lo identifica como un paquete RTCP SDES.

source count (SC): 5 bits. El número de los *chunks* SSRC/CSRC que contiene este paquete SDES. Puede valer cero.

Cada *chunk* consiste en un identificador SSRC/CSRC seguido por ninguno o varios *ítems*, que llevan información sobre SSRC/CSRC.

Los *ítems* SDES definidos se describen en los siguientes apartados. Sólo el *item* CNAME es obligatorio.

7.1.4.3.1 CNAME: Canonical identifier SDES item

figura 7.9 Item CNAME RTCP

Es obligatorio, el emisor siempre lo envía. Sus propiedades son:

- Como el identificador SSRC puede cambiar, el CNAME debe usarse para identificar a un participante en una sesión RTP
- CNAME debe ser único
- CNAME debe ser fijo para cada participante (no variar)
- El formato puede ser de tipo "*usuario@host*" o "*host*"

7.1.4.3.2 NAME: User name SDES item

figura 7.10 Item NAME RTCP

7.1.4.3.3 EMAIL: Electronic mail address SDES item

figura 7.11 Item EMAIL RTCP

7.1.4.3.4 PHONE: Phone number SDES item

figura 7.12 Item PHONE RTCP

7.1.4.3.5 LOC: Geographic user location SDES item

figura 7.13 Item LOC RTCP

7.1.4.3.6 TOOL: Application/Tool name SDES item

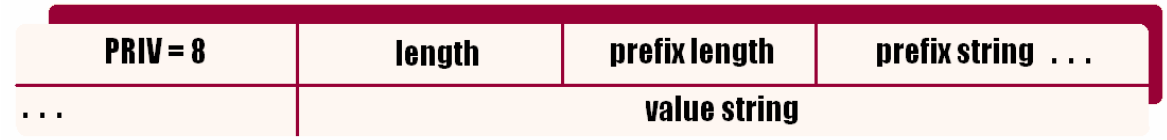
figura 7.14 Item TOOL RTCP

7.1.4.3.7 NOTE: Notice/Status SDES item

figura 7.15 Item NOTE RTCP

7.1.4.3.8 PRIV: Private extensions SDES item

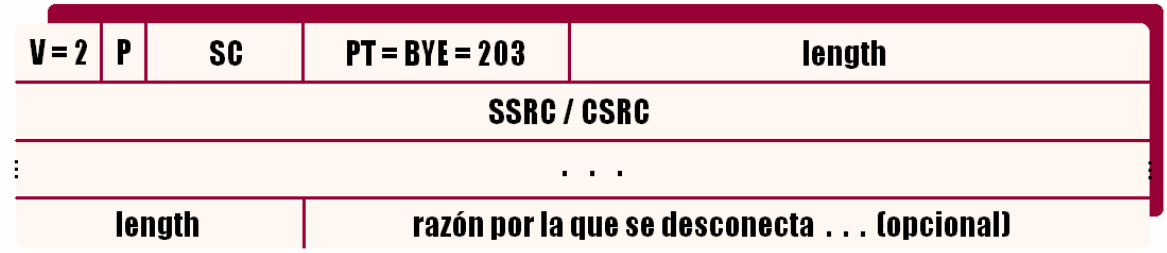
figura 7.16 Item PRIV RTCP



Permite utilizar un *item* con valores personales (privados) según la necesidad. Constan de un prefijo, y su valor.

7.1.4.4 BYE: Paquete Goodbye RTCP

figura 7.17 Paquete BYE RTCP



El paquete BYE indica que una o más fuentes no siguen activas.

versión (V), padding(P), lenght: Son como los descritos en el paquete SR (apartado 7.1.4.2.1).

packet type (PT): 8 bits. Contiene el valor de la constante 202 que identifica lo identifica como un paquete RTCP SDES.

source count (SC): 5 bits. El número identificadores SSRC/CSRC incluidas en este paquete BYE. Puede valer cero.

7.2 Introducción a la creación del controlador RTP

Una vez estudiada la implementación del controlador UDP, se tienen unas nociones básicas sobre la transmisión de video. La necesidad de un control sobre la información se hace imprescindible para ofrecer una determinada calidad de servicio.

El protocolo RTP, apoyándose sobre UDP ofrece gran funcionalidad. El control sobre el vídeo que se llevaba a cabo en UDP ya está implementado (de una forma mucho más completa y versátil) en las librerías RTP, así podemos limitarnos a llamar funciones ya implementadas que gestionen todos los mensajes de control sobre el video. La funcionalidad interna de RTP se encarga de controlar situaciones como pueden ser la desconexión o "caída" de clientes, medir retardos, paquetes perdidos o desordenados… mediante el protocolo RTCP, que trabaja de manera conjunta a RTP. RTCP nos ofrece toda esa información a través de sus paquetes de control, y es el programador el que decide qué hacer una vez se reciben. Así, si para un cliente determinado el video ocupa demasiado ancho de banda, se puede dar un mayor nivel de compresión, o utilizar otro formato de video. Esta última parte le corresponde al programador implementarla, aunque es cierto que existen librerías no gratuitas que pueden hacerlo.

xawtv está programado en C, y por ello era necesario la utilización de unas librerías RTP implementadas en C (aunque C++ también es válido, véase el ejemplo de AVStreams) y gratuitas. Las librerías gratuitas suelen implementar la especificación RTP de forma parcial, y tal vez contengan variaciones, o tengan multitud de *bugs*. Existen también implementaciones de pago.

El procedimiento que se llevó a cabo fue similar al de la implementación del controlador UDP. Se realizó una conexión punto a punto entre cliente y servidor empleando un formato de compresión que no consumía gran ancho de banda y que no necesitara fragmentación. RTP no fragmenta paquetes.

Más tarde, se consiguió con éxito que un servidor atendiera peticiones de varios clientes y que pudieran ser desconectados con éxito. Fue complejo conseguir que el servidor se mantuviera estable, pues se tuvieron muchos problemas con la librería RTP utilizada (al parecer la única gratuita disponible en lenguaje C).

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

7.2.1 Compresión y fragmentación

Las librerías RTP utilizadas, por defecto definían un tamaño de paquete de 1500 *bytes*, teniendo en cuenta que el tamaño máximo de cabecera RTP está fijado a 20 *bytes*. El tamaño de un fragmento IP es de 1480 *bytes* (IP fragmenta paquetes en la red).

El tamaño máximo de paquete RTP se redefinió con un tamaño de 20.000 *bytes*.

Recordemos que el tamaño de cada *frame* MJPEG oscila entre 8 y 15 *Kbytes*, así, el tamaño de 20.000 es adecuado para no realizar fragmentación. Únicamente se ha implementado el formato MJPEG.

Otros formatos como RGB24 también pueden implementarse fragmentando previamente, aunque tal vez no sean formatos de compresión (RGB24, RGB15…) adecuados para la transmisión de video mediante RTP.

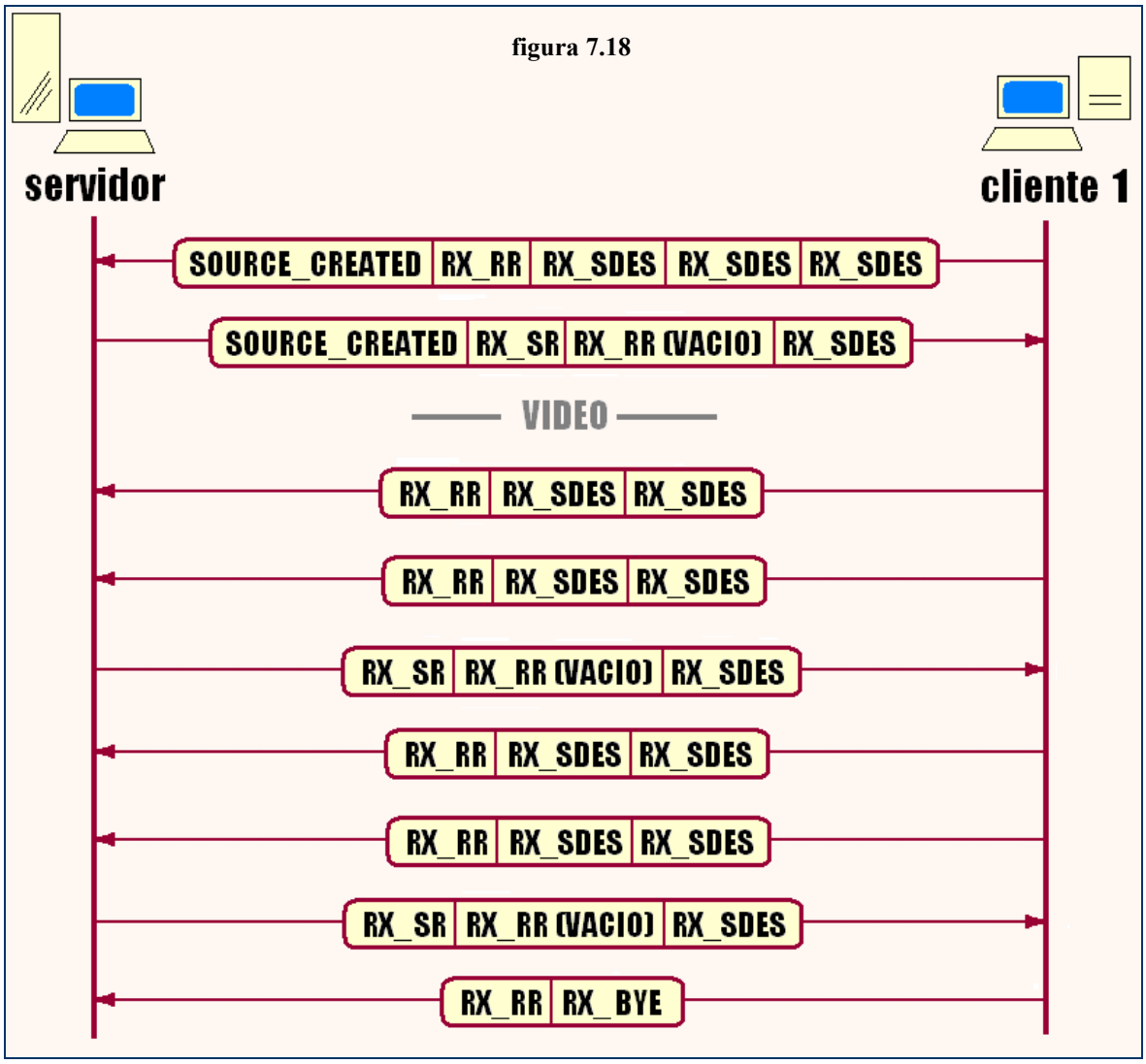
Se sabe que RTP es el protocolo para la transmisión de video por *internet*, donde se utilizan formatos de compresión ideados para ello. El video que se envía por internet:

- debe estar comprimido en gran proporción para consumir poco ancho de banda
- debe ser *streaming video*: son flujos multimedia que el cliente almacena en un *buffer* (que debe ser pequeño en aplicaciones a tiempo real) y los reproduce. Además al ser un flujo, el concepto de "fragmentación de imágenes" ya no tiene sentido, es un flujo de datos de video que se envían en paquetes. Formatos como RGB24 o MJPEG no proporcionan "un flujo" de video, sino que ofrecen imágenes comprimidas de forma totalmente independiente. Formatos como *rm* (*realmedia*) [19] o *asf* (*Advanced Streaming Format*, de Microsoft) [20] son más adecuados, pero no de libre distribución.

7.2.2 Intercambio de información de control RTCP. Eventos

Es importante conocer el sistema de eventos empleado en las librerías utilizadas antes de seguir avanzando en esta introducción a la creación del controlador RTP.

figura 7.18



En la figura 7.18 se muestra un ejemplo de ejecución en el que se establece una conexión cliente-servidor, se "intercambian" paquetes de control, (omitiéndose el envío de video) y se cierra. Aunque podría parecer que se trata de una secuencia de paquetes que se envían cliente y servidor, en realidad la implementación RTP utilizada habla de **eventos**. Así, la lectura de la figura sería de la forma:

1. **Cliente 1** genera los eventos de recepción de **SOURCE_CREATED** (se ha creado la fuente), **RX_RR** (recepción de *Receiver Report*), (recepción de *SDES*) **RX_SDES** y **RX_SDES** en el servidor.
2. **Servidor** genera los eventos de recepción **SOURCE_CREATED**, **RX_SR** (recepción de *Sender Report*)**, RX_RR** (vacío) y **RX_SDES** en el cliente.
3. Una vez se ha establecido la comunicación entre ambos, se produce el envío de vídeo y se generan eventos de recepción de mensajes de control en los dos extremos.
4. Finalmente el cliente genera los eventos **RX_RR** y **RX_BYE** (recepción del paquete **BYE**) en el servidor.

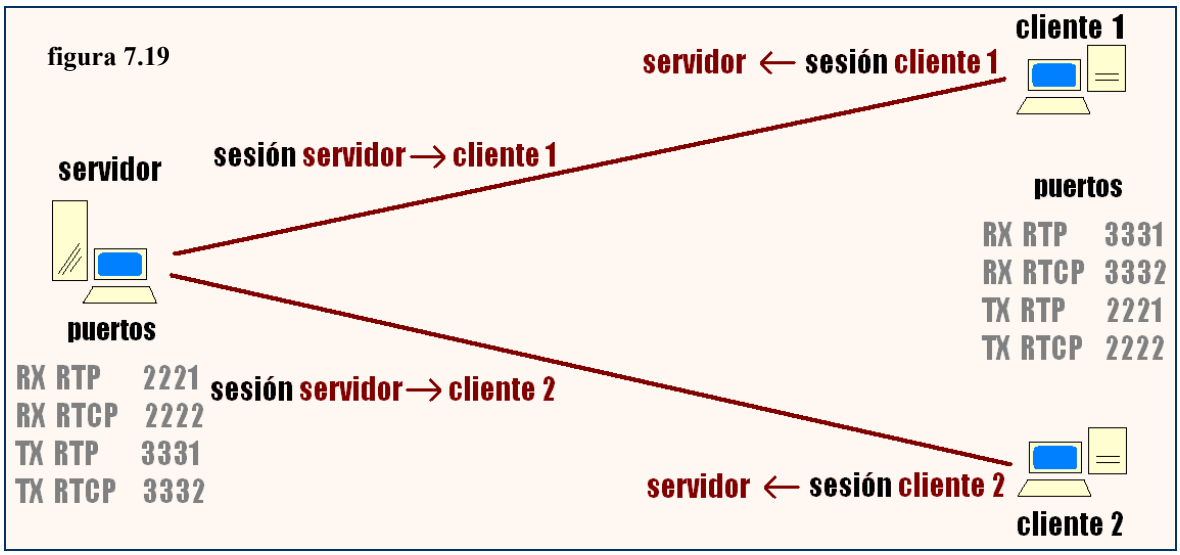
Cada evento representa la llegada de un paquete RTCP, por lo que en la figura se muestran paquetes compuestos RTCP (ver apartado 7.1.4.1). Por ejemplo, **RX_RR** correspondería a un paquete de control RTCP de tipo RR, **RX_SDES** a un paquete de tipo SDES…

Como se indica en la especificación de RTP, los paquetes de *report* se envían continuamente, así como los de tipo SDES (en concreto, hay una obligación de mandar siempre el *item* CNAME).

Cuando un evento de recepción de paquetes de control se genera, es el programador el que decide qué hacer con la información que contiene cada paquete.

7.2.3 Un servidor, varios clientes. Problemas de implementación

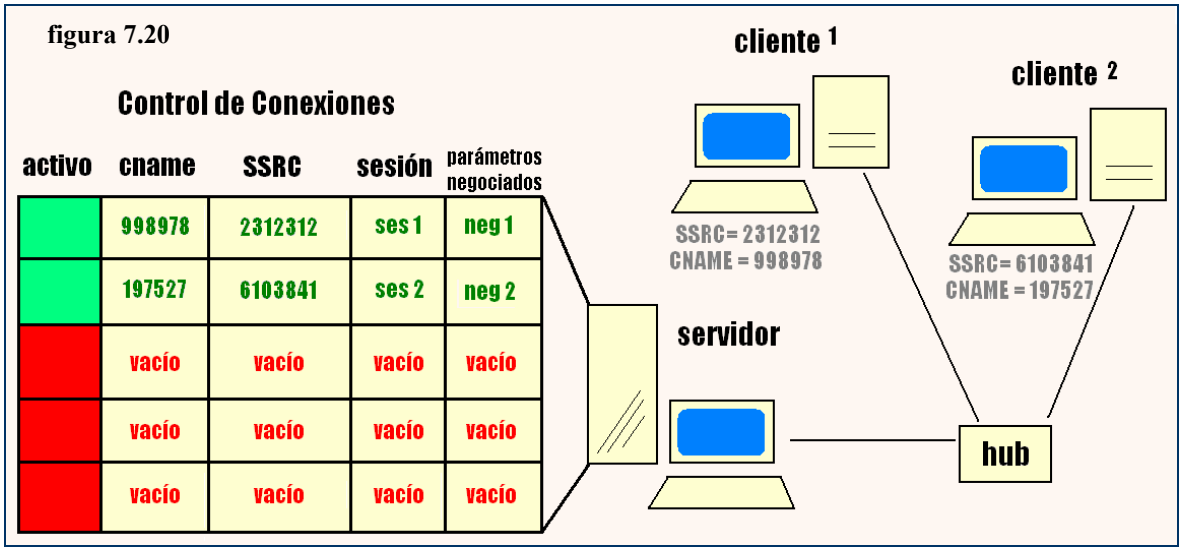
La comunicación entre cliente y servidor se establece mediante la creación de una sesión. En las librerías que se han utilizado, el cliente crea una sesión con el servidor, y el servidor una con cada cliente, como se muestra en la figura 7.19.



Además, el servidor tiene un puerto por el que espera recibir datos RTP y otro por el que espera datos de control RTCP (en la figura serían 2221 y 2222 respectivamente). Cuando el servidor tiene que enviar información RTP o RTCP, sabe a qué puertos de las máquinas cliente tiene que hacerlo (en la figura serían 3331 y 3332). Los clientes tienen que tener estos dos últimos puertos como puertos de recepción, y cuando deseen enviar, deben hacerlo a los puertos de recepción del servidor.

