POSGRADO EN INFORMÁTICA

PROYECTO FIN DE MÁSTER

UNIVERSIDAD DE ALMERIA

ESCUELA POLITÉCNICA SUPERIOR

MÁSTER EN INFORMÁTICA INDUSTRIAL

TÉCNICAS DE CONTROL DE FLUJO DE DATOS DE UN CLIENTE DE STREAMING DE VÍDEO ESCALABLE EN EL TIEMPO

Curso 2011/2012

Alumno/a:

César Pardo Parra

Director/es:

Dr. Vicente González Ruiz

TRABAJO FIN DE MÁSTER
MÁSTER EN INFORMÁTICA INDUSTRIAL
POSGRADO EN INFORMÁTICA

TÉCNICAS DE CONTROL DE FLUJO DE DATOS DE UN CLIENTE DE STREAMING DE VÍDEO ESCALABLE EN EL TIEMPO

Por:
César Pardo Parra

Para la obtención del
Título del Master en Informática Industrial
Posgrado en Informática

Director:
Dr. Vicente González Ruiz

Almería, Junio 2012

ÍNDICE

I. Introducción 3

II. Los sistemas VoD 4

III. La codificación de vídeo escalable 4

- III-A. JPEG . . . 4
  - III-A1. RGB $\rightarrow$ YCbCr . . . 4
  - III-A2. Submuestreo de la crominancia . . . 4
  - III-A3. $[0, 255] \rightarrow [-128, 127]$ . . . 4
  - III-A4. 8x8 2D-DCT . . . 4
  - III-A5. Cuantificación . . . 4
  - III-A6. Codificación entrópica . . . 5
- III-B. Motion JPEG (MJPEG) . . . 5
- III-C. MPEG-1 . . . 5
  - III-C1. Overview del codec . . . 5
  - III-C2. Estimación de movimiento (ME) . . . 7
  - III-C3. El predictor ($\mathcal{P}$) . . . 9
  - III-C4. La transformada discreta del coseno ($\mathcal{T}$) . . . 9
  - III-C5. El cuantificador ($\mathcal{Q}^q$) . . . 9
  - III-C6. Codificación de las texturas residuo (VLC) . . . 9
  - III-C7. Codificación de los vectores de movimiento (VLC) . . . 9
  - III-C8. En controlador del bit-rate (BC) . . . 9
  - III-C9. El multiplexor de texturas/movimiento (Mux) . . . 9
  - III-C10. Distribución de los datos en MPEG-1 . . . 10
- III-D. MPEG-2 . . . 10
- III-E. MPEG-4 . . . 11
  - III-E1. Niveles y perfiles . . . 11
  - III-E2. Mayor precisión en la escalabilidad de la granularidad . . . 11
- III-F. H.264/AVC . . . 11
  - III-F1. Diferencias con los codecs anteriores . . . 11
  - III-F2. Perfiles . . . 12
- III-G. H.264/SVC . . . 12
  - III-G1. Diferencias con los codecs anteriores . . . 13
  - III-G2. Escalabilidad espacial . . . 13
  - III-G3. Escalabilidad en calidad . . . 13
  - III-G4. Número de capas . . . 14
- III-H. JPEG 2000 (J2K) . . . 14
- III-I. Motion JPEG 2000 (MJ2K) . . . 15
- III-J. Motion Compensated Temporal Filtering (MCTF) . . . 15
- III-K. Motion Compensated JPEG 2000 (MCJ2K) . . . 16
  - III-K1. El papel de MCTF en MCJ2K . . . 16
  - III-K2. MCTF en MCJ2K . . . 16
  - III-K3. Escalabilidad temporal, espacial y en calidad . . . 17
  - III-K4. Organización de stream . . . 18

IV. Arquitectura cliente/servidor para el streaming de vídeo 18

- IV-A. La arquitectura cliente/servidor . . . 18
- IV-B. Hilos, servidores concurrentes e iterativos . . . 19
- IV-C. La pila de protocolos TCP/IP . . . 19
- IV-D. Código Implementado . . . 20

V. Propuesta 21

VI. Evaluación 30
VI-A. Resultados de recepción con 6 capas 30
VI-B. Resultados de recepción con 5 capas 32
VI-C. Resultados de recepción con 4 capas 32
VI-D. Resultados de recepción con 3 capas 32
VI-E. Resultados de recepción con 2 capas 32
VI-F. Resultados de recepción con 1 capas 32
VI-G. Comparación de resultados con 100 KBs en 1 capa, 3 capas y 6 capas. 32
VI-H. Comparación de resultados con 600k en 1 capa, 3 capas y 6 capas. 32

VII. Conclusiones y trabajo futuro 33

Apéndice A: Generación de una imagen nula 33

Apéndice B: Generación de campo de movimiento nulo 33

Apéndice C: Resultados de la recepción de un vídeo comprimido con 5 capas 34

Apéndice D: Resultados de la recepción de un vídeo comprimido con 4 capas 34

Apéndice E: Resultados de la recepción de un vídeo comprimido con 3 capas 34

Apéndice F: Resultados de la recepción de un vídeo comprimido con 2 capas 36

Apéndice G: Resultados de la recepción de un vídeo comprimido con 1 capas 37

Apéndice H: Comparación de resultados con un ancho de banda de 100kBs 39

Apéndice I: Comparación de resultados con un ancho de banda de 600kBs 39

Apéndice J: Temporización del trabajo realizado 39

Bibliografía 39

Técnicas de control de flujo de datos en un cliente de streaming de vídeo escalable en el tiempo

Cesar Pardo Parra
Vicente González Ruiz*

***Resumen*—En la actualidad, para la reproducción de un vídeo a través de Internet es necesario tener que parar el vídeo para darle tiempo para poder reproducir parte del vídeo sin tener que se pause. Con este proyecto fin de carrera lo que se quiere conseguir es diseñar un servicio de streaming de vídeo escalable donde el cliente va a ver una reproducción de dicho vídeo acorde al ancho de banda que este disponga. El lenguaje implementado para conseguir esto ha sido C.**

***Palabras Clave*—MCTF, escabilidad en el tiempo, control de capas, YUV, frame_types, motion_residue, high.**

***Abstract*—Today, for playing a video over the Internet you must have to stop the video to give you time to play part of the video without having to pause. With this final project what we want to achieve is to design a video streaming service scalable where the customer will see a reproduction of the video according to the bandwidth that this provides. The language implemented to achieve this has been C.**

***Index Terms*—MCTF, scalable time, layer management, YUV, frame_types, motion_residue, high.**

I. INTRODUCCIÓN

DESDE la invención de la red Internet a finales de los años sesenta, los diseñadores y usuarios de la red han intentado utilizarla para los más diversos propósitos. Inicialmente, sólo aplicaciones como la transmisión de ficheros, el correo electrónico o el chat estaban disponibles. Como se puede comprobar, incluso desde estos tiempos tan iniciales ya podían distinguirse dos tipos de aplicaciones: (1) aquellas, como el correo electrónico, en las que las restricciones temporales impuestas por la propia aplicación demandan un uso del ancho de banda muy bajo y flexible, y otras (2), como son la transmisión del ficheros o el chat, donde o es imprescindible disfrutar de una cantidad suficiente de ancho de banda para transmitir dichos ficheros rápidamente o además, como ocurre con el chat, el requerimiento más importante radica en que la latencia de la red debe ser lo más pequeña posible.

Para acomodar a estos dos tipos de tráfico existen dos filosofías básicas: (1) distinguir entre ambos tipos de flujos de datos (streams), tratándoles de forma diferente (fundamentalmente, dedicando mucho más ancho de banda aunque sea de forma puntual a las aplicaciones de la segunda clase) y (2), no diferenciarlas, y esperar que con la evolución de la propia red (que prácticamente ha doblado su tamaño y capacidad cada año desde su nacimiento) el servicio ofrecido sea suficientemente bueno. Por este motivo se han creado un número considerable de tecnologías de transmisión, algunas especialmente concebidas para transmitir flujos de datos multimedia como puede ser ATM (Asynchronous Transfer Mode), y otras más pensadas para transmitir datos en general, de las cuales sin duda la más exitosa a sido Ethernet.

Actualmente ambos tipos de tecnologías coexisten; las especializadas en flujos multimedia y las no especializadas, aunque en redes para usos diferentes debido fundamentalmente a motivos económicos. El resultado es que, para transmitir datos (y hoy en día en la Internet pública no se realiza un tratamiento diferente de los datos en función de su contenido) no se puede conocer qué cantidad de ancho de banda vamos a poder utilizar cuando transmitimos. Cuando enviamos un correo electrónico esto va a provocar que el tiempo que tarda el correo en llegar hasta el receptor pueda variar unos pocos segundos. Sin embargo, cuando enviamos datos sensibles al tiempo, como puede ser una conversación telefónica o un vídeo, podría ocurrir que el ancho de banda ofrecido en ese momento por la red fuera insuficiente. En este último caso, la única solución válida a este problema consiste en precargar una cantidad de vídeo concreta en los receptores antes de comenzar la reproducción. Dicho tiempo de precarga lo calcula el receptor que, en función del bit-rate del vídeo, su duración y del bit-rate de la red estima la cantidad de datos que deben precargarse para que, en el caso de que ambos bit-rates se mantengan más o menos durante todo el tiempo que dura el vídeo, este pueda reproducirse sin pausa.

En este trabajo se presenta una solución a la transmisión de vídeo bajo demanda (en inglés, Video on Demand o VoD) sobre la red Internet. Para ello se plantea una arquitectura cliente/servidor y se transmite un tipo de vídeo comprimido especialmente diseñado para minimizar la precarga de datos por parte del usuario. Se estudian la posibilidades de un sistema VoD que utiliza un codificador de vídeo escalable que permite maximizar la calidad del vídeo reproducido (en concreto, el número de imágenes por segundo de la reproducción) en función de bit-rate que existe en ese momento disponible para el usuario en la red. El resultado es un sistema de visualización remota de vídeos con una muy baja latencia inicial, ya que apenas existe precarga, y una alta resistencia a los cortes de la reproducción.

El resto del documento se organiza de la siguiente manera. En la Sección II se describe el funcionamiento básico de los sistemas de streaming de vídeo bajo demanda. La Sección III está dedicada a mostrar los fundamentos del codificador de vídeo escalable usado. La Sección IV se analizan las printipales propiedades de los sistemas cliente/servidor. En la Sección V se presenta la solución aportada en este trabajo fin de máster. En la Sección VI se realiza una evaluación de dicha

*Departamento de Arquitectura de Computadores y Electrónica de la Universidad de Almería.

propuesta. Finalmente, la Sección VII expone las principales conclusiones a las que se han llegado y líneas de trabajo futuro más interesantes.