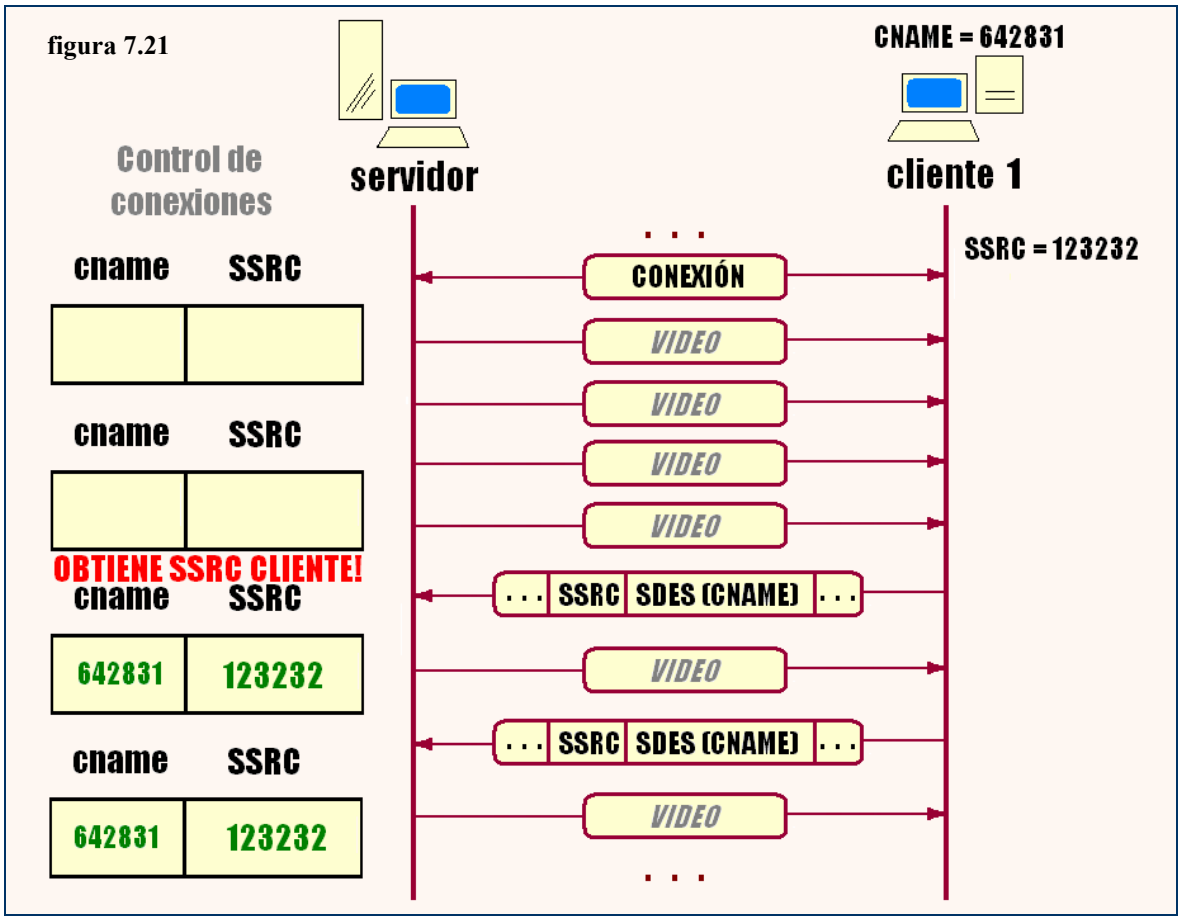
***Problemas de implementación**: inicialización de sesión y cambio de SSRC en el cliente*

La implementación de un servidor que sirviera a varios clientes no fue sencilla. Se trabajaba con una implementación de librerías RTP gratuita, que al parecer no funcionaba adecuadamente. El servidor tiene asignado un SSRC desde el momento en que crea la sesión con el cliente (en realidad es la sesión la que tiene asignada el SSRC), y cada cliente posee el suyo (al crear una sesión con el servidor). El SSRC del servidor no se ha utilizado: son los SSRC de los clientes los que el servidor debe guardar en una tabla para saber quién ha generado un evento de recepción de paquetes de control (guarda también otros valores, como puede ser CNAME asociado al cliente…). Ver figura 7.20



El servidor no conoce el SSRC del cliente hasta que no le llega el primer paquete de control RTCP, pero evidentemente ya ha creado una sesión con él y lo ha registrado en su tabla. Cuando le llega el primer paquete de control informándole del SSRC del cliente (generando el evento *SOURCE_CREATED* en el servidor), comprueba la IP del cliente (en concreto, de su sesión) que genera el evento y recorre la tabla para localizar el cliente que posee dicha IP y guarda su SSRC (figura 7.21). El servidor puede estar enviándole video antes de recibir el paquete que informa del SSRC cliente (para enviar video sólo se necesita la sesión). Si el cliente se desconectara en ese instante, podría haber problemas. La solución se explica en el apartado 7.2.4.

figura 7.21



Los motivos por los que se conserva el SSRC de cada cliente en la tabla de conexiones del servidor son:

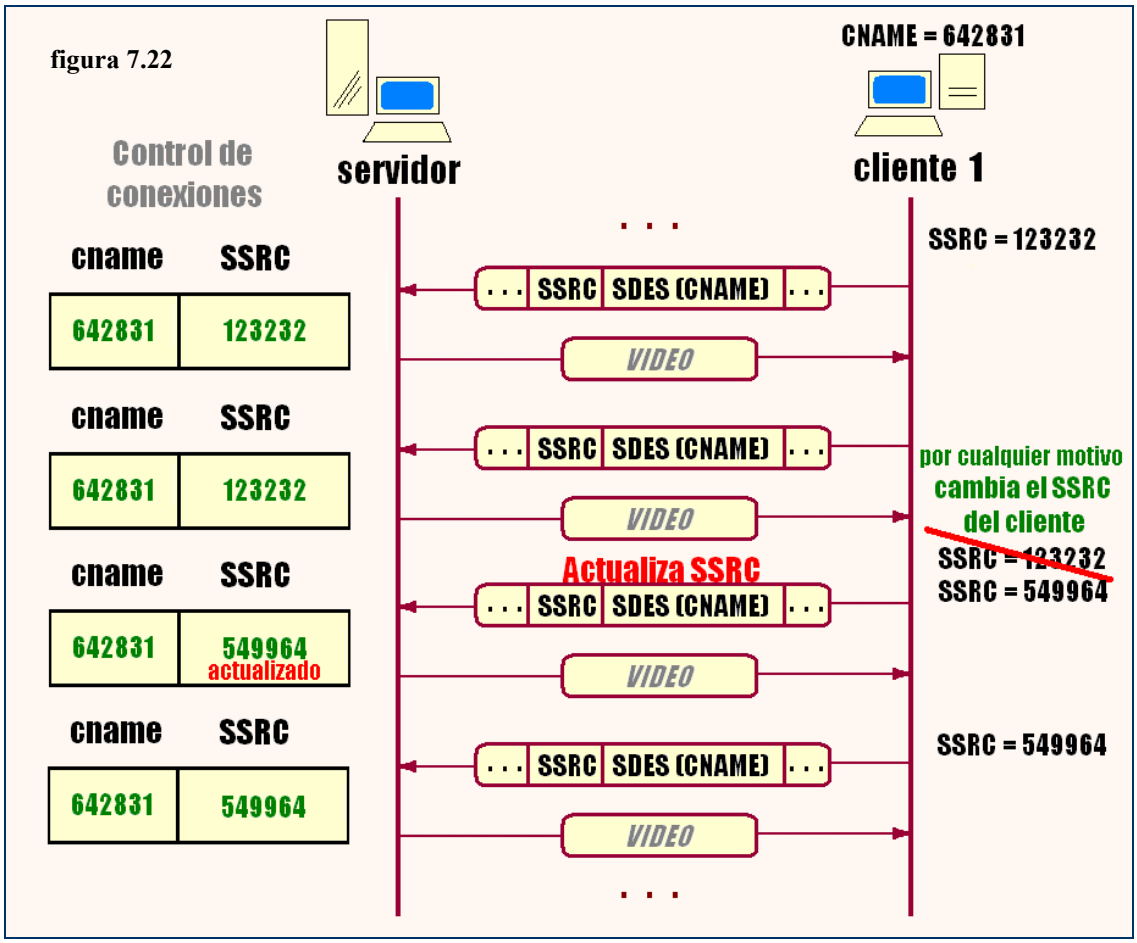
- Una sesión se identifica por su SSRC
- Las funciones implementadas en las librerías RTP necesitan un SSRC
- Debido a motivos desconocidos, en algunos casos, el servidor no obtiene la dirección IP correcta del cliente que ha generado un evento (ver apartado anterior). Por ello se sabe quién ha generado un evento de recepción, no a partir de una dirección IP, sino a partir del SSRC del cliente que ha generado el evento. Existe una caso en el que la dirección IP del cliente se obtiene de forma correcta, y es el caso explicado en la página anterior: cuando llega el primer paquete de control RTCP que genera el evento *SOURCE_CREATED*

En principio, el SSRC (que recordemos es generado de manera aleatoria) sólo cambia si existe una colisión (varios participantes con un mismo SSRC, lo que es realmente difícil), pero la implementación utilizada cambia el SSRC de un cliente sin motivo aparente cada cierto tiempo cuando hay conexiones y desconexiones de otros clientes.

Ahora sabemos que ni una dirección IP (pues no se obtiene de forma correcta en algunos casos) ni el SSRC (que cambia y el servidor no es consciente de ello) nos permiten identificar a un cliente (el SSRC identifica a una sesión) y en concreto conocer la procedencia de los mensajes de control.

Existe una posibilidad que permite que el servidor “se de cuenta” del cambio del SSRC cliente. Puede usarse el nombre canónico CNAME que es exclusivo para cada cliente y que lo define el programador. En este caso se ha optado por un número aleatorio.

figura 7.22



Los mensajes de control constantemente están informando sobre el CNAME del cliente, cuando se recibe un paquete que contiene información sobre el *item* CNAME, se puede obtener el SSRC del cliente que ha generado el evento de recepción de este *item*. Ese es el momento en el que se compara este CNAME con los que tiene almacenados el servidor en su tabla de clientes, una vez lo encuentra, actualiza el valor de SSRC de dicho cliente si este ha cambiado. Así se consigue que aunque el SSRC varíe, el servidor tenga constancia de ello y lo mantenga actualizado en su tabla. Ver figura 7.22

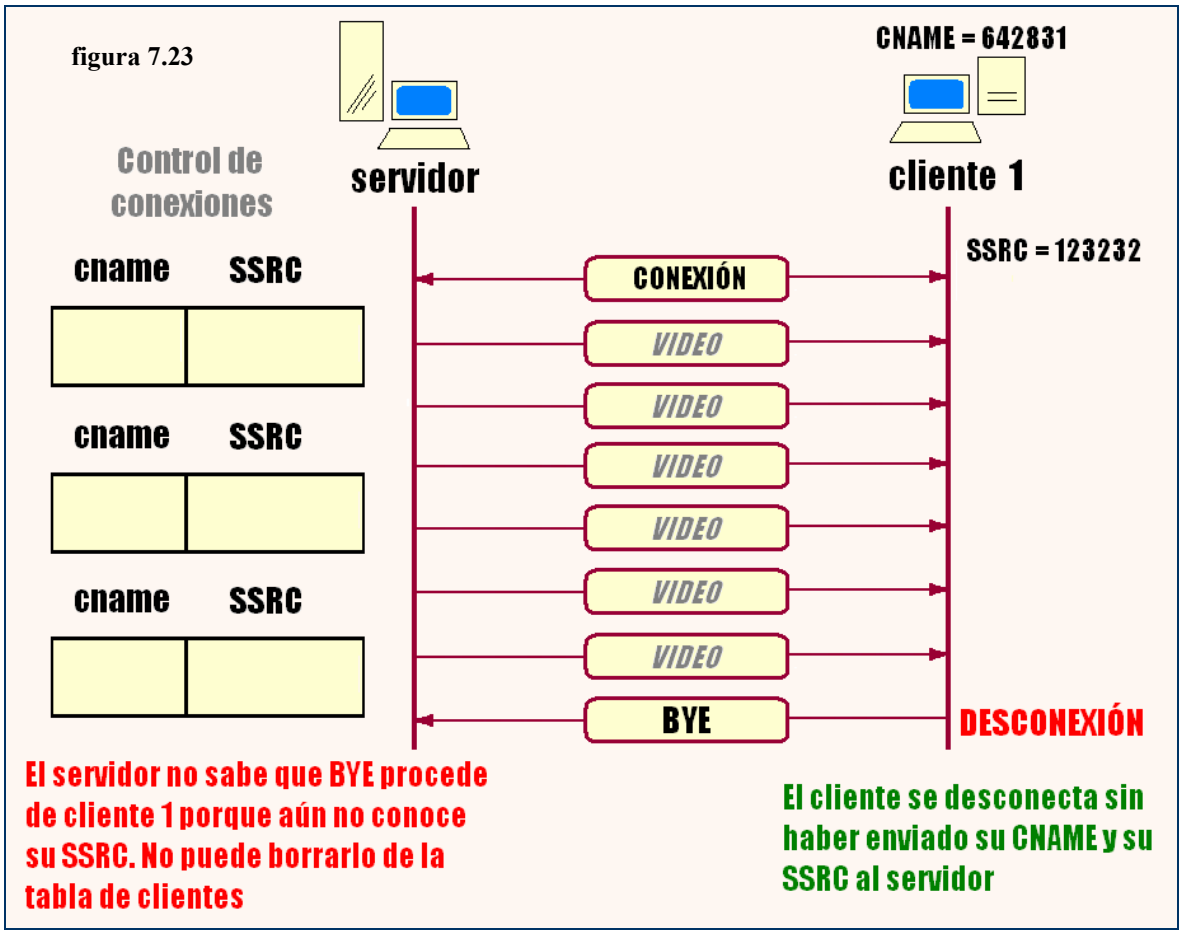
Para enviar video y mensajes de control, tan solo se necesita la sesión que crea el servidor con el cliente y no su SSRC, pero cuando se reciben mensajes de control, sólo se puede detectar de quién proceden mediante su SSRC, de ahí que sea tan importante.

Por otra parte, para evitar problemas si dos clientes se conectan al mismo tiempo, recordemos que TCP sólo aceptaba conexiones cada 6 segundos. Previene problemas relacionados con la obtención del SSRC cliente por parte del servidor si dos o más clientes conectan simultáneamente.

7.2.4 Desconexión. Problemas de implementación

Un cliente y un servidor están intercambiando continuamente paquetes de control RTCP. Cuando un cliente se desconecta, envía el paquete **BYE** al servidor. El servidor finaliza la sesión que creó con ese cliente y lo borra de su tabla interna de conexiones.

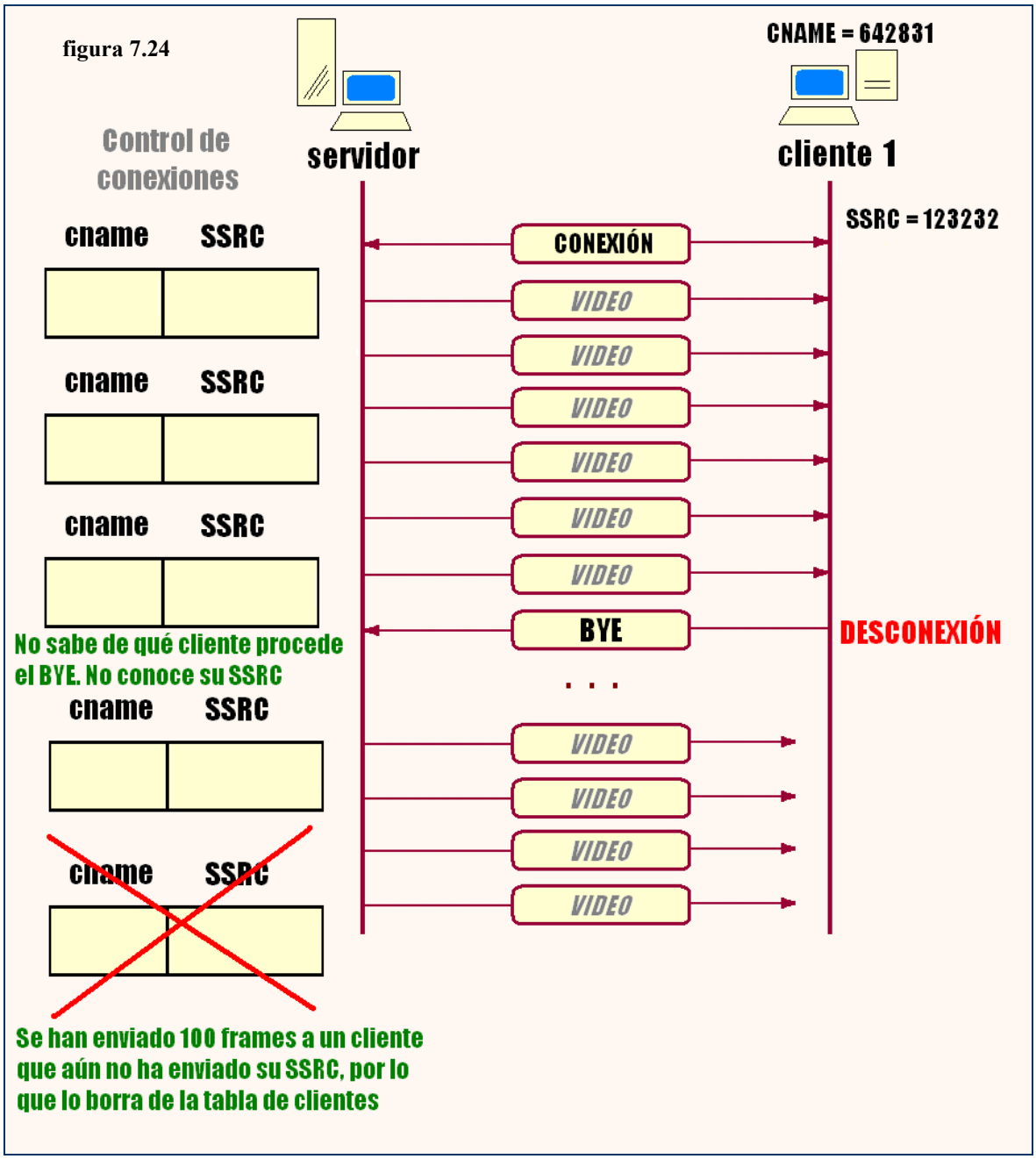
Ese sería el procedimiento normal de desconexión, pero también es posible que un cliente se desconecte y el servidor no reciba el paquete **BYE**. Puede ocurrir por varios motivos, se ha perdido el paquete, el cliente no lo manda porque ha sufrido una desconexión accidentada… Las librerías RTP que se han usado permiten el uso de un TIMEOUT, si el servidor no recibe paquetes de control de un determinado cliente durante un determinado espacio de tiempo, puede desconectarlo. Incluso es muy útil para el caso que se describe en el apartado anterior, aquel en el que el servidor envia video a un cliente sin conocer aún su SSRC. Si el cliente desconectaba antes de que el servidor lo conociera, ¿cómo iba a detectar la procedencia del paquete **BYE**, si la detección del origen de los paquetes de control se hace mediante el SSRC del cliente que generó el evento? Figura 7.23



Parece que la función de TIMEOUT es válida para salvar todos los problemas de desconexión. El problema es que aún, sin saber el motivo, saltan *timeouts* de todos los clientes cada cierto tiempo aunque la conexión esté activa. Por ello se ha desactivado la opción de TIMEOUT, y se ha optado por una solución propia.

Es cierto que si un paquete **BYE** se pierde, poco se puede hacer al no disponer de TIMEOUT, pero en condiciones normales no tiene por qué perderse. El otro problema (el

de la desconexión de un cliente sin conocer su SSRC) tiene fácil solución, si al cabo de haber enviado un número determinado de frames aún no se ha obtenido el SSRC cliente, el servidor lo desconecta. Ver figura 7.24.



7.3. Implementación del controlador RTP

7.3.1 Librerías RTP

Se han utilizado unas librerías RTP de libre distribución. En ocasiones, las implementaciones en código libre suelen ser implementaciones parciales. A esto hay que añadir que una implementación debe ceñirse a la especificación, pero en ocasiones aporta nuevas funcionalidades o variar algunas.

En la página web http://www.cs.columbia.edu/~hgs/rtp/ [20] se encuentra una relación de implementaciones del protocolo RTP disponibles, desde gratuitas hasta de pago, y para varios lenguajes de programación.

Para lenguaje C, se han encontrado dos librerías:

- http://srtp.sourceforge.net/news.html - Librerías para *Secure RTP* (SRTP). El autor actualiza las librerías cada poco tiempo, pero la implementación de protocolo RTP según la especificación aún está en sus primeros pasos.
- http://www-mice.cs.ucl.ac.uk/multimedia/software/common/index.html - Estas últimas son las que se han utilizado. Se ha usado la versión 1.2.8 con fecha 1 de Mayo de 2001. En los últimos días (10 de Mayo de 2003) se ha lanzado una nueva versión 1.2.14 que no se ha podido probar, ni se conocen las posibles mejoras.

En realidad, este último paquete [21] de librerías incluye implementaciones de distintos protocolos y algoritmos:

- Base64 codificación/decodificación
- Árboles binarios
- Números aleatorios
- Autentificación HMAC
- MD5
- DES
- **RTP**
- MBus
- SAP
- SDP

Las librerías descargadas se descomprimen con la instrucción

```
tar xzvf common-1.2.8.tar.gz
```

Como es habitual en linux, para compilar

```
./configure
make
```

Dependiendo del compilador instalado y de la versión descargada, pueden aparecer algunos *warnings* en la compilación. Las librerías no compilan si en las reglas de compilación se encuentra la opción *–Werrors* de *gcc*. Esto provoca que todos los *warnings* de compilación se traten como errores. Si esto ocurre, eliminar dicha opción.

Una vez compilado, se crea el fichero de librería *libuclmmbase.a* en el directorio */src* que contiene las librerías RTP. En ese mismo directorio se encuentran todos los ficheros de cabecera .h. Para RTP se necesitan los ficheros de cabecera:

memory.h
rtp.h
uclconf.h
debug.h

además del fichero *libuclmmbase.a*.

Para añadir las librerías al controlador RTP, se ha renombrado el fichero *libuclmmbase.a* como *rtp.lib*, y junto a los ficheros de cabecera anteriores se han guardado en el mismo directorio donde se guardan los controladores.

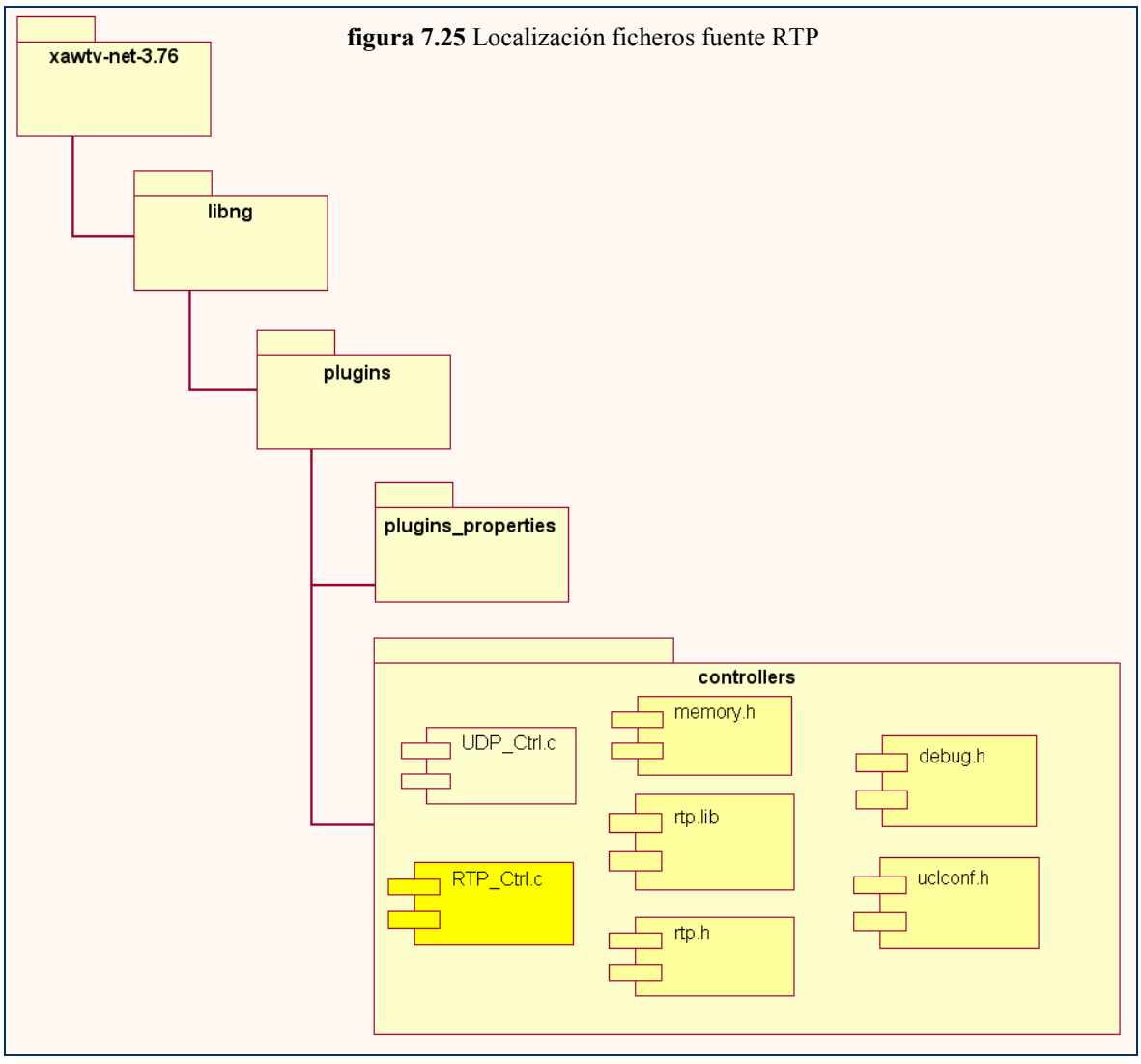


figura 7.25 Localización ficheros fuente RTP

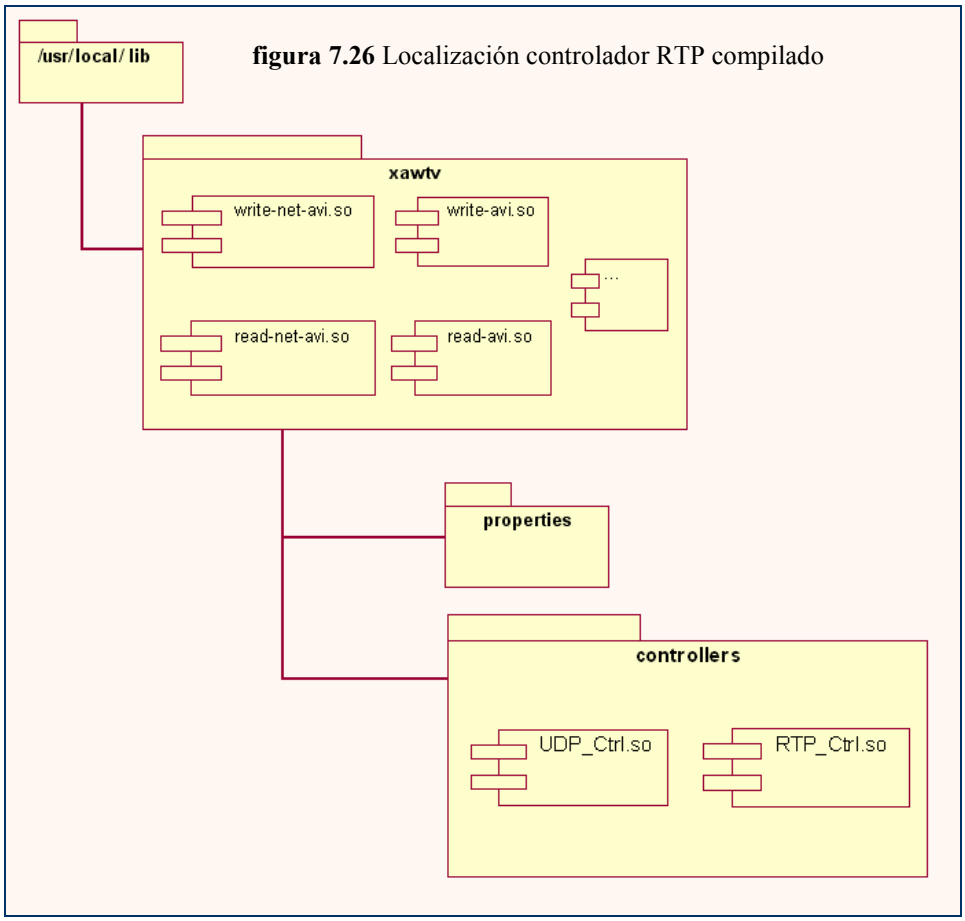


figura 7.26 Localización controlador RTP compilado

El enlace de las librerías con el fichero de controlador RTP en la compilación se explica en el anexo.

No se dispone de una documentación sobre estas librerías. Como se ha visto en apartados anteriores, en ocasiones han dado problemas. Desde problemas de compilación hasta en el funcionamiento de algunas funciones.

Se puede modificar el tamaño máximo de paquete RTP. Por defecto, en el fichero *rtp.h* el valor de la constante **RTP_MAX_PACKET_LEN** es 1500,

```
#define RTP_MAX_PACKET_LEN 1500
```

este valor se ha modificado y se ha asignado el 20.000. Con un tamaño de paquete RTP de 20.000 *bytes* no es necesaria fragmentación si se envía video comprimido en MJPEG. Se necesita recompilar las librerías cada vez que se cambia el tamaño máximo de paquete.

7.3.1.1 Estructuras y funciones definidas en las librerías RTP

A continuación se detallan los tipos, estructuras y funciones más importantes definidas en los ficheros *rtp.h* y *rtp.c*

7.3.1.1.1 Definición de tipos y estructuras.

`rtp_event_type` define los tipos de eventos que pueden producirse en el cliente o en el servidor.

```
typedef enum {

        RX_RTP,          //Se recibe paquete de datos RTP
        RX_SR,           //Se recibe paquete SR
        RX_RR,           //Se recibe paquete RR
        RX_SDES,         //Se recibe paquete SDES
        RX_BYE,          //Se recibe paquete BYE
        SOURCE_CREATED,
        SOURCE_DELETED, //Se borra la fuente
        RX_RR_EMPTY,     //Se recibe paquete RR vacío
        RX_RTCP_START,   //El inicio de procesado de un paquete compuesto RTCP
        RX_RTCP_FINISH, //Finaliza procesado paquete RTCP compuesto
        RR_TIMEOUT,
        RX_APP

} rtp_event_type;
```

`rtp_event` representa un evento.

```
typedef struct {

        uint32_t         ssrc;
        rtp_event_type   type;
        void            *data;
        struct timeval  *ts;

} rtp_event;
```

Esta estructura se pasa a la función **`rtp_callback()`**. El tipo de evento se guarda en el campo **`type`**. El campo **`ssrc`** contiene el SSRC del participante que "dispara" a la función de *callback*. El campo **`*data`** contiene un puntero a datos contenidos en este *callback*. **`*ts`** contiene el *timestamp* del paquete de recepción que causó este evento.

La estructura `struct` **`rtp`** define una sesión RTP

```
struct rtp {

      //OMITIENDO otros campos
      //      .   .   .

      socket_udp  *rtp_socket;
      socket_udp  *rtcp_socket;
      char        *addr;
      uint16_t     rx_port;
      uint16_t     tx_port;
      uint32_t     my_ssrc;
      uint16_t     rtp_seq;

      //      .   .   .
      //OMITIENDO otros campos

};
```

y está definida en *rtp.c*. Se han omitido la mayoría de sus campos. Se observa que crea un *socket* UDP para el protocolo RTP y otro para RTCP. Conserva la dirección IP que se utiliza para crear una sesión en **`*addr`**, así como los puertos de transmisión y recepción, **`tx_port`** y **`rx_port`** respectivamente. Una sesión RTP establece una comunicación entre dos participantes.

El paquete RTP está definido con la estructura `struct` **`rtp_packet`**

```
typedef struct {

        uint32_t        *csrc;
        char            *data;
        int              data_len;
        unsigned char   *extn;
        uint16_t         extn_len;
        uint16_t         extn_type;

        unsigned short    v:2;
        unsigned short    p:1;
        unsigned short    x:1;
        unsigned short    cc:4;
        unsigned short    m:1;
        unsigned short    pt:7;

        uint16_t          seq;
        uint32_t          ts;
        uint32_t          ssrc;

} rtp_packet;
```

Los campos son los que se pueden ver en el apartado 7.1.3.1. Los *bytes* se almacenan según el orden *big endian*[7].

[7] En un sistema *big endian*, el valor más significativo en la secuencia es el que se guarda en la dirección de memoria más baja

Los paquetes RTCP *Sender Report (SR)* y *Receiver Report (RR)* se definen en las estructuras:

```
typedef struct {

        uint32_t        ssrc;
        uint32_t        ntp_sec;
        uint32_t        ntp_frac;
        uint32_t        rtp_ts;
        uint32_t        sender_pcount;
        uint32_t        sender_bcount;

} rtcp_sr;
```

```
typedef struct {

        uint32_t        ssrc;
        uint32_t        fract_lost:8;
        uint32_t        total_lost:24;
        uint32_t        last_seq;
        uint32_t        jitter;
        uint32_t        lsr;
        uint32_t        dlsr;

} rtcp_rr;
```

Los campos corresponden a los que se pueden ver en los apartados 7.1.4.2.1 y 7.1.4.2.2 respectivamente.

En **`rtcp_sdes_type`** se enumeran los identificadores de los posibles *ítems* de SDES (apartado 7.1.4.3). Los ítems se definen en la estructura `struct` **`rtcp_sdes_item`**.

```
typedef enum {

        RTCP_SDES_END   = 0,
        RTCP_SDES_CNAME = 1,
        RTCP_SDES_NAME  = 2,
        RTCP_SDES_EMAIL = 3,
        RTCP_SDES_PHONE = 4,
        RTCP_SDES_LOC   = 5,
        RTCP_SDES_TOOL  = 6,
        RTCP_SDES_NOTE  = 7,
        RTCP_SDES_PRIV  = 8

} rtcp_sdes_type;
```

```
typedef struct {

        rtcp_sdes_type  type;
        uint8_t         length;
        char            data[1];

} rtcp_sdes_item;
```

7.3.1.1.2 Funciones.

La función **rtp_callback()**, maneja los eventos RTP. Ver **rtp_event** para una descripción de los posibles eventos y como **rtp_callback()** debería manejarlos.

```
void (*rtp_callback) (struct rtp *session, rtp_event *e)
```

***session** es la sesión RTP que genera el evento, y ***e** contiene la información sobre el evento.

Para crear una sesión, se debe llamar a la función **rtp_init()**

```
rtp_t rtp_init                  (const char *addr,
                                 uint16_t   rx_port,
                                 uint16_t   tx_port,
                                 int ttl,
                                 double rtcp_bw,
                                 rtp_callback  callback,
                                 int8_t userdata)
```

donde el valor devuelto de **rtp_t** es de tipo **struct rtp** (ver apartado anterior) y representa a la sesión que se crea. ***addr** contiene la dirección IP destino de la sesión (que puede ser *unicast* o *multicast*). El puerto por el que se escucha es **rx_port**, y a través de **tx_port** se envían paquetes UDP. Si se trata de una sesión *multicast*, **ttl** identifica el "tiempo de vida".

La frecuencia con la que se envían los paquetes de control RTCP depende del ancho de banda de la red. Éste se especifica cuando se crea la sesión a través de la variable **rtcp_bw**, expresado en *bytes* por segundo. **userdata** contiene datos asociados a la sesión (no se ha usado), y la función **callback** se encarga de manejar eventos (ver arriba).

Para obtener información sobre una sesión, como puede ser la IP destino, los puertos de recepción y transmisión, SDES (también asignar *ítems* de SDES al SSRC de un participante), o su SSRC, se hace uso de las funciones:

```
char*          rtp_get_addr        (struct rtp *session)

uint16_t       rtp_get_rx_port     (struct rtp *session)

uint16_t       rtp_get_tx_port     (struct rtp *session)

const char*    rtp_get_sdes        (struct rtp *session,
                                    uint32_t ssrc,
                                    rtcp_sdes_type type)

int            rtp_set_sdes        (struct rtp *session,
                                    uint32_t ssrc,
                                    rtcp_sdes_type type,
                                    const char *value,
                                    int length

uint32_t       rtp_my_ssrc         (struct rtp *session)
```

Para enviar el video, se usa la función **`rtp_send_data()`**

```
int  rtp_send_data              (struct rtp *session,
                                 uint32_t rtp_ts,
                                 char pt,
                                 int m,
                                 int cc,
                                 uint32_t csrc[],
                                 char *data,
                                 int data_len,
                                 char *extn,
                                 uint16_t extn_len,
                                 uint16_t extn_type)
```

que envía un paquete RTP. La mayoría de las aplicaciones solo utilizan los argumentos **`session, rtp_ts, pt, m, data,`** y **`data_len.`** *Mixers y translators* fijan los campos **`cc`** y **`csrc`**. La descripción de los parámetros es la siguiente:

- **`session`**, el puntero a la sesión (devuelto por **`rtp_init()`**)
- **`rtp_ts`**, *timestamp* que refleja el instante en que se muestreó el primer octeto de los datos RTP a enviar
- **`pt`**, identificador de *payload*
- **`m`**, *marker bit*
- **`cc`**, número de fuentes contribuyentes
- **`cscr`**, *array* de identificadores SSRC para las fuentes contribuyentes
- **`data`**, los datos (el video) RTP que van a ser enviados
- **`data_len`**, el tamaño de los datos en *bytes*
- **`extn`**, *extension data*
- **`extn_len`**, tamaño de **`extn`** en *bytes*
- **`extn_type`**, indicador de tipo de *extension*

La función devuelve el número de *bytes* transmitidos.

Los paquetes de control se envían mediante la función rtp_send_ctrl(),

```
void  rtp_send_ctrl          (struct rtp *session,
                              uint32_t rtp_ts,
                              rtcp_app_callback appcallback)
```

esta función comprueba el *timer* **rtp_ts** RTCP y envía datos RTCP cuando sea necesario. La frecuencia del envío de paquetes de control depende del ancho de banda de la sesión ***session**, del número de participantes y si el participante local es un emisor. La función debería ser llamada por lo menos una vez por segundo, y puede llamarse frecuentemente sin que pueda ocasionar problemas. El último parámetro suele ser NULL.

Los paquetes tanto de control RTCP como de datos RTP se reciben mediante la función **rtp_recv()**

```
int  rtp_recv                (struct rtp *session,
                              struct timeval *timeout,
                              uint32_t curr_rtp_ts)
```

donde ***session** es el puntero a una sesión devuelta por **rtp_init()**, ***timeout** es el tiempo que **rtp_recv()** puede permanecer bloqueado, **curr_rtp_ts** es la *timestamp* actual. La función devuelve TRUE si se ha habido recepción y FALSE si salta el *timeout*.

Un participante que se desconecta, envía el mensaje de control RTCP BYE a través de la función **rtp_send_bye()**. Cuando se recibe este paquete, se elimina la sesión que se creó con ese participante mediante la función **rtp_done()**.

```
void  rtp_send_bye           (struct rtp *session)

void  rtp_done               (struct rtp *session)
```

Otras función de la que se ha hecho uso en la implementación del controlador RTP, es

```
void  rtp_update             (struct rtp *session)
```

rtp_update(), que actualiza estructuras internas y facilita la administración. Debe ser llamada al menos una vez por segundo, puede llamarse con tanta frecuencia como se desee.