II. Los sistemas VoD

En un sistema VoD el usuario selecciona un vídeo (generalmente previamente almacenado y comprimido, evidentemente, junto con su audio asociado) que está almacenado en un servidor remoto, y lo visualiza (previa transmisión) de la misma manera que si estuviera almacenado en su reproductor de vídeo local, esto es, iniciando, deteniendo, rebobinando hacia delante y o hacia detrás, y pudiendo acceder a cualquier instante de tiempo del vídeo cuando lo desea. Generalmente los tiempos de retardo en cada una de estas acciones son de unos pocos segundos.

Los primeros sistemas VoD aparecieron a principios de los años noventa y estaban implementados sobre redes dedicadas, especialmente sobre ATM. Hoy en día existen multitud de servidores VoD, algunos para contenidos de pago (como puede ser MegaVideo) y otros con contenidos de acceso libre. Entre estos últimos, el más famoso sin duda alguna es YouTube.

Cuando un usuario utiliza su navegador Web (parte cliente) para visualizar un vídeo almacenado en YouTube (parte servidora), el cliente realiza una petición al servidor que, mediante HTTP (HyperText Transfer Protocol), finaliza en la transmisión interactiva de una secuencia de vídeo comprimido.

Aunque el servidor probablemente ofrezca diferentes versiones del mismo vídeo, con distintas resoluciones espaciales, el navegador precarga una cierta cantidad de datos antes de comenzar la reproducción con el objetivo anteriormente mencionado: reducir o eliminar las pausar provocadas por la falta de ancho de banda. Si dicha precarga es insuficiente, el usuario tiene solamente una opción: detener la reproducción actual e intentar de nuevo con una versión de menor resolución (que seguramente tendrá un bit-rate de codificación menor). Como es evidente, dicha técnica es sólo una solución parcial ya que el número de resoluciones no suele ser superior a dos o tres, con lo que no existe en realidad un número alto de posibilidades para acomodar el bit-rate de codificación al bit-rate de la red (ancho de banda).

En este trabajo se plantea el uso de un compresor de vídeo que ofrece muchas más posibilidades a la hora de encajar ambos bit-rates. Además, la conmutación entre dichas posibilidades se realiza de forma transparente para el usuario. En concreto, el usuario recibe de un vídeo cuya resolución temporal (el número de imágenes por segundo) depende de la capacidad instantánea de canal de transmisión. Cuando el número de imágenes/segundo se reduce, el reproductor interpola las imágenes que faltan para que la sensación de movimiento percibida no se vea afectada.

III. La codificación de vídeo escalable

Describiremos el sistema de codificación de vídeo escalable a partir de sucesivas mejoras de codecs de vídeo estándar.

III-A. JPEG

El estándar de compresión de imágenes JPEG [1] (Joint Photographic Experts Group) fué desarrollado por la ISO en 1992 [2] y es el compresor de imágenes lossy por excelencia. No en vano está soportado por casi todos los dispositivos de tratamiendo multimedia. El gran motivo de su éxito radica en que el codec JPEG está especialmente diseñado para comprimir imágenes a color a una tasa de 1 bits/pixel (bpp) sin que un humano sea capaz normalmente de diferenciar la imagen comprimida de la original.

EL algoritmo de compresión consiste básicamente en lo siguiente:

1. Convertir la imagen al dominio YCbCr.
2. Submuestrear la crominancia.
3. Desplazar cada componente al rango $[-128, 127]$.
4. Para cada componente (Y, Cb y Cr):
   a) Aplicar la (8×8)-DCT a cada componente.
   b) Cuantificar los coeficientes DCT.
   c) Codificar entrópicamente los coeficientes cuantificados.

A continuación iremos describiendo cada una de estas fases con mayor detalle:

III-A1. RGB → YCbCr: Consiste en cambiar del dominio de color RGB al YCbCr. Esto se realiza mediante la transformada [3]

$$\begin{pmatrix} \text{Y} \\ \text{Cb} \\ \text{Cr} \end{pmatrix} = \begin{pmatrix} 0,299 & 0,587 & 0,144 \\ -0,1687 & -0,3313 & 0,5 \\ 0,5 & -0,4187 & -0,0813 \end{pmatrix} \begin{pmatrix} \text{R} \\ \text{G} \\ \text{B} \end{pmatrix}. \tag{1}$$

III-A2. Submuestreo de la crominancia: El sistema visual humano es más sensible a la luminancia (Y) que a la crominancia (CbCr). Por tanto, la información sobre el color de una imagen puede submuestrearse sin que sea perceptible. El patrón de submuestreo más frecuentemente usado es el 4:2:2 (véase la Figura 1).

III-A3. $[0, 255] \rightarrow [-128, 127]$: Las muestras se desplazan al rango $[-128, 127]$ con el objetivo de que la media sea 0. Esto es necesario para que el rango dinámico de la DCT (Discrete Cosine Transform) sea mínimo.