7.3.2 La estructura *RTP_Ctrl*

Implementa a la estructura **controller.**

```
struct Controller RTP_Ctrl = {

        name:           "RTP",
        id:             1,
        init:           init_RTP,           //SERVIDOR
        add_Client:     add_RTP_Client,     //SERVIDOR
        remove_Client:  remove_RTP_Client,  //SERVIDOR
        connect:        connect_RTP,        //CLIENTE
        send:           send_RTP,           //SERVIDOR
        recv:           recv_RTP,           //CLIENTE
        close:          close_RTP,          //SERVIDOR Y CLIENTE
};
```

FUNCIÓN	*IMPLEMENTA*	*Descripción*
init_RTP()	init()	Inicializa controlador
add_RTP_Client()	add_Client()	Añade cliente al servidor
remove_RTP_Client()	remove_Client()	Borra cliente del servidor
connect_RTP()	connect()	Conecta cliente con servidor
send_RTP()	send()	Envía video
recv_RTP()	recv()	Recibe video
close_RTP()	close()	Cierra conexión

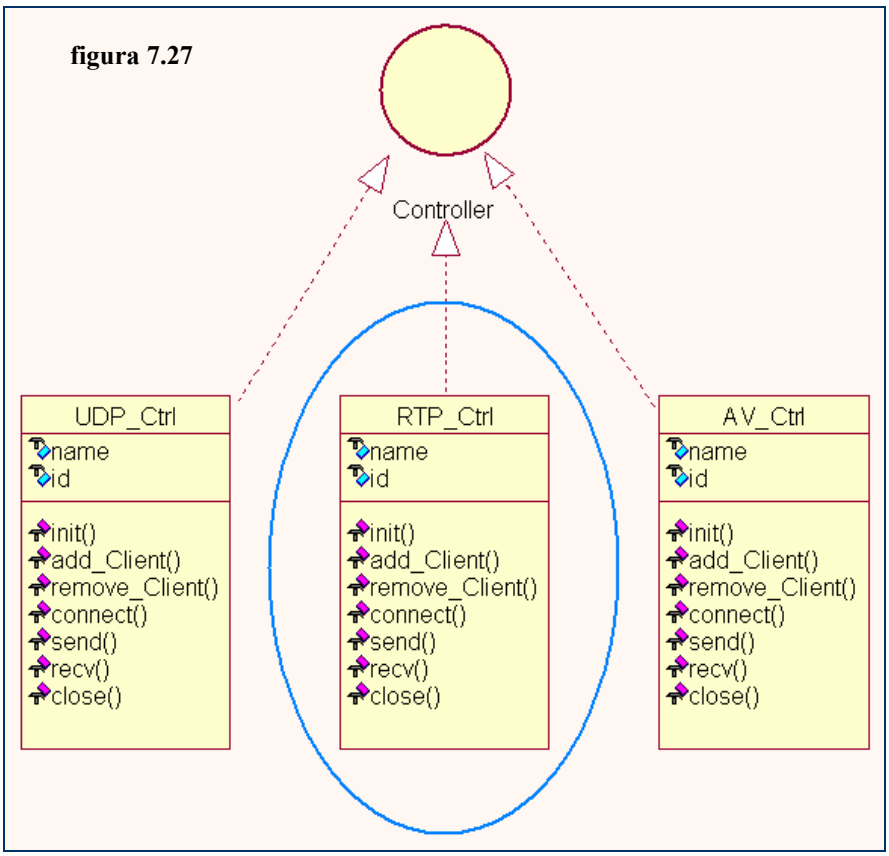


figura 7.27

7.3.3 Otras estructuras

7.3.3.1 La estructura *connection*

```
struct connection {

	int active;
	int round;
	int ssrc_check_count;
	int cname_recv;

	char cname_id[10];

	uint32_t client_video_SSRC;
	uint32_t client_audio_SSRC;

	struct negotiation n;

	struct rtp *video_session;
	struct rtp *audio_session;

};
```

Contiene la información relacionada con la conexión de un cliente. El campo **`active`** se pone a 1 cuando el cliente está conectado (en caso contrario a 0). Los parámetros que se negociaron con el cliente se guardan en la estructura ***`negotiation`*** **`n`**. Estos dos campos de la estructura realizan la misma función que en el controlador UDP.

`round` se emplea para *timestamp* local en los mensajes de control, **`ssrc_check_count`** es un contador interno que posee el servidor para cada cliente.

Si un cliente no ha generado paquetes de control RTCP antes de recibir 100 *frames*, el servidor lo elimina borrando su estructura `struct` **`connection`** (ver figuras 7.23 y 7.24), la variable que lleva la cuenta de los *frames* enviados antes de recibir el SSRC cliente es **`ssrc_check_count`**. Como ya se ha visto, un cliente se identifica mediante su CNAME, y a partir de este el servidor sabe si el SSRC cliente ha cambiado. Cuando se recibe el primer mensaje de control RTCP que contiene el CNAME del cliente, la variable **`cname_recv`** se pone a 1 (vale 0 mientras no lo haya recibido) y éste se guarda en el *array* **`cname_id[]`**.

El SSRC del cliente se guarda en **`client_video_SSRC`**. La transmisión de audio no está implementada, pero si lo estuviera el cliente tendría otra sesión para el audio con otro SSRC diferente que se guardaría en **`client_audio_SSRC`**.

La sesión que crea el servidor con el cliente para enviarle el video se guarda en **`*video_session`** (y si creara una sesión de audio, se guardaría en **`*audio_session`**). Esta sesión es la que crea el servidor con el cliente y tendrá un SSRC que no nos interesa, pues es el del servidor. El campo **`client_video_SSRC`** contiene el SSRC de la sesión que creó el cliente con el servidor, y es el SSRC que sí nos interesa para poder identificar la procedencia de los paquetes de control RTCP que recibe el servidor.

```
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. Ésta es la tabla de conexiones que posee el servidor.

7.3.3.2 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;
};
```

Es la misma que se definió en el apartado 5.3.4.

7.3.4 RTP_Ctrl

Importa las librerías RTP

```
#include "uclconf.h"
#include "debug.h"
#include "memory.h"
#include "rtp.h"
```

Debe respetarse ese orden de importación de las librerías.

Todas las funciones RTP que se han utilizado y se nombran en el apartado siguiente se encuentran definidas y explicadas en el apartado 7.3.2.2.

7.3.4.1 Implementación de funciones servidor

A continuación se describen las funciones relacionadas con el servidor que contiene el controlador RTP.

7.3.4.1.1 *init_RTP()*

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

Inicializa todos los campos de todas las estructuras de tipo **`connection`** del *array* de conexiones a cero.

7.3.4.1.2 *add_RTP_Client()*

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

Añade un cliente creando una sesión con él. La secuencia que se lleva a cabo para añadir un cliente RTP es la siguiente:

- Si el cliente que intenta conectar (cuya dirección se encuentra en **`cli_ad`**) está activo en la tabla de conexiones del servidor (*array* de conexiones), indica que no se desconectó correctamente y provocaría que no pudiera volver a conectar. Se lleva a cabo esta comprobación.
  - Si el cliente se encuentra en la tabla, se borra para que pueda volver a conectar
  - Si no se encuentra, se intenta añadir una nueva conexión
- Se comprueba si hay sitio en el *array* de conexiones.
  - si queda sitio:
    - Se añade al *array*
    - Se crea la sesión Servidor → Cliente con la llamada a la función **`rtp_init()`** (ver apartado 7.2.3 y figura 7.19). La función de *callback* **`server_rtp_event_handler()`** se explica en el apartado 7.4.3.4.1.6

```
unsigned short int tx_Server_port = 4500;
unsigned short int rx_Server_port = 9000;
unsigned short int Time_To_Live   = 2;
unsigned short int Bandwidth      = 100000;


connections[i].video_session = rtp_init(

                          inet_ntoa(cli_ad.sin_addr), //Host/Group direccion IP
                          rx_Server_port,             //Puerto recepción
                          tx_Server_port,             //Puerto transmisión
                          Time_To_Live,               //Tiempo de vida
                          Bandwidth,                  //Ancho de banda estimado
                          server_rtp_event_handler,   //RTP event callback
                          NULL);                      //Datos específicos de la aplic
```

- Se establecen parámetros sobre la conexión.
- 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 hace uso del *array* **`delay[]`** cuya función es la misma que en la del controlador UDP (controla el *rate* de cada cliente)
- Se devuelve el valor 0.

o Si no queda sitio, no se añade. La función devuelve -1

7.3.4.1.3 *remove_RTP_Client()*

No se ha implementado. Un cliente es borrado por el servidor al recibir el paquete de control RTCP BYE.

7.3.4.1.4 *send_RTP()*

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

Envía un *frame* (la información de video contenida en **`*buf)`**, también puede enviar y recibir información de control RTCP. **`*handle`** es de tipo **`avi_handle`** (apartado 6.4.2.3), de ella se obtiene el formato de video y la compresión. Sólo se ha implementado el formato MJPEG.

Lo primero que se hace es comprobar si hay alguna conexión activa en el *array* de conexiones a través de la variable **`empty`**, si no la hay, no se hace nada. Si hay conexiones activas, la secuencia de ejecución que se sigue para cada conexión activa, es la siguiente:

- Se llama a la función **`rtp_recv()`**, que puede recibir tanto datos de video como de control. Los clientes nunca envían datos de video, por lo que el servidor sólo puede recibir datos de control. Esta función sólo recibe si le han enviado información, pero nunca se queda bloqueada (tiene un *timeout*). Por cada recepción, se invoca a la función de *callback* manejadora de eventos **`server_rtp_event_handler()`**

- Se vuelve a comprobar si la conexión sigue activa, porque el cliente podría haber enviado un paquete **BYE** (en la acción anterior), y la función manejadora de eventos haberlo eliminado de la tabla.
  - Si se recibe un paquete BYE:
    - Se llama a la función **`rtp_done()`** que finaliza la sesión RTP
    - Se vuelve al inicio de la función para servir video a otros posibles clientes conectados.
  - Si no se recibe, y la conexión sigue activa:
    - Se comprueba si se conoce ya SSRC cliente, a través de la variable **`ssrc_check_count`** como se explicó en el apartado 7.3.3.1 y se observa en las figuras 7.23 y 7.24.
      - Si al cabo de 100 intentos de recepción de información del cliente (**`rtp_recv()`**) no se ha obtenido su SSRC, la sesión se cierra con la llamada a la función **`rtp_done()`** y se borra al cliente de la tabla.
      - En caso contrario, no se hace nada.
    - Se actualiza la información sobre el envío y recepción de datos de control
    - Se envían datos de control al cliente mediante la función **`rtp_send_ctrl()`**.
    - Se envía video con la función **`rtp_send_data()`**

```
rtp_send_data (connections[i].video_session,    //Sesión
               rtp_ts, MULAW_PAYLOAD,           //Timestamp
               0, 0, 0,
               (char*)buf->data, buf->size,     //Video
               0, 0, 0);
```

    - Llamada a **`rtp_update()`**
    - Actualización del *array* de retardos **`delay[]`**

7.3.4.1.5 *close_RTP()*

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

La función de cerrar una sesión RTP la llaman tanto cliente como servidor, si la llama el servidor, **`cs = SERVER`**.

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

Recorre el *array* de conexiones activas, manda el paquete de control RTCP BYE con la función **`rtp_send_bye()`** para informar a los clientes de que desconecta y finaliza todas las sesiones con la función **`rtp_done()`** .

7.3.4.1.6 Función *callback* manejadora de eventos: *server_rtp_event_handler()*

```
static void server_rtp_event_handler(struct rtp *session, rtp_event *e)
```

Maneja todos los eventos que provoca el cliente en el servidor. Esta función se llama cuando se recibe cualquier tipo de paquete RTCP o RTP, se encarga de analizar el tipo de información que contiene y actuar en consecuencia. De la estructura **`rtp_event *e`** se obtiene el tipo de evento que se genera, los datos y el SSRC del cliente que generó el evento. **`*session`** es la sesión de la que procede el evento (sesión Servidor → Cliente). Un ejemplo de generación de eventos cliente/servidor puede verse en el apartado 7.2.2

Se analiza el tipo de evento generado (descritos en el apartado 7.3.2.1):

- **RX_RTP, Recepción de datos RTP.** El servidor no recibe datos de vídeo. El código asociado a este evento está comentado, pero podría usarse si el servidor recibiera datos de video, y llamaría a la función **`server_data_recv()`**
- **RX_SDES, Recepción de datos sobre descripción de sesión.** Los datos sobre descripción de sesión se reciben constantemente. Esta información la maneja la función **`server_sdes_ctrl()`** (apartado siguiente)
- **RR_TIMEOUT**, parece ser que este *timeout* no funciona bien… y no ha podido usarse.
- **RX_BYE, Recepción de paquete BYE.** Se recibe cuando el cliente se ha desconectado.
    - Compara SSRC de quien genera el evento con los SSRC de las conexiones activas
    - Cuando se conoce la conexión que ha generado el evento (a través de su SSRC), fija el campo **`active`** = 0 e inicializa los campos asociados a la conexión.
    - Se comprueba si era la única conexión activa
        - Si es la única activa fija **`empty`** = 1, para que no se produzcan errores y se intente enviar a un cliente que no existe
        - Si no es la única, no hace nada
- **SOURCE_CREATED, Creada la fuente cliente**. Este es el único evento en el que cuando se intenta obtener la dirección IP del cliente que lo genera, lo hace correctamente (para los demás eventos, la obtención de la IP cliente nunca es la correcta, aún no se sabe por qué…). Sólo se produce una vez por conexión. Los pasos que sigue son:
    - Como este evento se produce instantes más tarde de la creación de la sesión que ya ha sido añadida al *array* de conexiones (incluso el servidor ya está enviándole video al cliente), se lleva a cabo una comparación entre la IP de la sesión que ha generado el evento y la dirección IP de todas las conexiones activas.
    - Cuando la encuentra, asocia el SSRC del evento a la variable **`client_video_SSRC`** de la estructura **`connection`** relacionada con ese cliente. Aquí es cuando se obtiene por primera vez el SSRC de un cliente.

- Para los demás eventos:
    - **SOURCE_DELETED**
    - **RX_SR**
    - **RX_RR**
    - **RX_RR_EMPTY**
    - **RX_RTCP_START**
    - **RX_RTCP_FINISH**
    - **RX_APP**

  no se realiza ninguna operación.

7.3.4.1.7 Función de control de descripción de sesión: *server_sdes_ctrl()*

```
static void server_sdes_ctrl(struct rtp *session,
                             uint32_t ssrc,
                             rtcp_sdes_type stype)
```

Esta función realiza un *UPDATE* de la información de todas las conexiones. Se llama para manejar la información de descripción de sesión cuando se recibe un paquete de tipo **RX_SDES**.

Realiza operaciones a partir del tipo de información SDES descrita en la variable **`stype`**. Recordemos que puede ser de tipo CNAME, NAME, EMAIL, TELEPHONE, LOCATION, TOOL, NOTE o PRIV. Sólo se hace uso del tipo CNAME, los demás tipos se omiten. **`*session`** identifica a la sesión que provoca el evento de recepción de SDES, y **`ssrc`** es el SSRC del cliente que ha generado este evento.

Es la función que se encarga de actualizar el SSRC del cliente en la tabla de conexiones del servidor si éste ha cambiado (remotamente). La secuencia que lleva a cabo esta función es:

- Se comprueba que los datos recibidos son de tipo CNAME. Sólo se contempla el caso de que lo sean.
- De todas las conexiones del *array* **`connections`**, se comprueban las que están activas y si tienen asignado o no un CNAME (dependiendo del valor de la variable **`cname_recv`**).
    - Si una conexión no tiene asignado un CNAME* y está activa, se compara el SSRC que tiene asignado y el SSRC que se encuentra en el paquete que generó el evento. Si coincide, se asigna el CNAME que se obtiene del paquete SDES a esa conexión. Este CNAME no cambia nunca y permite diferenciar unívocamente una conexión. Así, un cliente puede contar con dos sesiones (dos SSRC) diferentes y ser identificado mediante su CNAME.

* sólo puede existir una conexión que no tenga asignado el CNAME, pues entre dos conexiones consecutivas se “fuerza” el transcurso de un tiempo determinado, y el CNAME se asigna en los primeros instantes de creación de la sesión.

- Para las conexiones que ya tienen asignado un CNAME, se compara el valor de CNAME contenido en los datos del evento, con los CNAME del *array* de conexiones.
  - Si el CNAME coincide con el de alguna conexión, se actualiza el SSRC de la variable **`client_video_SSRC`** con el valor **`ssrc`**.
  - Si CNAME no coincide, no se hace nada.

Este proceso es el ya explicado en el apartado 7.2.3.

7.3.4.2 Implementación de funciones cliente

Se describen a continuación las funciones relacionadas con el cliente que contiene el controlador RTP.

7.3.4.2.1 *connect_RTP()*

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

Con la llamada a la función **`connect_RTP()`** el cliente crea una sesión con el servidor de video.

Los valores resultantes de la negociación están guardados en **`ng`**, y **`*handle`** es de tipo `struct` **`avi_handle`** que contiene la información relacionada con el formato de video y la dirección IP del servidor que se pasa como parámetro desde la línea de comandos.

El cliente crea la sesión con el servidor Cliente → Servidor con la función **`rtp_init()`** (ver apartado 7.2.3). La función de *callback* **`client_rtp_event_handler()`** se explica en el apartado 7.4.3.4.2.4. Como se observa, los puertos a los que transmite el cliente son por los que recibe el servidor, y viceversa.

```
unsigned short int tx_Client_port = 9000;
unsigned short int rx_Client_port = 4500;
unsigned short int Time_To_Live   = 2;
unsigned short int Bandwidth      = 100000;

Client_Session = rtp_init(
                          h->IP,                    //direccion IP servidor
                          rx_Client_port,           //puerto recepción
                          tx_Client_port,           //puerto transmisión
                          Time_To_Live,             //tiempo de vida
                          Bandwidth,                //B/W estimado
                          client_rtp_event_handler, //RTP event callback
                          NULL);                    //datos especificos de la aplic
```

Finalmente, con el uso de la función **`rtp_set_sdes()`** se asocia el CNAME y los demás *ítems* SDES con la sesión y el SSRC cliente.

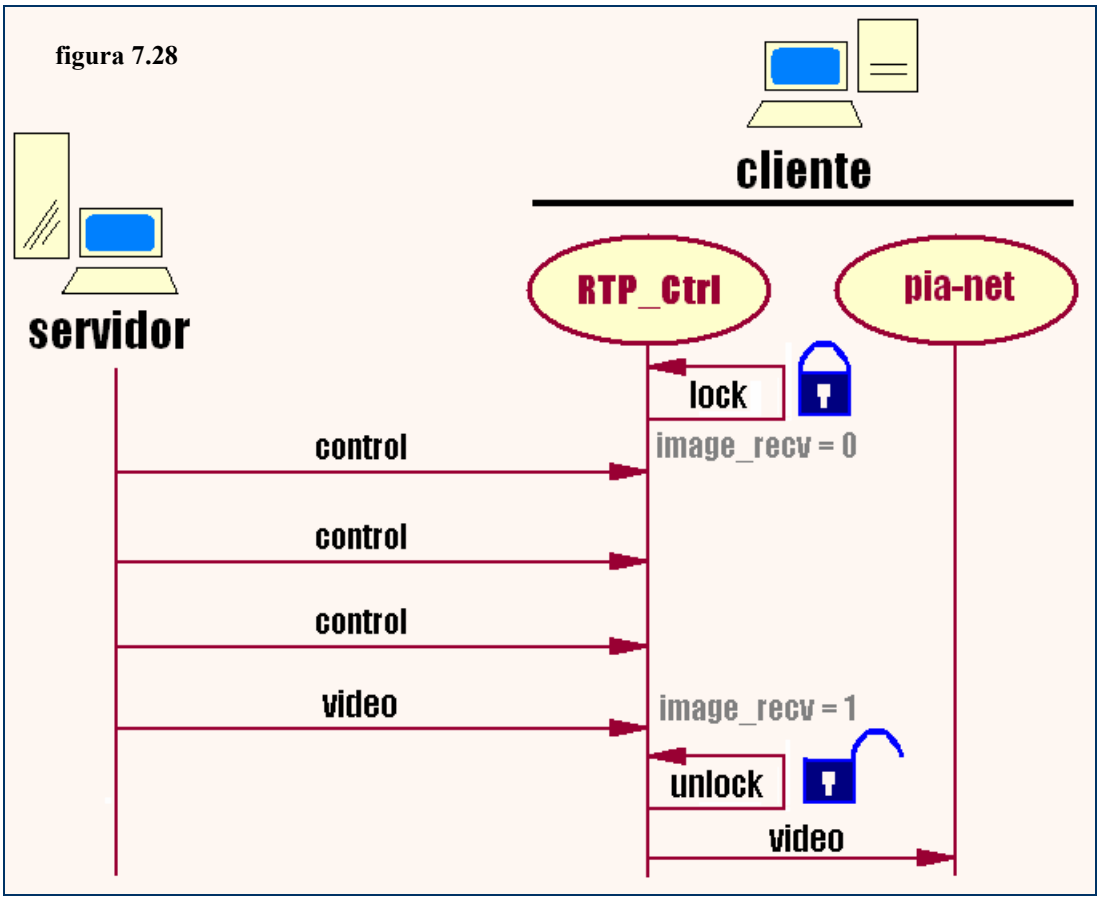
7.3.4.2.2 *recv_UDP()*

```
struct ng_video_buf* recv_RTP(void *handle)
```

Recibe información de control RTCP y de video a través de paquetes RTP. La función devuelve la imagen recibida en una estructura de tipo **`struct`** `ng_video_buf*`. La secuencia de recepción es la siguiente:

- Se reserva memoria para cada *frame* que recibe, tanta como el tamaño máximo de un paquete RTP que pueda contener un *frame*. *pia* internamente se encarga de liberarla.
- La compresión soportada por RTP es únicamente MJPEG. Se comprueba que el formato de video de recepción es MJPEG.
- Se entra en un bucle de recepción del que no se sale hasta que no se ha recibido una imagen. Se sabe cuándo se recibe en el momento en que la variable global **`image_recv`** = 1. (ver figura 7.28)
    - Se actualizan los valores de periodicidad de envío de datos de control RTCP.
    - Se recibe información a través de la función **`rtp_recv()`**, que recordemos, puede recibir información RTCP o de datos RTP (video). Cada vez que se recibe, se invoca a la función de *callback* **`client_rtp_event_handler()`** que maneja los paquetes recibidos. Esta última función, detecta:
        - si se ha recibido una imagen, en ese caso hará uso de la variable global **`image_recv`** (por defecto vale 0). Si una imagen se recibe, fijará el valor de la variable **`image_recv`** = 1, y así se sale del bucle infinito de recepción enviando previamente información de control al servidor mediante **`rtp_send_ctrl()`**. Finalmente se devuelve la imagen recibida.
        - si no se ha recibido imagen (**`image_recv`** = 0), indica que la información recibida es de control RTCP y no se sale del bucle (se llama a **`rtp_update()`** y se vuelve al principio del mismo), pues no hay ninguna imagen que pueda ofrecerse a la aplicación *pia*.

figura 7.28



7.3.4.2.3 *close_UDP()*

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

La función de cerrar una sesión RTP la llaman tanto cliente como servidor, si la llama el servidor, **`cs = CLIENTE`**.

```
if( cs == CLIENT )
{
      rtp_send_bye(Client_Session);
      rtp_done(Client_Session);
}
```

Envía el paquete de control RTCP BYE con la función **`rtp_send_bye()`** para informar al servidor de que se desconecta y finaliza la sesión con la función **`rtp_done()`** .