III-A4. 8x8 2D-DCT: Las muestras $I[n]$ son trasformadas usando la DCT que responde a:

$$\text{DCT}[u] = \frac{\sqrt{2}}{\sqrt{N}} K(u) \sum_{n=0}^{N-1} I[n] \cos \frac{(2n+1)\pi u}{2n}, \tag{2}$$

donde

$$K(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{si } u = 0 \\ 1 & \text{if } u > 0. \end{cases} \tag{3}$$

La DCT es separable lo que significa que la 2D-DCT se calcula aplicando la DCT primero a las filas de la imagen y luego a las columnas (o viceversa).

III-A5. Cuantificación: Los coeficientes DCT se cuantifican con el objeto de eliminar fundamentalmente el ruido de la imagen. Esta fase es similar a un proceso de filtrado en que se eliminan las altas frecuencias de la imagen.

Luminance							
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Chrominance							
17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Figura 2. Matrices de cuantificación propuestas por el JPEG.

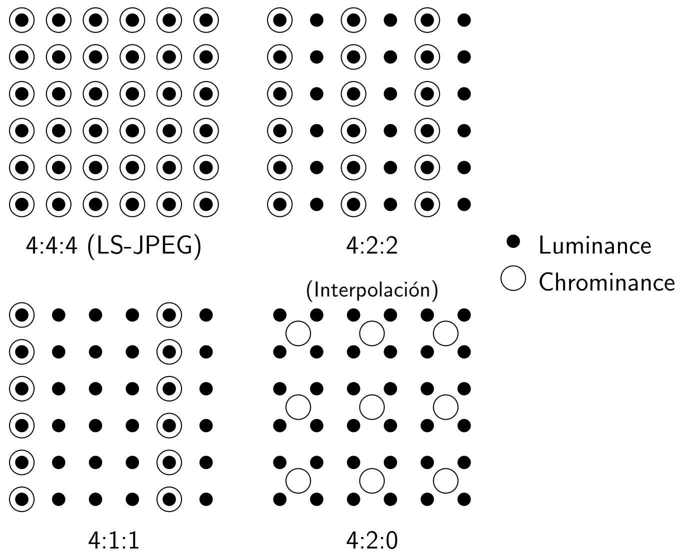

Figura 1. Patrones de sumuestreo de la crominancia usado en los estándares ISO.

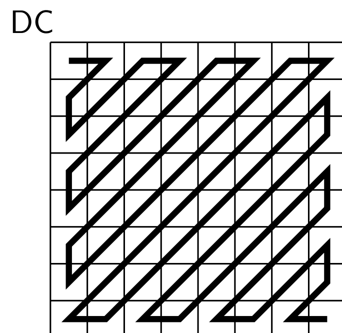

Figura 3. Recorrido en zig-zag de los coeficientes AC en JPEG.

El JPEG propone usar las matrices de cuantificación que se muestran en la Figura 2. Este proceso queda descrito por la ecuación

$$\text{2D-DCT}'[u,v] = \text{round}\Big(\frac{8\times 8\text{-DCT}[u,v]}{\text{Z}[u,v]}\Big), \quad (4)$$

donde $\text{Z}[\cdot,\cdot]$ es la correspondiente matriz de cuantificación.

III-A6. Codificación entrópica: Finalmente los coeficientes DCT cuantificados se comprimen usando un código de Huffman. Más concretamente:

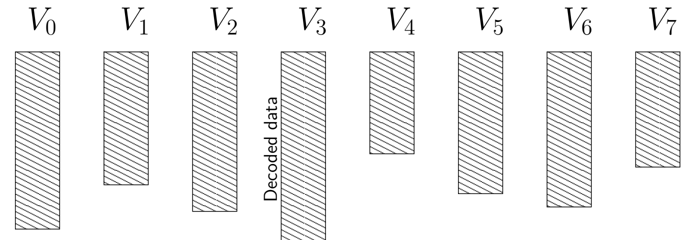

Figura 4. Un ejemplo de las longitudes asociadas a cada imagen en MJPEG.

1. Eliminar al el coeficiente DC de la loseta actual el coeficiente DC de la anterior loseta siguiendo un recorrido de tipo *raster*. Esto elimina la correlación entre bloques.
2. Los coeficientes AC de recorren en *zig-zag* (véase la Figura 3). Este recorrido responde a un incremento en las componentes de frecuencia.
3. Los runs generados por el recorrido anterior se comprimen usando un código de longitud variable que asigna a aquellos runs más frecuentes códigos de longitud menor.

III-B. Motion JPEG (MJPEG)

El estándar de compresión de vídeo MJPEG [4] es básicamente una aplicación del estándar de compresión de imágenes JPEG [5].

Un aspecto importante del codec MJPEG es que a calidad constante el tamaño de cada imagen puede ser diferente dependiendo del contenido visual de las mismas (véase la Figura 4.

III-C. MPEG-1

III-C1. Overview del codec: El estándard MPEG-1 [6] (formalmente llamado *Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s* o *Estándar ISO/IEC 11172*) define cómo debe descomprimirse un bit-stream MPEG-1, consiguiendo una tasa de bits de aproximadamente 1.5 Mbit/s (lo que se corresponde con uno ratio de compresión de aproximadamente 180:1), para una señal de 25 imágenes por segundo con una profundidad de color de 24 bits y un tamaño por imagen de 640x480 puntos.