7.3.4.2.4 Función *callback* manejadora de eventos: *client_rtp_event_handler()*

```
static void client_rtp_event_handler(struct rtp *session, rtp_event *e)
```

Maneja todos los eventos que provoca el servidor en el cliente. Esta función se llama cuando se recibe cualquier tipo de paquete RTCP o RTP, se encarga de analizar el tipo

de información que contiene y actuar en consecuencia. De la estructura **`rtp_event *e`** se obtiene el tipo de evento que se genera, los datos y el SSRC del servidor que generó el evento. **`*session`** es la sesión de la que procede el evento (sesión Cliente → Servidor). Un ejemplo de generación de eventos cliente/servidor puede verse en el apartado 7.2.2

Se analiza el tipo de evento generado (descritos en el apartado 7.3.2.1):

- **RX_RTP, Recepción de datos RTP.** Se han recibido datos de video. Se llama a **`client_data_recv()`** (ver abajo) y se libera memoria del paquete recibido.
- **RX_SDES, Recepción de datos sobre descripción de sesión.** Los datos sobre descripción de sesión se reciben constantemente. Esta información la maneja la función **`client_sdes_ctrl()`** (ver abajo). El cliente no hace nada con la información SDES del servidor.
- **RX_BYE, Recepción de paquete BYE.** Se recibe el paquete BYE cuando el servidor se ha desconectado. La aplicación cliente se sale.
- **SOURCE_CREATED, Creada la fuente**. Se genera este evento cuando el cliente contacta con el servidor

- Para los demás eventos:

    - **SOURCE_DELETED**
    - **RX_SR**
    - **RX_RR**
    - **RX_RR_EMPTY**
    - **RX_RTCP_START**
    - **RX_RTCP_FINISH**
    - **RR_TIMEOUT**
    - **RX_APP**

  no se realiza ninguna operación.

```
static void client_data_recv(struct rtp *session, rtp_packet *p)
```

Es llamada por la función **`client_rtp_event_handler()`** cuando se produce el evento **RX_RTP**, es decir, cuando se recibe una imagen. Extrae la imagen contenida en el paquete RTP **`*p`** y la guarda. Se encarga de "avisar" cuándo se recibe una imagen fijando la variable global **`image_recv`** = 1. Esta variable global se utiliza en el bucle de recepción para saber si se ha recibido el video (apartado 7.4.3.4.2.2).

```
static void client_sdes_ctrl(struct rtp *session,
                             uint32_t ssrc,
                             rtcp_sdes_type stype)
```

La función **`client_sdes_ctrl()`** está sin implementar para el cliente, pues no maneja información de control SDES.

7.3.5 Diagramas de despliegue y de secuencia

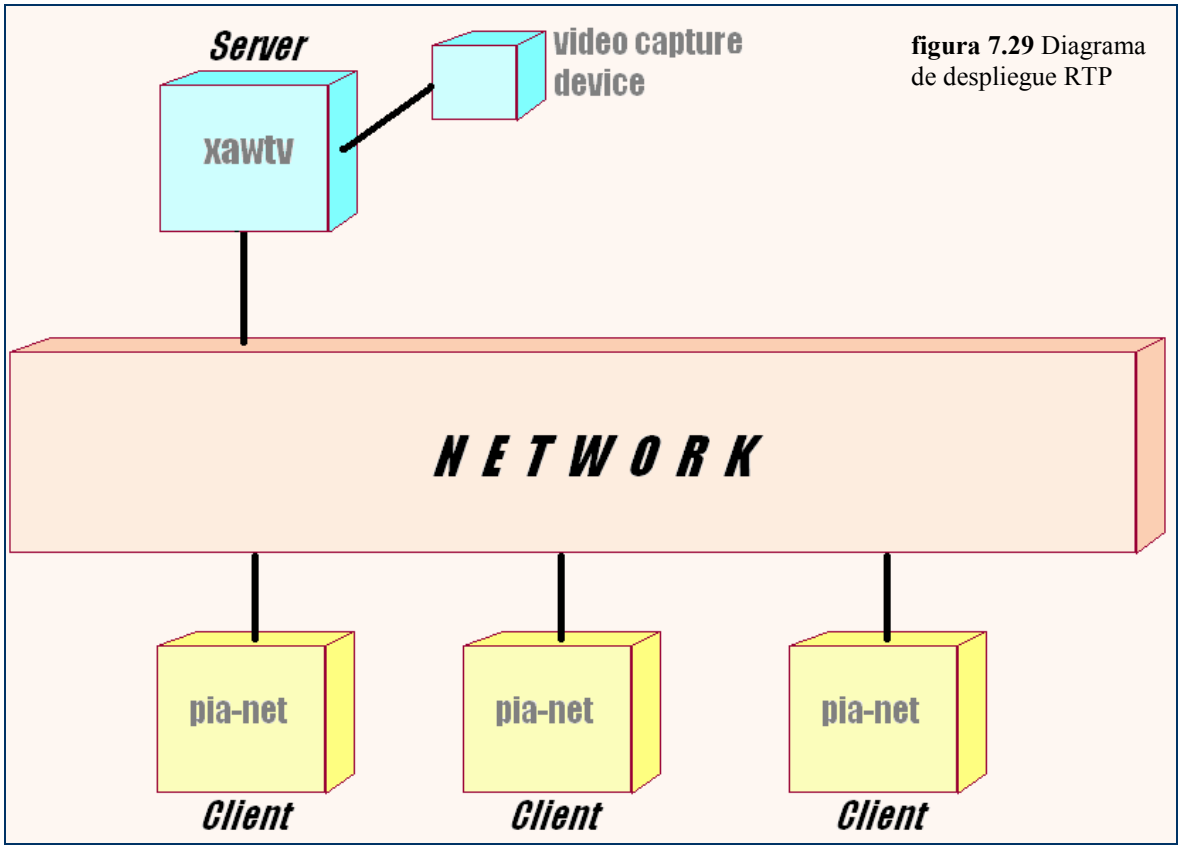
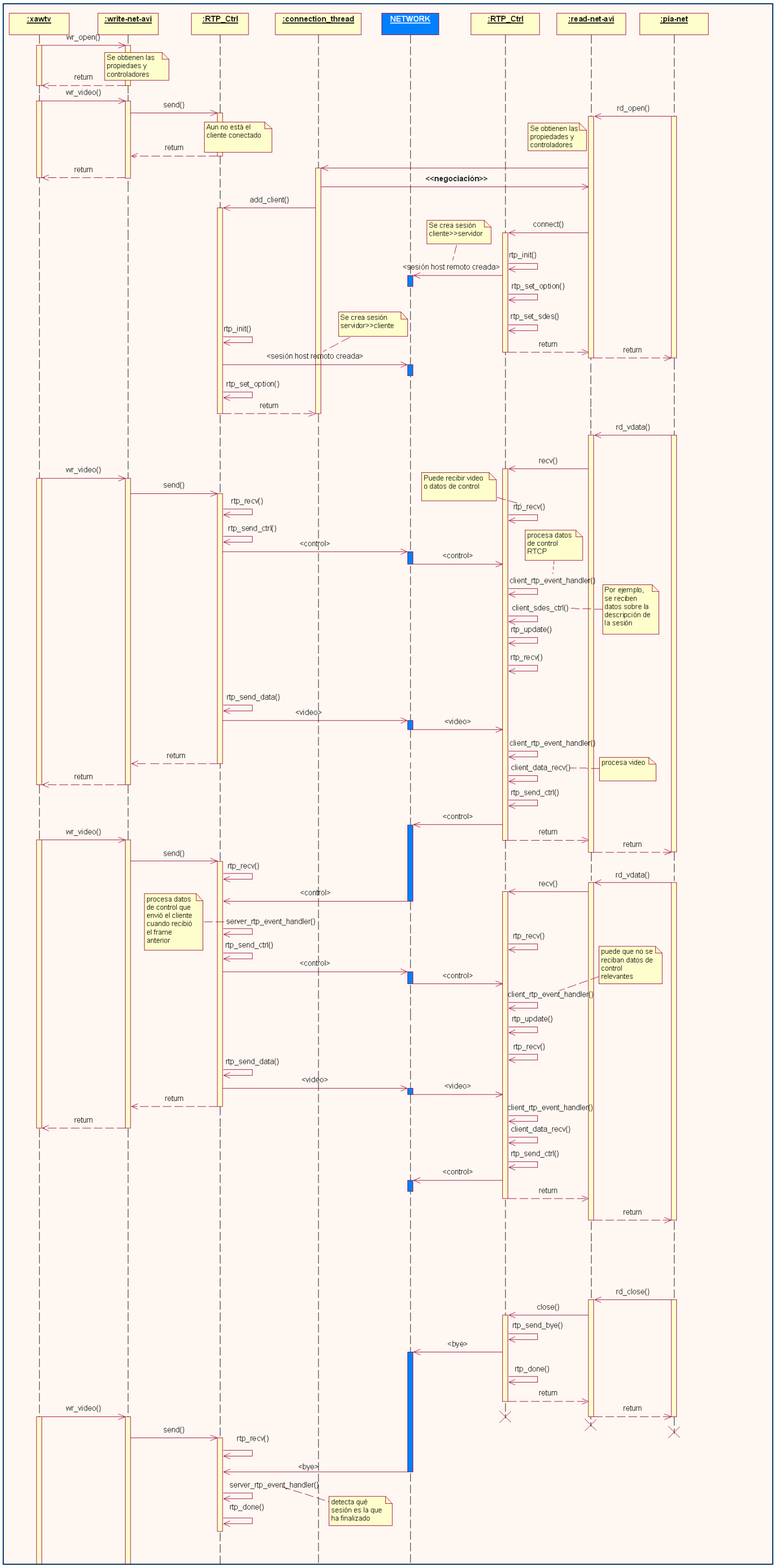


figura 7.29 Diagrama de despliegue RTP

NOTA: En el diagrama de secuencia la figura 7.30, 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 RTP es capaz de manejar varios clientes al mismo tiempo.


:xawtv
:write-net-avi
:RTP_Ctrl
:connection_thread
NETWORK
:RTP_Ctrl
:read-net-avi
:pia-net
wr_open()
Se obtienen las propiedaes y controladores
return
wr_video()
send()
rd_open()
Aun no está el cliente conectado
Se obtienen las propiedades y controladores
return
return
<<negociación>>
add_client()
Se crea sesión cliente>>servidor
connect()
rtp_init()
<sesión host remoto creada>
rtp_set_option()
rtp_set_sdes()
Se crea sesión servidor>>cliente
rtp_init()
return
return
<sesión host remoto creada>
rtp_set_option()
return
rd_vdata()
recv()
wr_video()
send()
Puede recibir video o datos de control
rtp_recv()
rtp_recv()
rtp_send_ctrl()
procesa datos de control RTCP
<control>
<control>
client_rtp_event_handler()
Por ejemplo, se reciben datos sobre la descripción de la sesión
client_sdes_ctrl()
rtp_update()
rtp_recv()
rtp_send_data()
<video>
<video>
client_rtp_event_handler()
return
client_data_recv()
procesa video
return
rtp_send_ctrl()
<control>
wr_video()
return
return
send()
rtp_recv()
rd_vdata()
recv()
<control>
procesa datos de control que envió el cliente cuando recibió el frame anterior
server_rtp_event_handler()
rtp_recv()
rtp_send_ctrl()
puede que no se reciban datos de control relevantes
<control>
<control>
client_rtp_event_handler()
rtp_update()
rtp_recv()
rtp_send_data()
<video>
<video>
return
client_rtp_event_handler()
return
client_data_recv()
rtp_send_ctrl()
<control>
return
return
rd_close()
close()
rtp_send_bye()
<bye>
rtp_done()
return
return
wr_video()
send()
rtp_recv()
<bye>
server_rtp_event_handler()
rtp_done()
detecta qué sesión es la que ha finalizado

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.

Además (ver abajo), 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.

```
•••

# 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

•••
```

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*

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

```
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
```