Un compresor MPEG-1 es un sistema híbrido que combina un codec lossy-DPCM aplicado a las imágenes de la secuencia de vídeo y un compresor de imágenes semejante al usado en JPEG. Si observamos la Figura 5 podemos ver que se

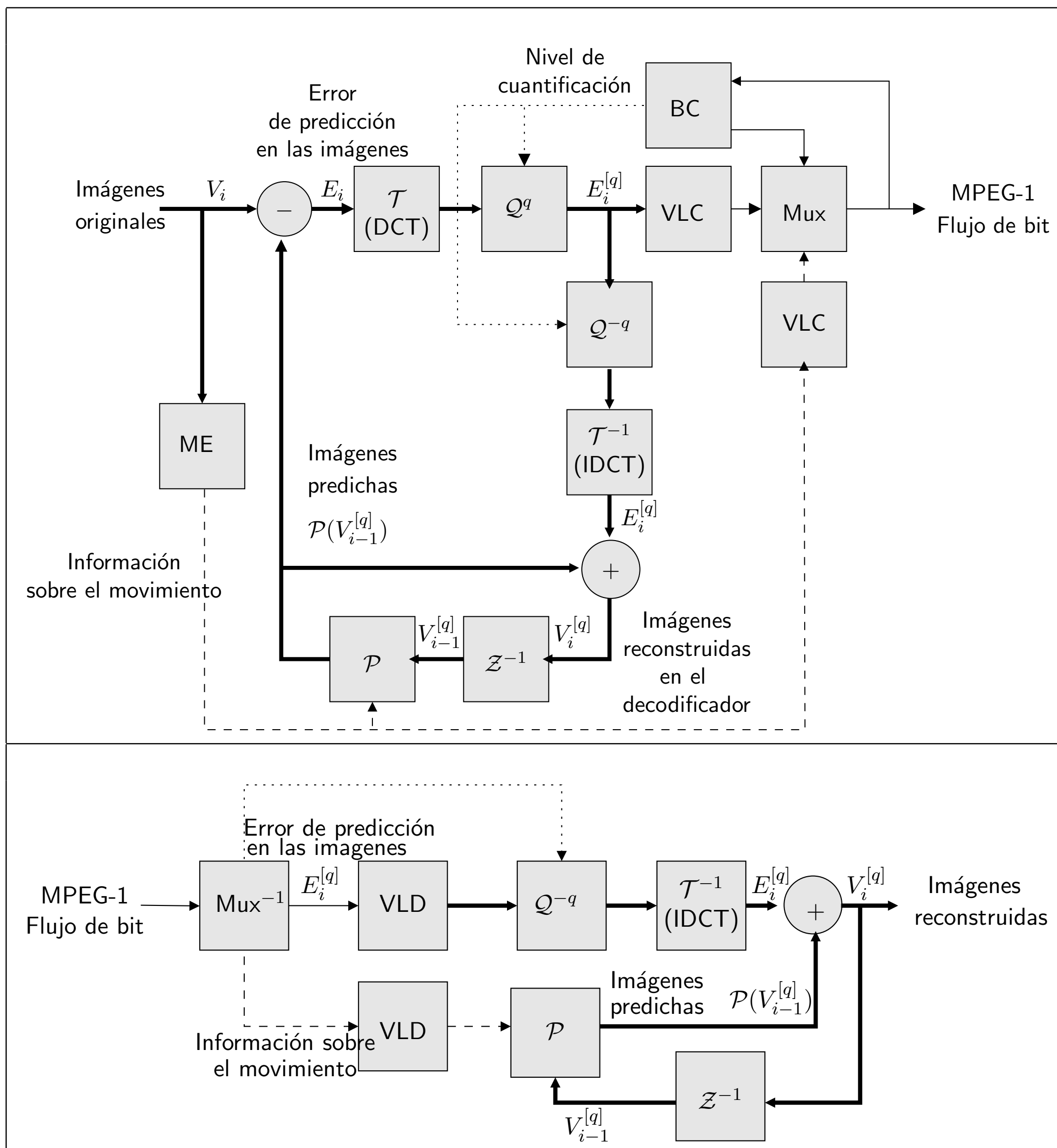

Figura 5. El codec MPEG-1. Arriba el compresor y debajo el descompresor.

trata básicamente de un codec lossy-DPCM de orden 1 y que elimina la redundancia temporal mediante estimación de movimiento, en inglés, Motion Estimation (ME). Como puede apreciarse, para cada imagen de entrada V_i se genera una imagen residuo $E_i^{[q]}$, cuya compresión entrópica usando codificación de longitud variable, en inglés Variable Length Coding (VLC) genera, junto con la información sobre el movimiento detectado en el vídeo por el estimador de movimiento el code-stream MPEG-1.