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)
```

9. La interfaz gráfica *PiaGui*

9.1 Introducción

PiaGui es una interfaz de usuario realizada en Java [22], 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.

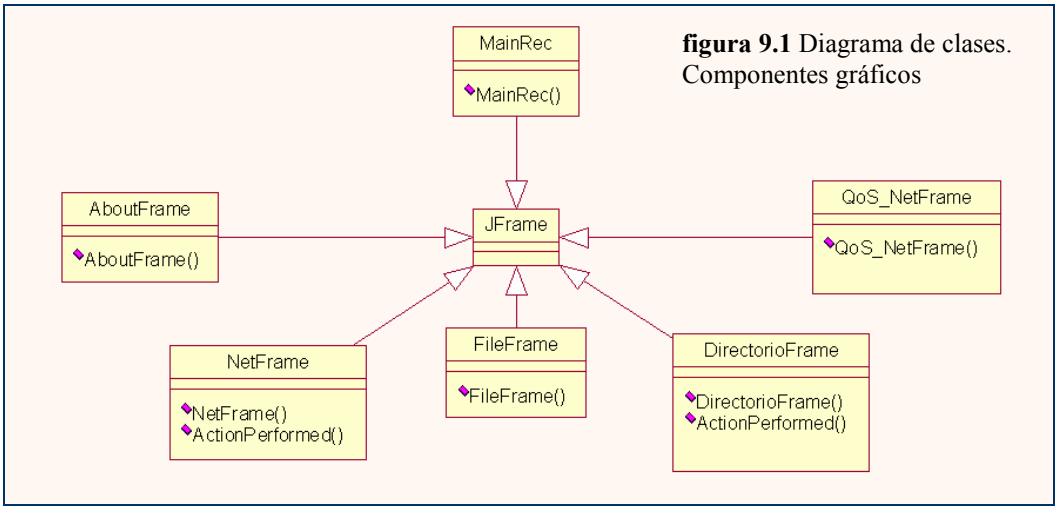


figura 9.1 Diagrama de clases. Componentes gráficos

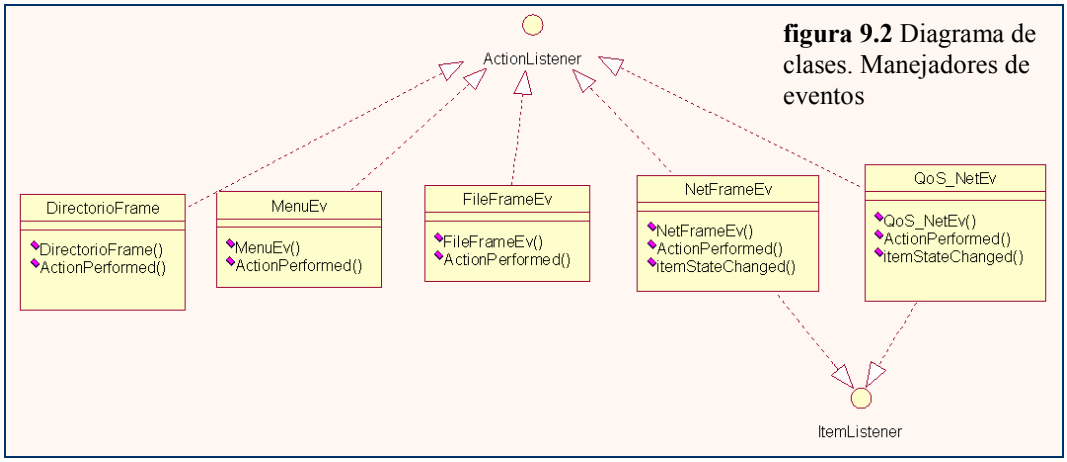


figura 9.2 Diagrama de clases. Manejadores de eventos

AboutFrame.class

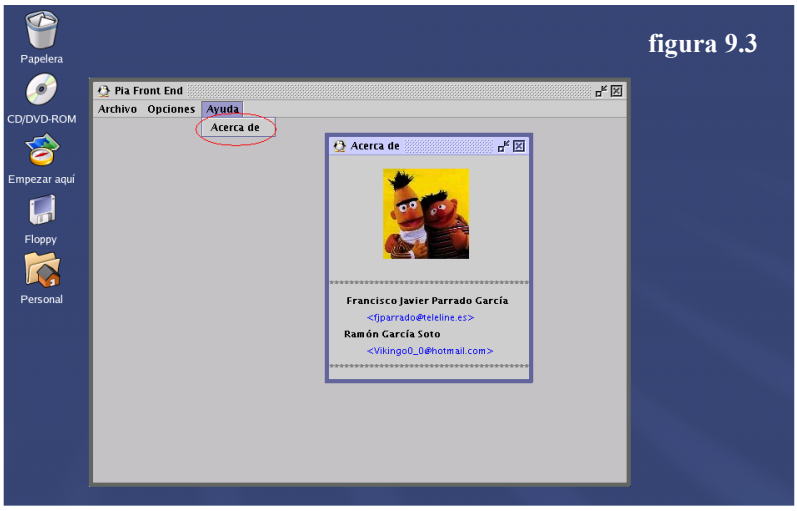


figura 9.3

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

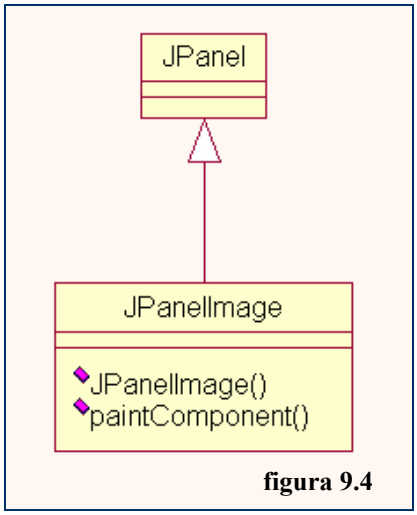
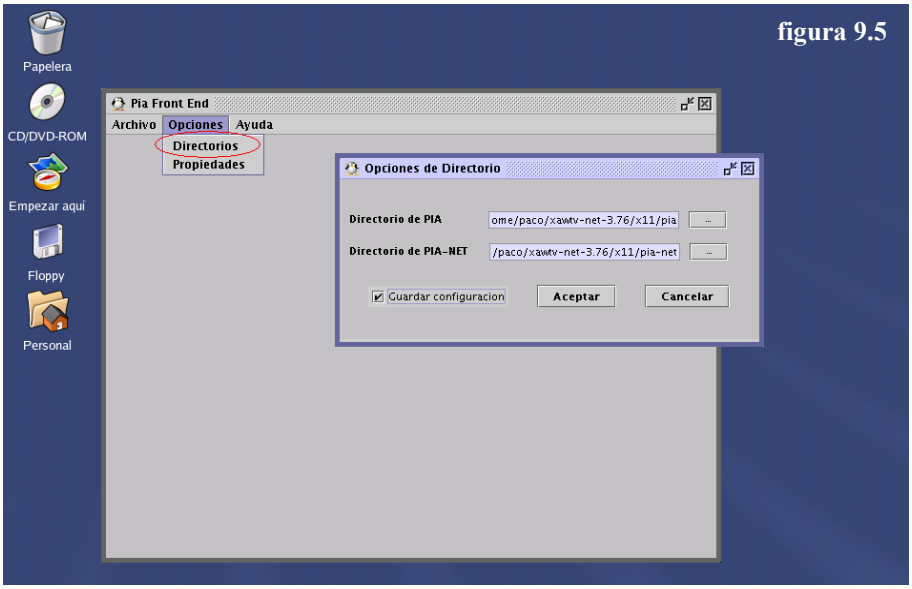


figura 9.4

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

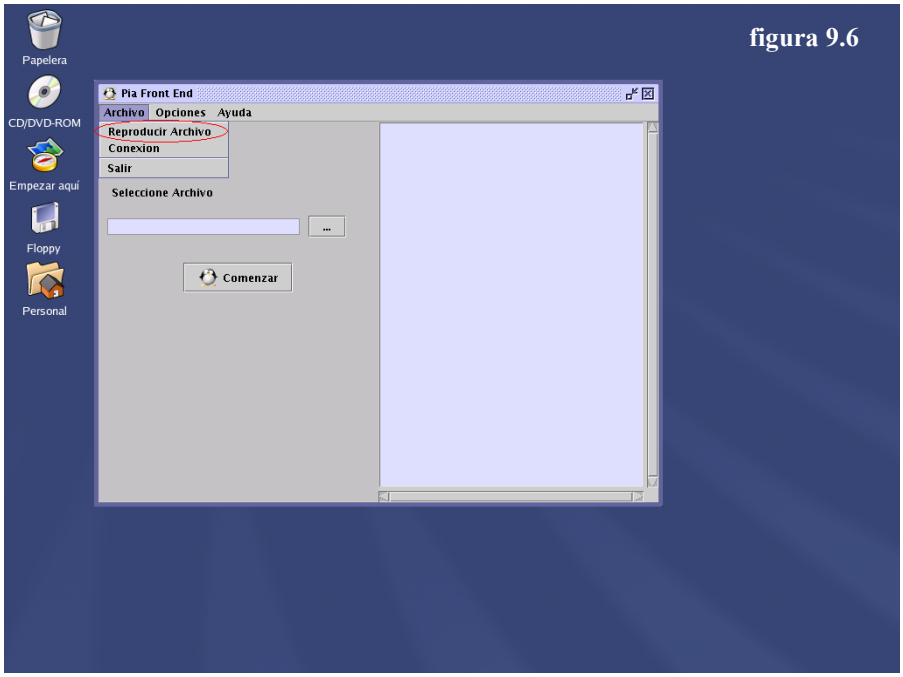
DirectorioFrame.class



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

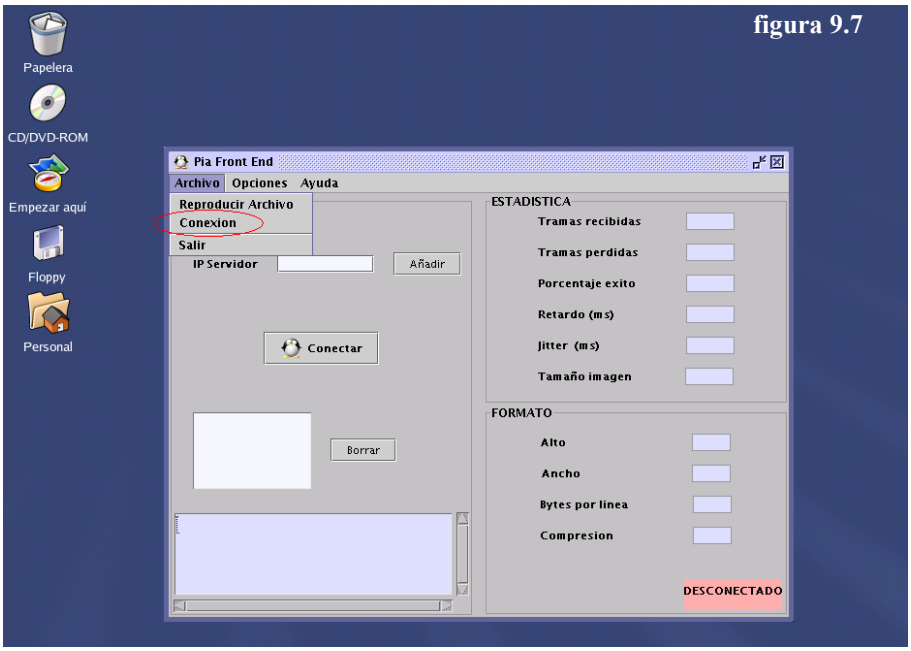


figura 9.7

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.

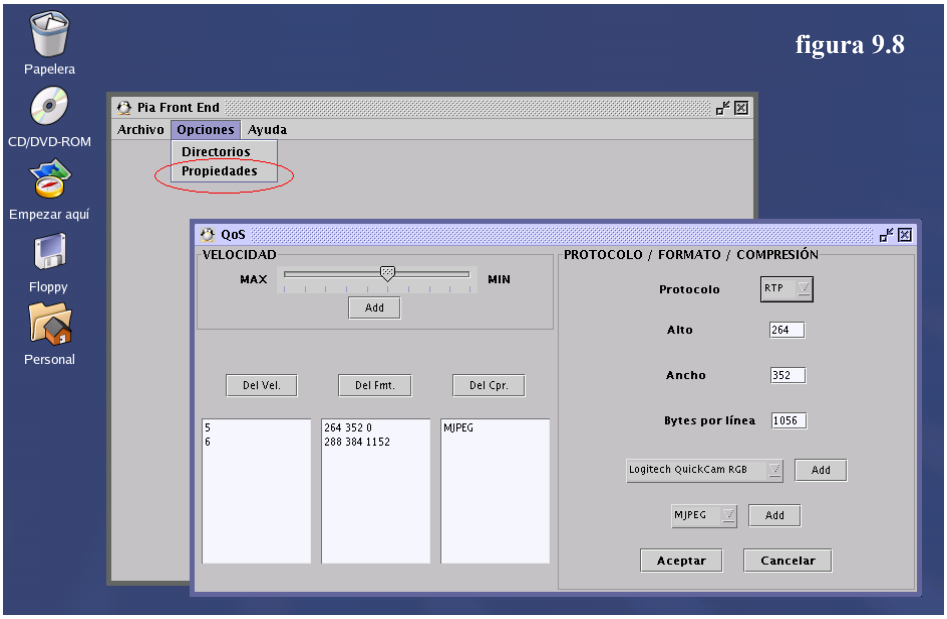
El peso del manejo de eventos se divide entre la propia clase *NetFrame* y una clase manejadora de eventos de tipo *NetFrameEv*.

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*.

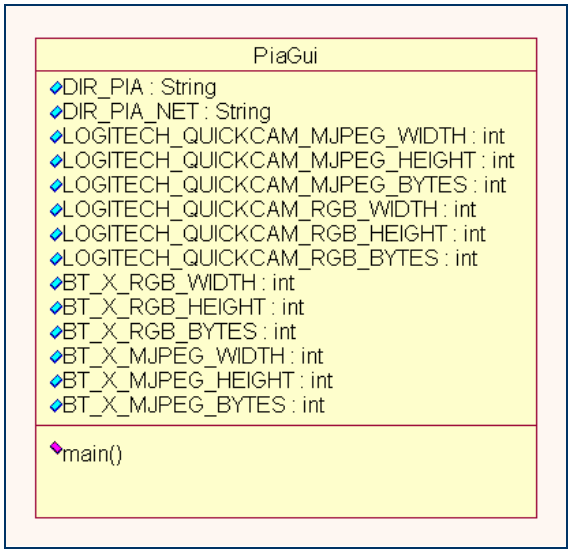
9.2.2 Paquete Images

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

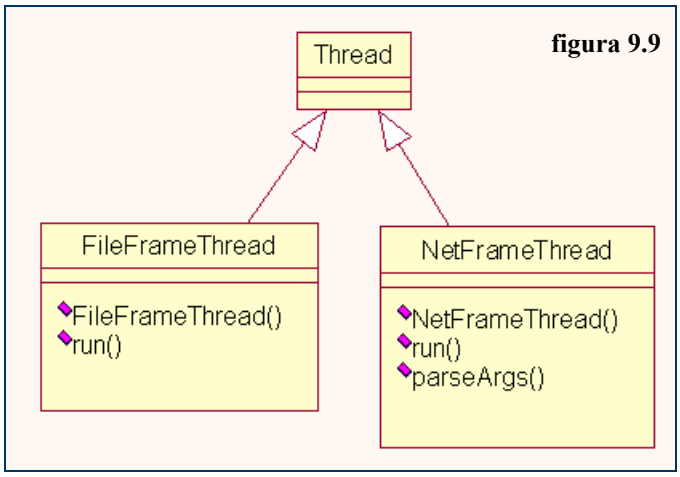


figura 9.9

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

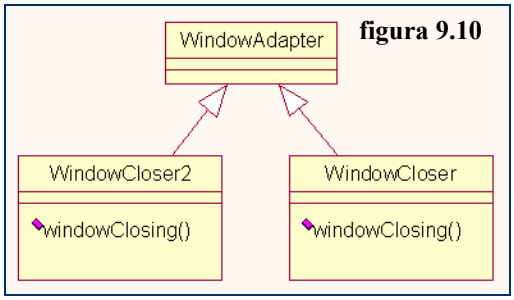


figura 9.10

Manejador para cierre de ventanas.

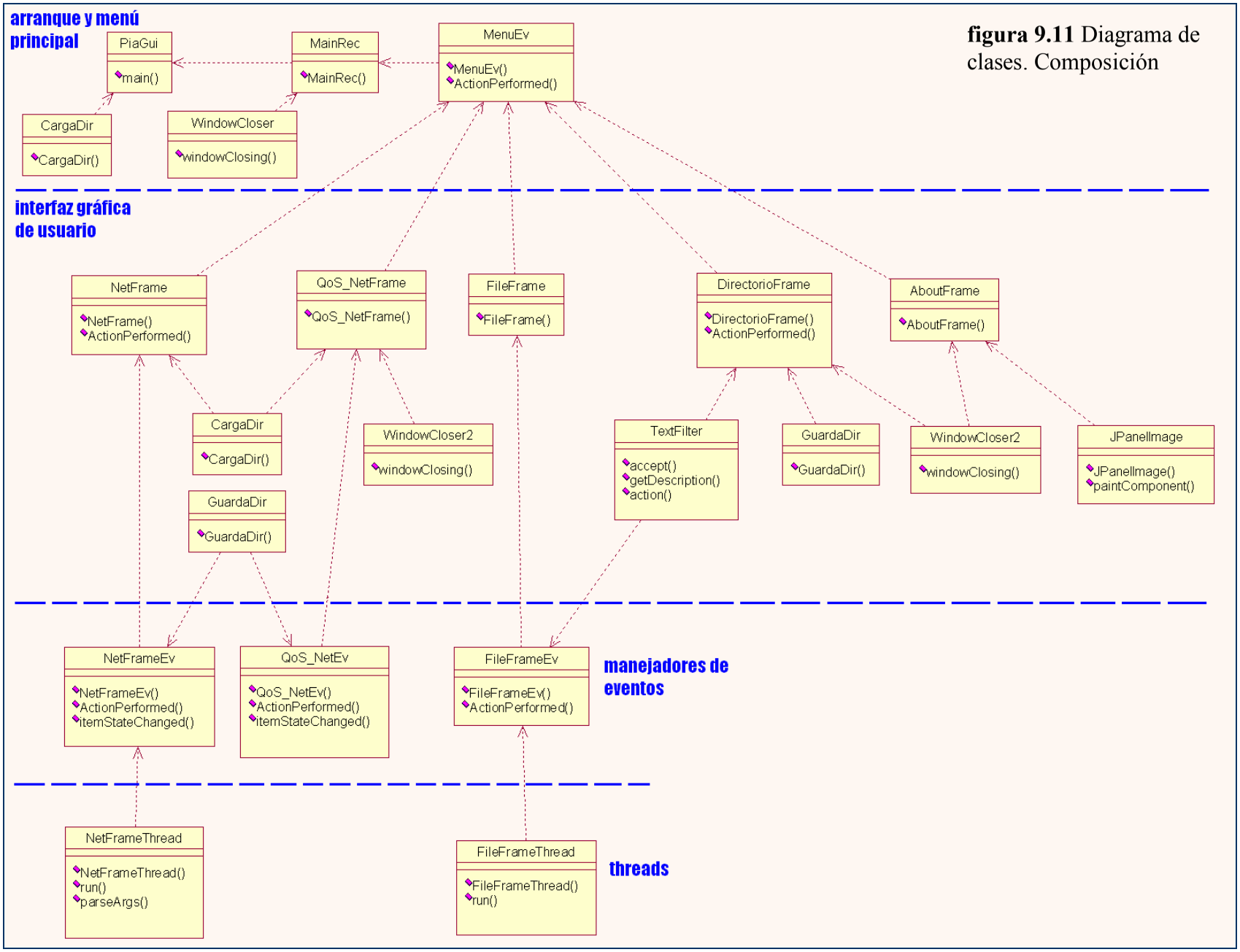


figura 9.11 Diagrama de clases. Composición

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 sobre 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.

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 video 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 video, como se vió en el apartado 7.2.1 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* (*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 RTP tiene sus propias cabeceras.

Esto último no es problema, para RTP 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.

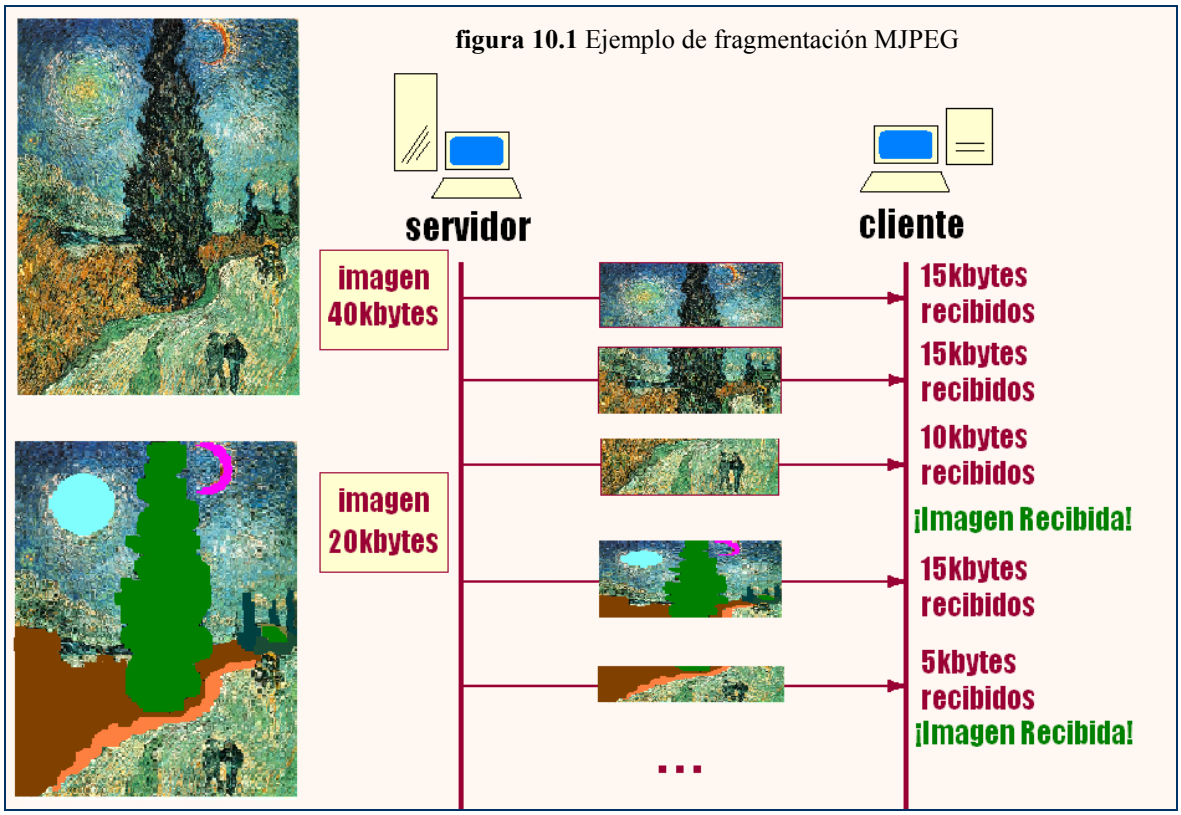


figura 10.1 Ejemplo de fragmentación MJPEG

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 RTP

Las librerías RTP utilizadas han sido usadas parcialmente. En la página web http://www.cs.columbia.edu/~hgs/rtp/ existe una relación de librerías RTP disponibles, tanto gratuitas como de pago.

Las mejores librerías RTP están implementadas en lenguaje C++. Recordemos que el protocolo RTP es un protocolo "nuevo" y que prácticamente no existen implementaciones en lenguaje C. Lo ideal sería trabajar, por ejemplo, con las librerías RTP C++ desarrolladas por los laboratorios Bell, pero *xawtv* está programado en C. Los lenguajes C y C++ pueden trabajar juntos si un programa escrito en ambos lenguajes se compila con un compilador C++, pero las reglas de compilación de *xawtv* están diseñadas para un compilador de C, y no son las mismas que para un compilador de C++. La posibilidad que queda de comunicar ambos lenguaje es mediante la carga de librerías dinámicas escritas en C++, pero es incómodo trabajar de esta forma.

La implementación actual del controlador RTP no gestiona la información RTCP (*Sender y Receiver report*) que intercambian cliente y servidor, y sólo ha sido utilizada parcialmente:

- La información sobre **estampas de tiempo** puede ser útil para conocer los retardos de la red, y un servidor inteligente sería capaz de ajustar la tasa de generación de *frames* a un cliente de forma automática dependiendo de las colisiones de la red.

- Los paquetes ***RR*** y ***SR*** informan sobre el *jitter*. Un cliente puede tener un *buffer* de recepción de tamaño óptimo que permita corregir ese *jitter*, tal y como se vió en la figura 6.2.

- Los **números de secuencia** permiten reordenar imágenes próximas entre sí que llegan desordenadas dentro de ese mismo *buffer*, así como detectar pérdidas de paquetes en la red.

- La **información SDES** es útil para poder crear sesiones selectivas. Determinados clientes reciben determinada información, y el servidor es capaz de identificarlos por sus características.

- Implementar ***mixers*** y ***translators***. Estos últimos permitirían recomprimir la información conociendo el tipo de *payload* y enviarla a clientes que dispongan de menor ancho de banda.

- Intentar implementar la función de ***TIMEOUT*** que parece ser que no funciona correctamente. Si un cliente se desconecta y el servidor no recibe el paquete BYE, nunca conseguiría detectarlo.

- Una comunicación ***multicast*** es mucho más efectiva que las comunicaciones *unicast*.

- En temas de seguridad, la **encriptación** de la información se hace imprescindible.

Así es posible calcular estadísticas y que estas sean mostradas en pantalla a través de la interfaz *PiaGUI*.

10.5 Seguridad en la transmisión: SecureRTP (SRTP)

SRTP es un *perfil* (*profile*, ver apartado 7.1.3.1, campo *Marker*) de seguridad para RTP. Provee al protocolo:

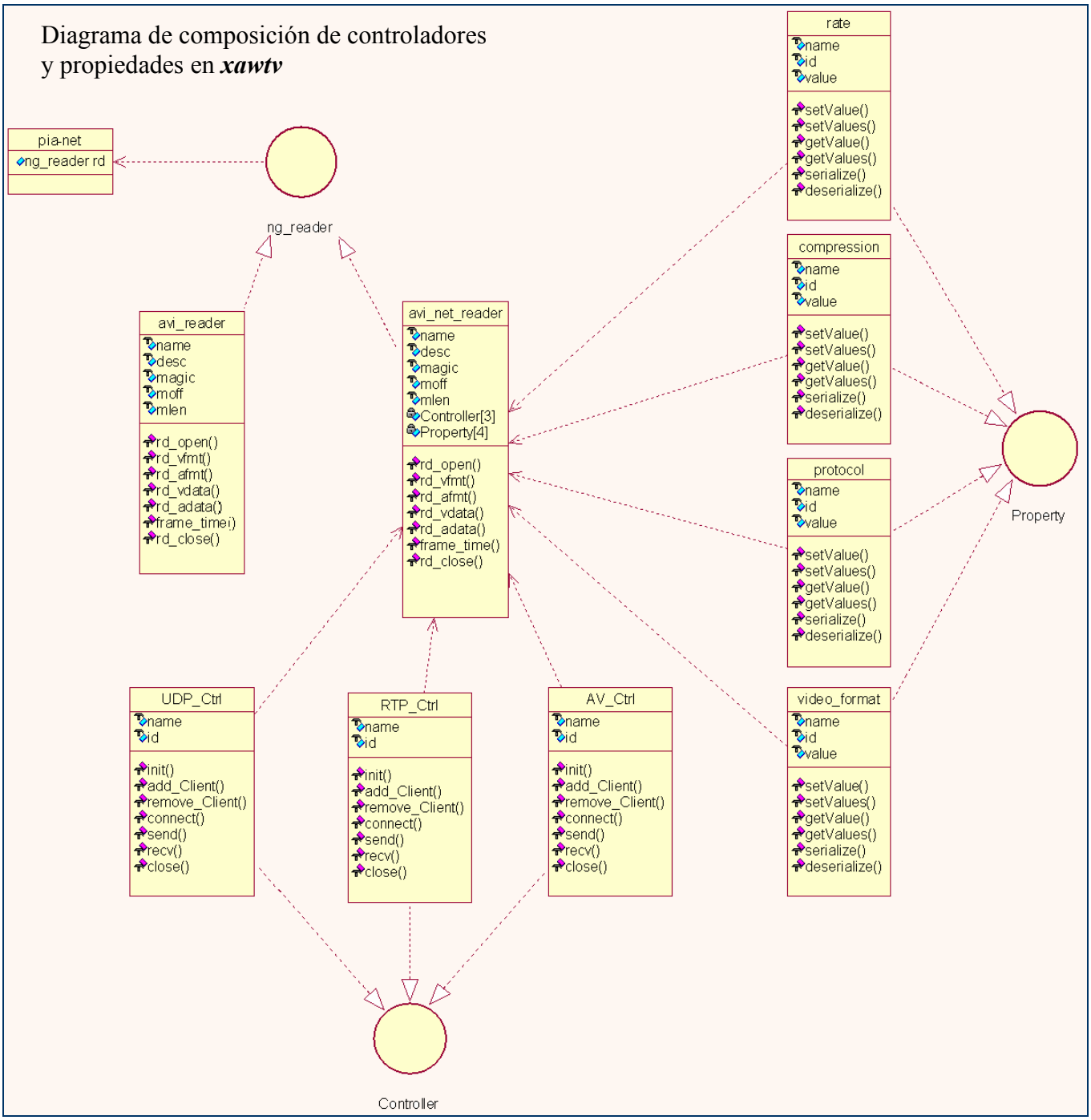
- **Confidencialidad**, la información sólo la ve el cliente al que va dirigida
- **Autentificación de mensajes**, quien manda el mensaje es quien realmente dice ser
- **Protección de respuesta**

Es ideal para proteger tráfico de voz sobre IP, o videoconferencias privadas, porque se caracteriza por ocupar poco caudal en la red (aumenta muy poco tamaño a cada paquete), por lo que no afecta a la calidad de servicio.

Existen librerías gratuitas SRTP en la página http://srtp.sourceforge.net/news.html, aunque aún se encuentran en sus primeras fases de desarrollo.

ANEXO I

Diagramas de *xawtv* y *pia-net*



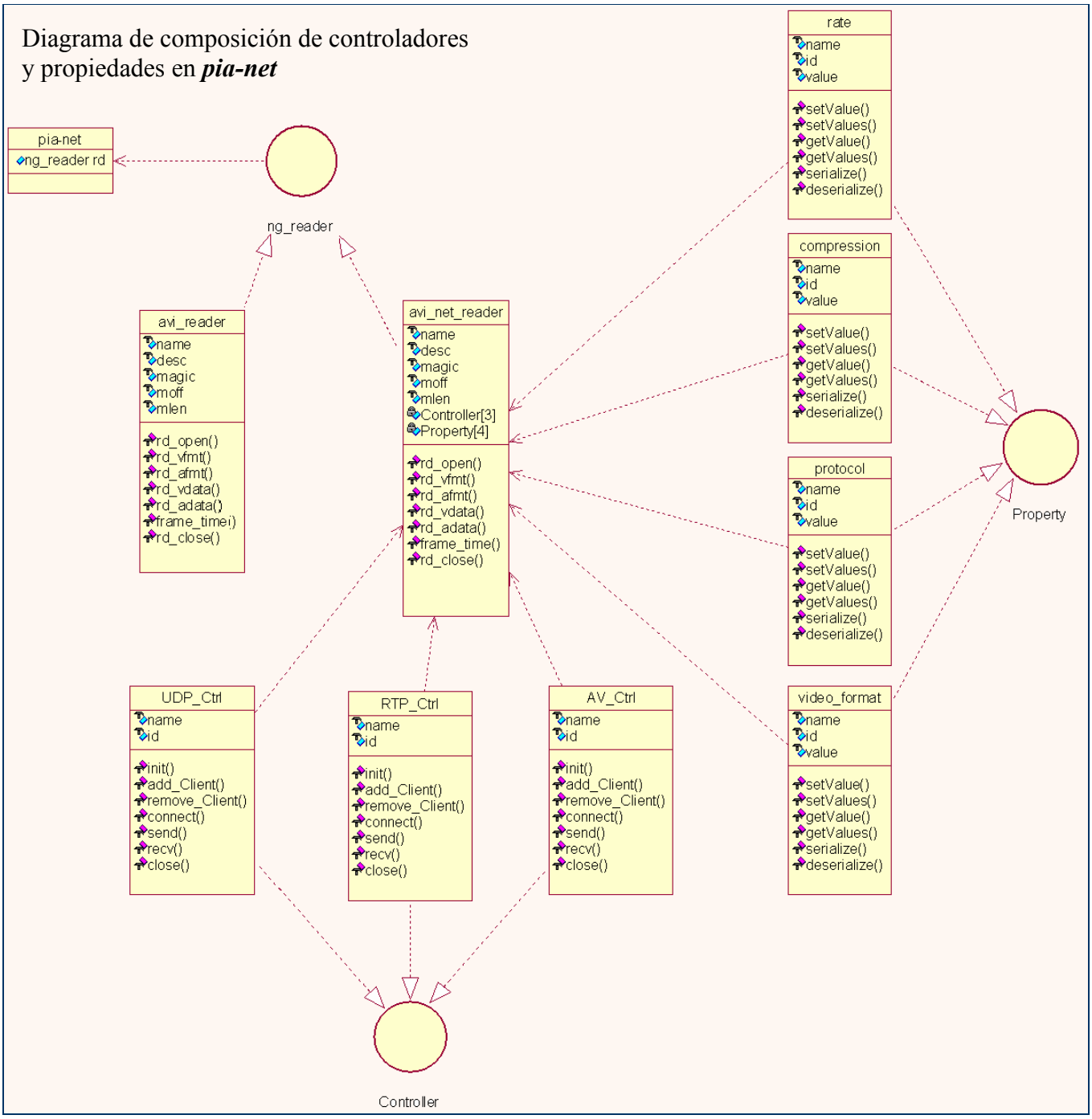
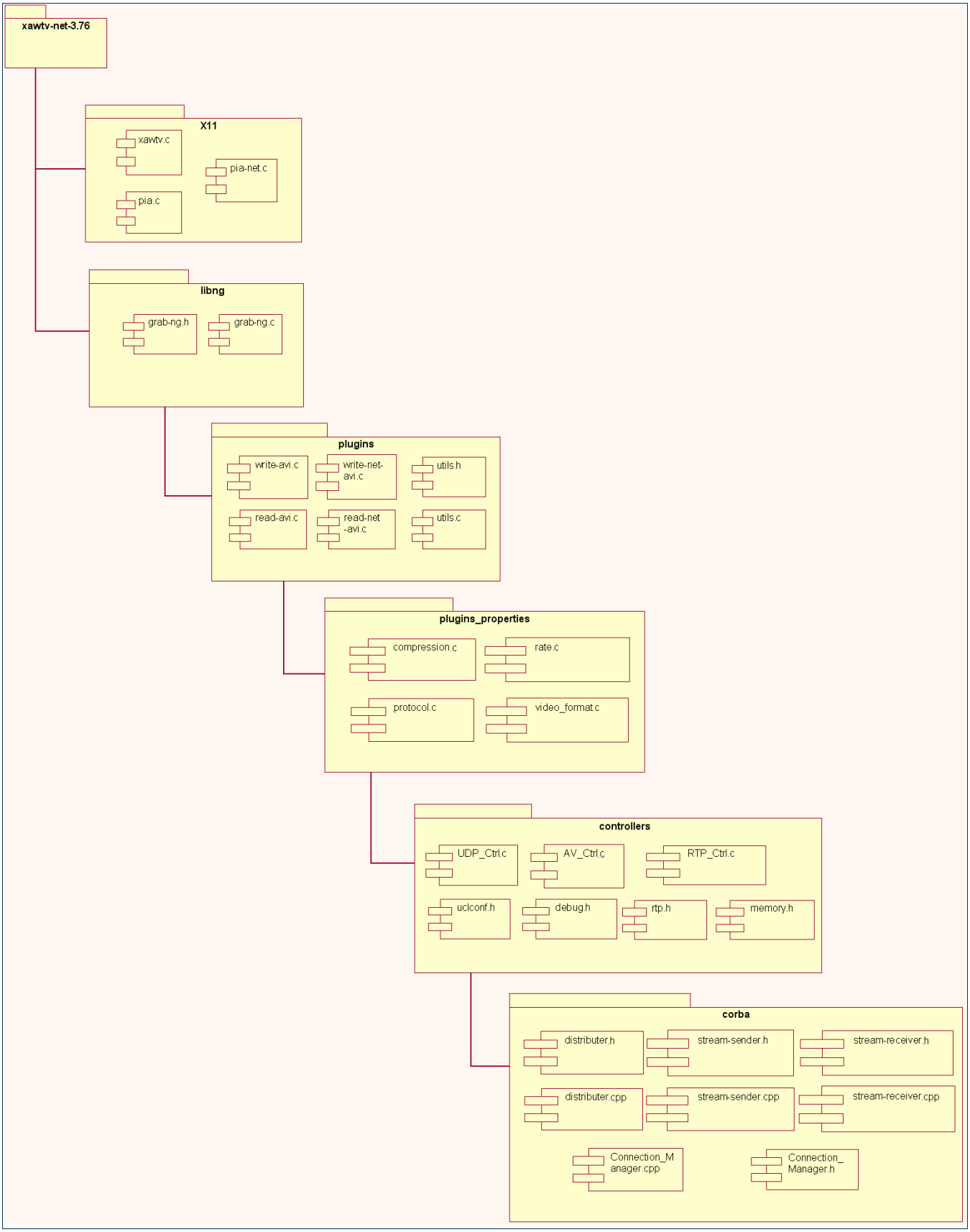
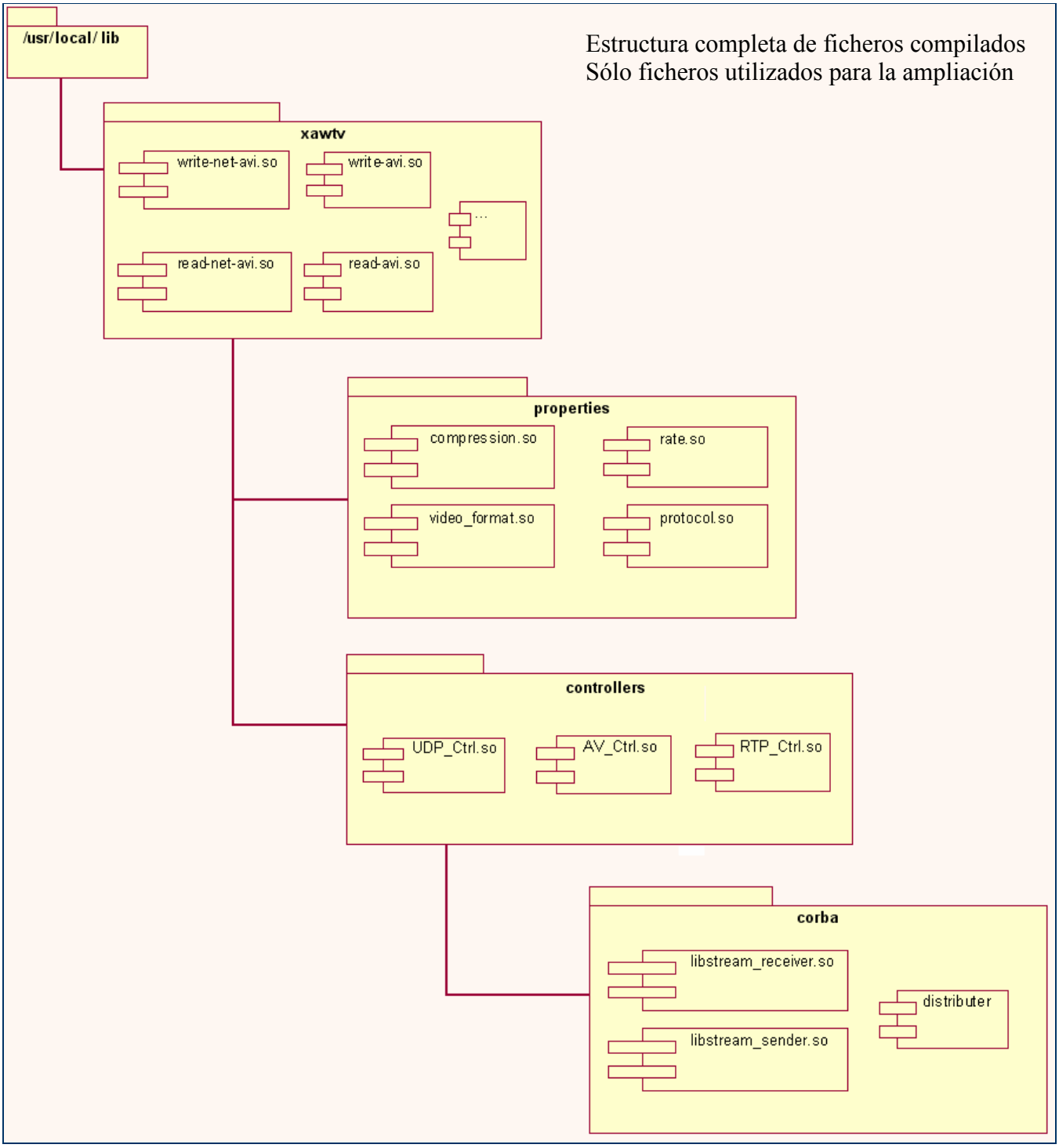


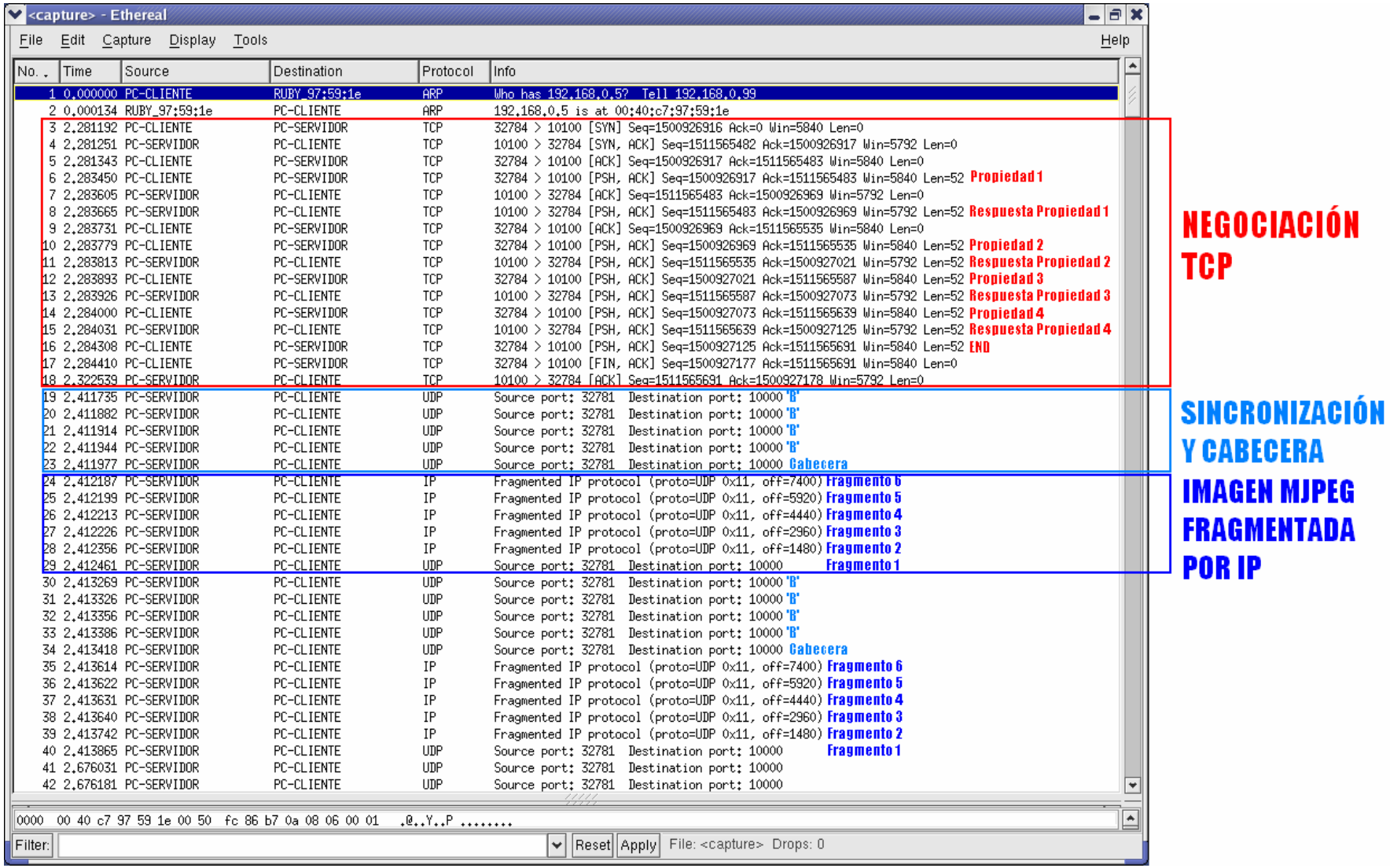
Diagrama de composición de controladores
y propiedades en ***pia-net***
pia-net
ng_reader rd
ng_reader
avi_reader
name
desc
magic
moff
mlen
rd_open()
rd_vfmt()
rd_afmt()
rd_vdata()
rd_adata()
frame_timei)
rd_close()
avi_net_reader
name
desc
magic
moff
mlen
Controller[3]
Property[4]
rd_open()
rd_vfmt()
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

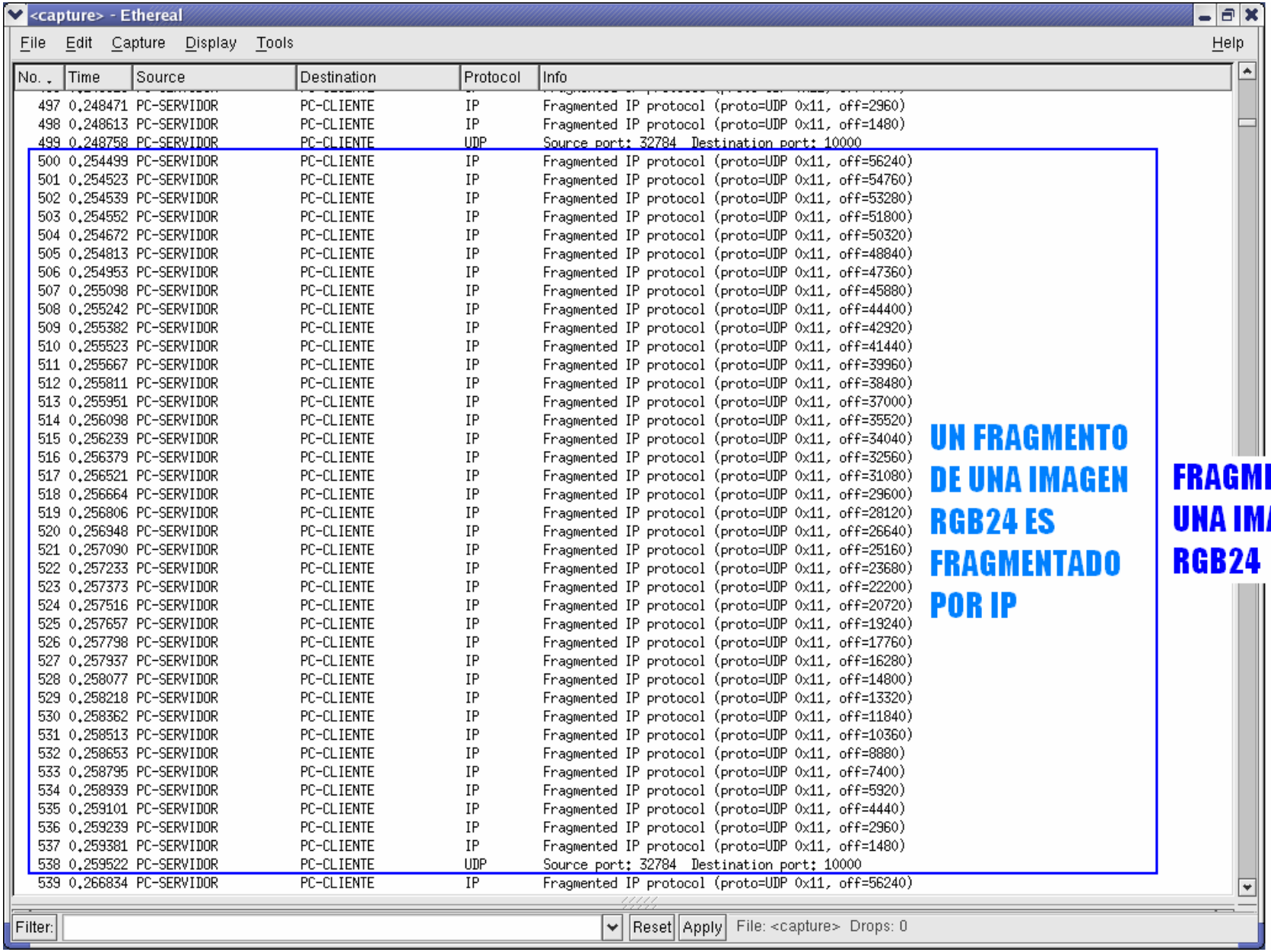

xawtv-net-3.76
X11
xawtv.c
pia-net.c
pia.c
libng
grab-ng.h
grab-ng.c
plugins
write-avi.c
write-net-avi.c
utils.h
read-avi.c
read-net-avi.c
utils.c
plugins_properties
compression.c
rate.c
protocol.c
video_format.c
controllers
UDP_Ctrl.c
AV_Ctrl.c
RTP_Ctrl.c
uclconf.h
debug.h
rtp.h
memory.h
corba
distributer.h
stream-sender.h
stream-receiver.h
distributer.cpp
stream-sender.cpp
stream-receiver.cpp
Connection_Manager.cpp
Connection_Manager.h

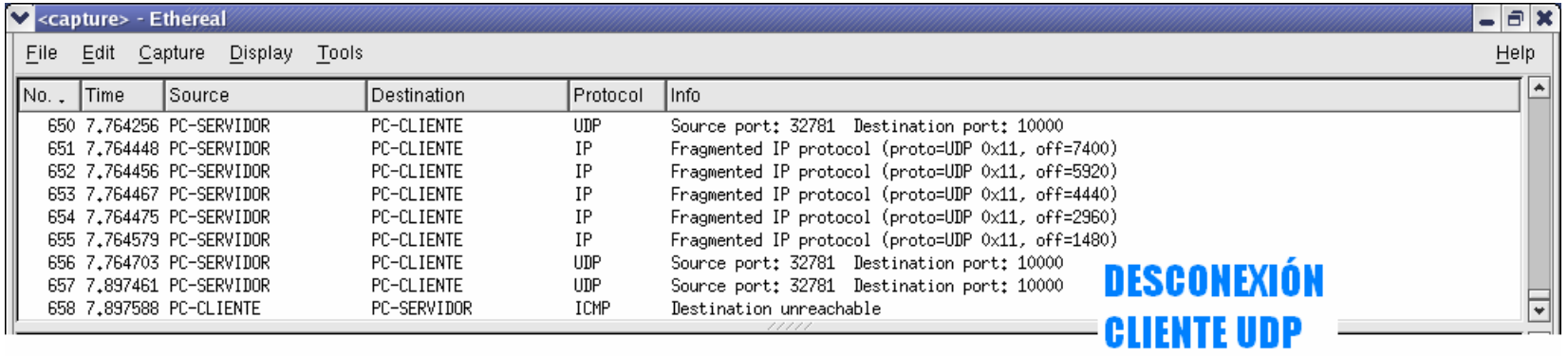

/usr/local/lib
Estructura completa de ficheros compilados
Sólo ficheros utilizados para la ampliación
xawtv
write-net-avi.so
write-avi.so
...
read-net-avi.so
read-avi.so
properties
compression.so
rate.so
video_format.so
protocol.so
controllers
UDP_Ctrl.so
AV_Ctrl.so
RTP_Ctrl.so
corba
libstream_receiver.so