Por otra parte, tal y como puede apreciarse en la parte inferior de la Figura 5, el descompresor utiliza dicho residuo junto con la imagen previamente reconstruida para recuperar una versión aproximada $V_i^{[q]}$ de la secuencia original. En todas estas expresiones, el superíndice $[q]$ indica el nivel de cuantificación utilizado por el cuantificador $\mathcal{Q}^q$. Dicha cuantificación reduce la precisión numérica de los coeficientes de la transformada espacial $\mathcal{T}$ que en este caso se trata de la DCT (Discrete Cosine Transform, aplicada en losetas de 8x8 píxeles) de las imágenes residuo E_i resultantes de restar a la imagen actual la imagen predicción generada por la imagen anterior que ha sido reconstruida por el descompresor. En otras palabras,

$$E_i = V_i - \mathcal{P}(V_{i-1}^{[q]}) \tag{5}$$

donde $\mathcal{P}(V_{i-1}^{[q]})$ es la predicción usando estimación de movimiento para generar la imagen V_i a partir de la imagen $V_{i-1}^{[q]}$, que es la última imagen generada por el descompresor. Las imágenes

$$V_i^{[q]} = \mathcal{T}^{-1}\big(\mathcal{Q}^{-q}(E_i^{[q]})\big) + \mathcal{P}(V_{i-1}^{[q]}) \tag{6}$$

donde

$$E_i^{[q]} = \mathcal{Q}^q(E_i). \tag{7}$$

Tal y como se ha descrito, MPEG-1 es un compresor de vídeo lossy debido a la etapa de cuantificación $\mathcal{Q}$, que aplica un nivel de cuantificación generalmente variable y que dependiente del bit-rate instantáneo producido. Dicho nivel

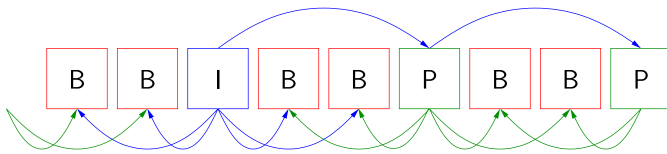

Figura 6. Dependencia de imágenes I, B y P.

de cuantificación es controlado por el módulo BC (Bit-rate Control) y la información resultante de dicha estimación del nivel de cuantificación se incluye, junto con el residuo de las texturas cuantificadas ($E_i^{[q]}$) y la información sobre el movimiento, en el bit-stream que el compresor produce y consume el descompresor.

III-C2. Estimación de movimiento (ME): Consiste en la eliminación de la redundancia temporal, es decir, la información que se repite en una secuencia de imágenes. Se trata de localizar regiones en las imágenes que se mantengan invariables en posteriores imágenes de la secuencia de vídeo, de manera que sólo es necesario almacenar la diferencia entre dichas imágenes, ya que el resto es igual y no es necesario almacenarlo repetidamente. Esta diferencia entre la imagen anterior y posterior o posteriores en la secuencia, es codificada en lo que se denomina una imagen de predicción. Es evidente, que el proceso de compresión en la estimación de movimiento se lleva a cabo una vez para cada una de esas regiones invariables, sin embargo, la descompresión se hará tantas veces como dicha región aparezca en las imágenes posteriores de la secuencia.

Dependiendo de la manera en la que son construidas estas predicciones, en MPEG-1 se encuentran los siguientes tipos de codificación de imágenes:

1. **Intra-coded:** Una imagen intra (I) es aquella en la que su codificación depende únicamente de la propia imagen y de ninguna otra. Esto es, no hay necesidad de llevar a cabo una estimación del movimiento, y por tanto, no contiene información sobre vectores de movimiento.
2. **Predictive-coded:** Las imágenes denominadas P, son aquellas cuya codificación depende de *una* imagen previa (en la secuencia de imágenes). Esta imagen de la que depende P puede ser otra imagen P o una imagen I.
3. **Bidirectionally predictive-coded:** Al igual que una imagen P, la dependencia de las B puede depender de imágenes I o P, con la salvedad de que una imagen B depende de *dos* imágenes, no sólo de una. Por tanto, la dependencia es bidireccional, es decir, depende de una imagen anterior y otra posterior, tomando en cuenta la secuencia de imágenes.

En la Figura 6 se exponen las distintas dependencias entre los tres tipos de imágenes, que se generan al codificar una imagen a partir de otra u otras. En dicha figura se representan las imágenes tipo I, P y B y sus dependencias en los colores azul, verde y rojo, respectivamente.

En todos los compresores de vídeo existe el concepto de GOP (Group Of Pictures) que no es más que la forma en que las imágenes del vídeo se descorrelacionan. En la mayoría de los GOP's la primera imagen es intra-codificada y el resto, cada una depende de la anterior (P) o de dos imágenes, una anterior y otra posterior (B). En la Figura 6 se muestra un ejemplo de GOP BBIBBPBBP, lo que significa que el resto del vídeo está codificado siguiendo un patrón que se forma repitiendo dicho GOP tantas veces como sea necesario. En la Figura 7 se presentan otros ejemplos diferentes de GOP.

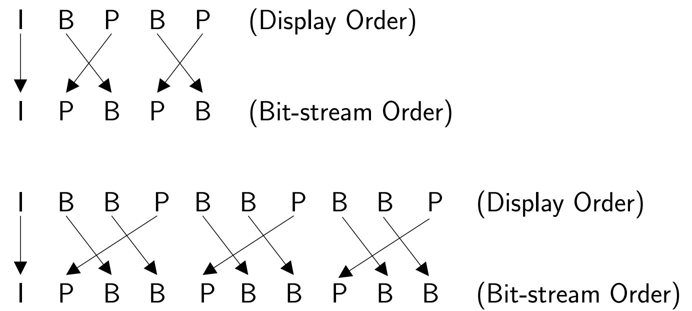

Figura 7. Ejemplos de ordenación de imágenes en un GOP.

Como puede deducirse de este patrón de descorrelación temporal, para la descompresión de una imagen $V_i^{[q]}$, puede ser necesario, por ejemplo, descomprimir la imagen previa, la $V_{i-1}^{[q]}$, y así sucesivamente hasta que se alcance una imagen I. Evidéntemente, esto no es lo ideal cuando, por ejemplo, queremos descomprimir única y directamente la imagen $V_i^{[q]}$. Además, si una imagen con la que existe una dependencia temporal ha sido perdida por algún error de almacenamiento o transmisión, las imágenes que dependen de ella carecen de toda la información necesaria para ser descomprimidas y, por tanto, poder ser visualizadas. Siendo conscientes de esta limitación que impone el sistema, en necesidad de un mayor número de dependencias conforme mayor ratio de compresión se desee, es evidente que es necesario adoptar un compromiso entre ambos.

En este contexto, las imágenes son comprimidas como un todo, denominando a esta unidad GOP. Nótese que el concepto de GOP funciona del mismo modo usando imágenes P, como imágenes B. Cuando son frames (imágenes de la secuencia) tipo B en un GOP, podemos hablar sobre diferentes tipos de ordenación de las imágenes (en la Figura 7 se presentan algunos ejemplos):

1. **Display order:** Es el orden usado para mostrar las imágenes, es decir, el orden en que la secuencia de imágenes forma el vídeo al sucederse.
2. **Bit-stream order:** Es el orden en el cual las imágenes son almacenadas.

Por otra parte, en relación a la forma en que se diseñan los GOPs, en una secuencia MPEG-1 se pueden encontrar dos tipos de GOP:

1. **Cerrado:** Un GOP cerrado no necesita de otros GOP para ser descomprimido. Necesariamente, es aquel en el que su primera imagen es tipo intra.
2. **Abierto:** Por otra parte, un GOP abierto indica que él mismo no tiene toda la información para descomprimir las imágenes que contiene, dependiendo de este modo, de otros. Los GOP abiertos son muy poco usuales puesto que no permiten un acceso aleatorio a las imágenes de la secuencia, ni soportan mecanismos de control de errores.

$$\begin{array}{|lll} \mathsf{A} & X_1 & \mathsf{B} \\ X_2 & X_3 & X_4 \\ \mathsf{C} & X_5 & \mathsf{D} \end{array} \qquad \begin{array}{l} X_1 = (\mathsf{A}+\mathsf{B})/2 \\ X_2 = (\mathsf{A}+\mathsf{C})/2 \\ X_3 = (\mathsf{A}+\mathsf{B}+\mathsf{C}+\mathsf{D})/4 \end{array}$$

Figura 8. Interpolación de píxeles en MPEG1.

Las referencias a otras imágenes están especificadas a porciones de la imagen cuadradas que se denominan macro-bloques. Un macro-bloque es una región de una imagen de 16x16 píxeles. Cada (macro-)bloque se busca en la imagen que se quiere predecir y las mejores coincidencias se usan para construir un vector de movimiento, el cual indica donde se encontrara esa región de la imagen en el siguiente instante de tiempo. Dependiendo de si la búsqueda en bloques coincidentes en ambas imágenes a sido exitosa o no y al número de éstas, es decir, el número de referencias encontradas, los macro-bloques son clasificados en:

1. **I:** Cuando la compresión de los bloques residuo genera mas bits que el original. Por tanto, no se suele comprimir.
2. **P:** Cuando es mejor comprimir el bloque residuo y hay sólo una referencia a un macro-bloque.
3. **B:** Idem a P, pero hay dos referencias a macro-bloques.
4. **S (Omitidos):** Cuando la energía del bloque residuo es menor que la dada por el umbral establecido. Por tanto, al no ser relevante para el conjunto de la imagen, no se codifica nada.

La estimación del movimiento puede ser llevada a cabo usando la resolución real de la imagen, o identificando una localización intermedia entre varios píxeles como su fracción. En la Figura 8 se ilustra dicho concepto.

Ahora bien, ¿qué criterio usa el compresor para determinar si efectivamente dos macro-bloques presentan una referencia entre ellos, siendo útil como predicción de movimiento? Existen varios como son la diferencia de error o error entrópico. El mas usado se denomina criterio de similitud y su forma de funcionar consiste en medir la diferencia o error entre dos macro-bloques. Ésta medición se lleva a cabo de dos formas: Los macro-bloques que se quieren comparar están representados por a y b.