distributer
libstream_sender.so

ANEXO II

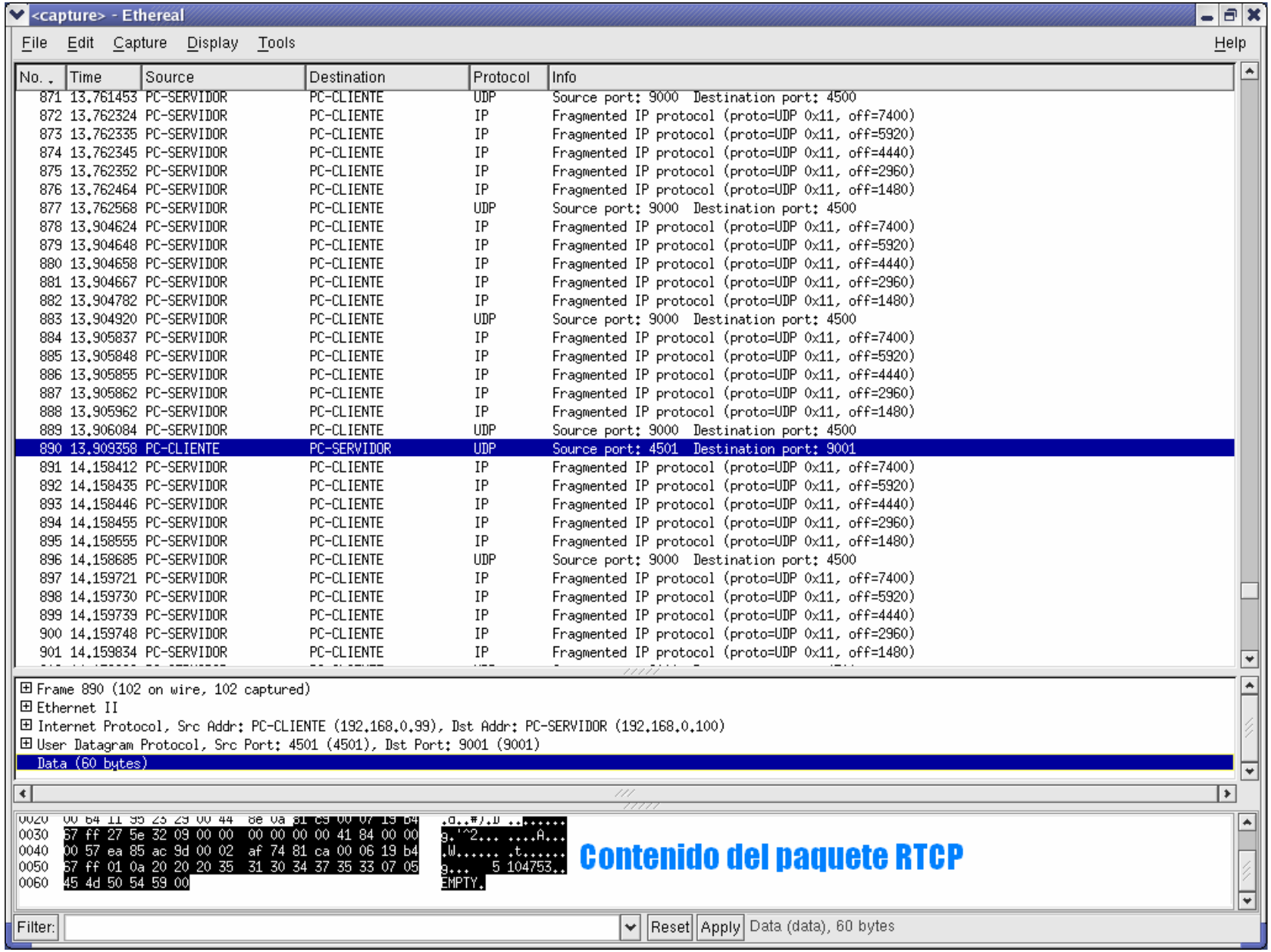
Capturas de red

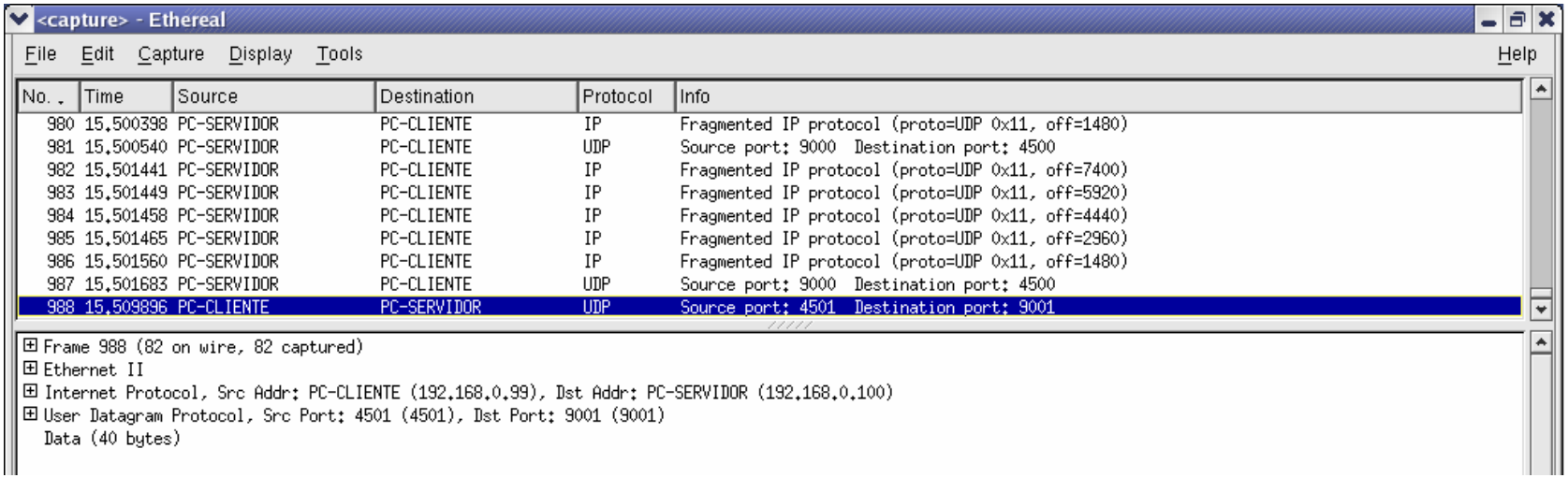



<capture> - Ethereal
File Edit Capture Display Tools Help
No. Time Source Destination Protocol Info
497 0.248471 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=2960)
498 0.248613 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=1480)
499 0.248758 PC-SERVIDOR PC-CLIENTE UDP Source port: 32784 Destination port: 10000
500 0.254499 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=56240)
501 0.254523 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=54760)
502 0.254539 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=53280)
503 0.254552 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=51800)
504 0.254672 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=50320)
505 0.254813 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=48840)
506 0.254953 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=47360)
507 0.255098 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=45880)
508 0.255242 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=44400)
509 0.255382 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=42920)
510 0.255523 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=41440)
511 0.255667 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=39960)
512 0.255811 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=38480)
513 0.255951 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=37000)
514 0.256098 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=35520)
515 0.256239 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=34040)
516 0.256379 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=32560)
517 0.256521 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=31080)
518 0.256664 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=29600)
519 0.256806 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=28120)
520 0.256948 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=26640)
521 0.257090 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=25160)
522 0.257233 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=23680)
523 0.257373 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=22200)
524 0.257516 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=20720)
525 0.257657 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=19240)
526 0.257798 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=17760)
527 0.257937 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=16280)
528 0.258077 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=14800)
529 0.258218 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=13320)
530 0.258362 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=11840)
531 0.258513 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=10360)
532 0.258653 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=8880)
533 0.258795 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=7400)
534 0.258939 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=5920)
535 0.259101 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=4440)
536 0.259239 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=2960)
537 0.259381 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=1480)
538 0.259522 PC-SERVIDOR PC-CLIENTE UDP Source port: 32784 Destination port: 10000
539 0.266834 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=56240)
Filter: Reset Apply File: <capture> Drops: 0
UN FRAGMENTO DE UNA IMAGEN RGB24 ES FRAGMENTADO POR IP
FRAGMENTO DE UNA IMAGEN RGB24



La negociación en RTP es idéntica a la mostrada en la primera figura, y las imágenes MJPEG también son fragmentadas por IP.


<capture> - Ethereal
File Edit Capture Display Tools Help
No. Time Source Destination Protocol Info
871 13.761453 PC-SERVIDOR PC-CLIENTE UDP Source port: 9000 Destination port: 4500
872 13.762324 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=7400)
873 13.762335 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=5920)
874 13.762345 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=4440)
875 13.762352 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=2960)
876 13.762464 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=1480)
877 13.762568 PC-SERVIDOR PC-CLIENTE UDP Source port: 9000 Destination port: 4500
878 13.904624 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=7400)
879 13.904648 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=5920)
880 13.904658 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=4440)
881 13.904667 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=2960)
882 13.904782 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=1480)
883 13.904920 PC-SERVIDOR PC-CLIENTE UDP Source port: 9000 Destination port: 4500
884 13.905837 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=7400)
885 13.905848 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=5920)
886 13.905855 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=4440)
887 13.905862 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=2960)
888 13.905962 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=1480)
889 13.906084 PC-SERVIDOR PC-CLIENTE UDP Source port: 9000 Destination port: 4500
890 13.909358 PC-CLIENTE PC-SERVIDOR UDP Source port: 4501 Destination port: 9001
891 14.158412 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=7400)
892 14.158435 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=5920)
893 14.158446 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=4440)
894 14.158455 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=2960)
895 14.158555 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=1480)
896 14.158685 PC-SERVIDOR PC-CLIENTE UDP Source port: 9000 Destination port: 4500
897 14.159721 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=7400)
898 14.159730 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=5920)
899 14.159739 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=4440)
900 14.159748 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=2960)
901 14.159834 PC-SERVIDOR PC-CLIENTE IP Fragmented IP protocol (proto=UDP 0x11, off=1480)
Frame 890 (102 on wire, 102 captured)
Ethernet II
Internet Protocol, Src Addr: PC-CLIENTE (192.168.0.99), Dst Addr: PC-SERVIDOR (192.168.0.100)
User Datagram Protocol, Src Port: 4501 (4501), Dst Port: 9001 (9001)
Data (60 bytes)
0030 67 ff 27 5e 32 09 00 00 00 00 00 00 41 84 00 00
0040 00 57 ea 85 ac 9d 00 02 af 74 81 ca 00 06 19 b4
0050 67 ff 01 0a 20 20 20 35 31 30 34 37 35 33 07 05
0060 45 4d 50 54 59 00
EMPTY.
Filter: Reset Apply Data (data), 60 bytes
PAQUETE DE CONTROL RTCP
Contenido del paquete RTCP



PAQUETE RTCP BYE

ANEXO III

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 fichero, 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 IV

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 V

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 [23].

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 hay que ejecutar:

```
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.

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] **OMG (Object Management Group)**
http://www.omg.org/

[3] **ACE TAO (Universidad de Washington)**
http://www.cs.wustl.edu/~schmidt/TAO.html

[4] **Directshow**
http://msdn.microsoft.com/library/en-us/directx9_c/directx/htm/directshow.asp

[5] **The Java Media Framework (JMF)**
http://java.sun.com/products/java-media/jmf/

[6] ***xawtv***
http://www.bytesex.org/xawtv

[7] **Red Hat Linux**
http://www.redhat.com

[8] **C Programming Language, The.** Kerninghan, Ritchie

[9] **"Object-Oriented Application Frameworks", Communications of the ACM, Vol. 40, No 10, October 1997.** M. Fayad, D.C. Schmidt

[10] **"Designs Patterns: Elements of Reusable Object Oriented Software", Addison Wesley, Reading Mass, 1995.** E. Gamma, R. Helm, R. Johnson, J. Vlissides

[11] **Manual GNU Make**
http://www.gnu.org/manual/make-3.80/make.html

[12] **Linux Complete Command Reference,** Compiled By J. Purcell

[13] **Computer Networks 4ª Edición, 2003.** Tanenbaum, Andrew S

[14] **Transmission Control Protocol**
http://www.ietf.org/rfc/rfc0793.txt

[15] **User Datagram Protocol**
http://www.ietf.org/rfc/rfc0768.txt

[16] **RTP: A Transport Protocol for Real-Time Applications (RTP, RTCP)**
http://www.ietf.org/rfc/rfc1889.txt

[17] **Audio Video transport**
http://www.ietf.org/html.charters/avt-charter.html

[18] **Real Media (rm)**
http://www.realnetworks.com

[19] **Advanced Streaming Format (asf)**
http://www.microsoft.com/windows/windowsmedia/format/asfspec.aspx

[20] **About RTP and the Audio-Video Transport Working Group. Relación completa de documentos, librerías, rfc y libros sobre RTP y RTCP**
http://www.cs.columbia.edu/~hgs/rtp

[21] **Common Multimedia Library**
http://www-mice.cs.ucl.ac.uk/multimedia/software/common/index.html

[22] **Java 2 Platform, Standard Edition**
http://java.sun.com/j2se

[23] **Librerías ACE-TAO**
http://deuce.doc.wustl.edu/Download.html