1. Medida del error al cuadrado
$$\frac{1}{16\times 16}\sum_{i=1}^{16}\sum_{j=1}^{16}(a_{ij}-b_{ij})^2$$
2. Medida del error absoluto
$$\frac{1}{16\times 16}\sum_{i=1}^{16}\sum_{j=1}^{16}|a_{ij}-b_{ij}|$$

Pero definamos primero cómo se llevan a cabo básicamente estas búsquedas. Hay dos parámetros básicos en toda búsqueda, y son: qué buscar y dónde buscarlo. Lo que se busca es un macro-bloque que esté en 2 imágenes consecutivas de la secuencia, llamadas imagen de referencia e imagen predicha. Como ya hemos visto, determinar si esta o no, depende de los "criterios de coincidencia" que usemos, ya que no sólo se buscan macro-bloques idénticos, sino con un parecido razonable. Lo que ocurre es que buscar en toda la imagen un macro-bloque es muy costoso computacionalmente y además innecesario. Las búsquedas mas productivas se llevan a cabo considerando que el elemento que enmarca el macro-bloque tiene una limitación de desplazamiento de un frame de la secuencia a otro, y por tanto, es muchísimo mas productivo buscar un macro-bloque en las inmediaciones de donde estaba en el frame anterior. Esta región de la imagen se denomina "área de búsqueda". A continuación se exponen los distintos tipos de búsqueda:

1. **Búsqueda completa:** Haciendo un uso intensivo de la CPU, se comprueban todas las posibilidades. La ventaja es que se encuentran todas las coincidencias existentes y, por tanto se llega a la mejor compresión posible para MPEG1.
2. **Búsqueda logarítmica:** Es una versión de la 'búsqueda completa', pero de mayor eficiencia. Esto se consigue al reducir los macro-bloques y las áreas de búsqueda, al trabajar sobre un nivel de resolución de la imagen menor al 1:1, tanto como se desee. Una vez encontrada la mejor coincidencia entre macro-bloques, se transforma[1] la imagen al siguiente nivel de resolución, acercándonos a las proporción 1:1. Y de igual modo se adaptan la información recogida (las referencias o vectores de movimiento) hasta el momento a la nueva resolución, esto es, aumentando también las proporciones del propio macro-bloque y reubicando la referencia al mismo, en un área de tan sólo ± 1. Es decir, se lleva a cabo una búsqueda completa, pero en una imagen mas pequeña, esto es así porque la hemos reducido, cuantas veces queramos, a un nivel resolución X veces menor al original [7], [8]. De esta manera obtenemos dos ventajas: La primera y evidente es que al aplicar la búsqueda en una imagen mas pequeña, el proceso se lleva a cabo requiriendo mucho menos tiempo. La segunda es que la granularidad de los vectores de movimiento encontrados se progresiva y de alta calidad.
En primer lugar, el algoritmo es progresivo. Al tener la imagen en una resolución menor, los datos que conserva dicha imagen son los pertenecientes a los elementos mas grandes y notorios, descartándose de esta forma tan sencilla todos los pequeños detalles de la imagen. De esta manera los primeros vectores de movimiento que se encuentran son los mas importantes, ya que ellos predicen el movimiento de los elementos más visibles en la imagen. Así tenemos un sistema que proporciona las referencias más importantes desde el primer instante de ejecución y según le demos tiempo, nos provee no sólo de mas referencias, sino también de una mayor precisión en las referencias encontradas anteriormente.
Y en segundo lugar, proporciona estimaciones de ca-

[1] La transformación consiste en aplicar la transformada Wavelet e inversa de Wavelet para llevar la imagen a niveles resolución menores o mayores, respectivamente.

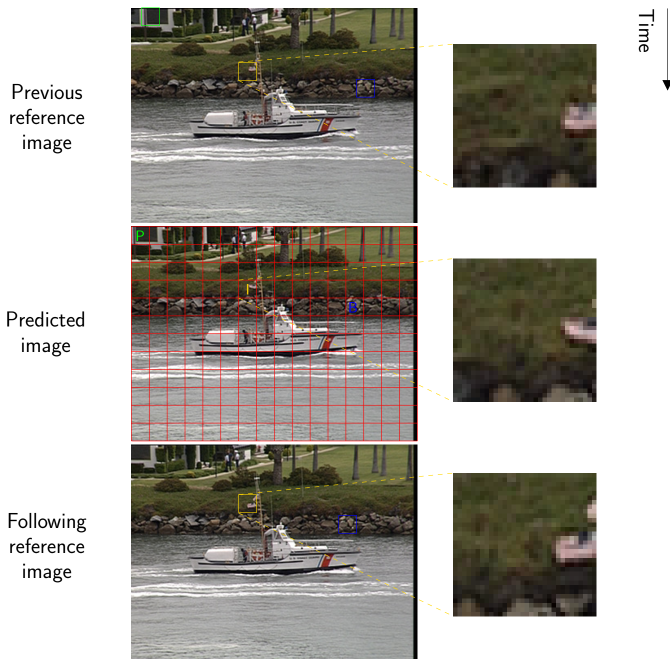

Figura 9. El proceso de predicción en MPEG-1.

Motion VLC code	offset	1	0
0000 0011 001	−16	010	1
0000 0011 011	−15	0010	2
0000 0011 101	−14	0001 0	3
0000 0011 111	−13	0000 110	4
0000 0100 001	−12	0000 1010	5
0000 0100 011	−11	0000 1000	6
0000 0100 11	−10	0000 0110	7
0000 0101 01	−9	0000 0101 10	8
0000 0101 11	−8	0000 0101 00	9
0000 0111	−7	0000 0100 10	10
0000 1001	−6	0000 0100 010	11
0000 1011	−5	0000 0100 000	12
0000 111	−4	0000 0011 110	13
0001 1	−3	0000 0011 100	14
0011	−2	0000 0011 010	15
011	−1	0000 0011 000	16

Figura 10. Códigos de longitud variable usados para codificar los vectores de movimiento en MPEG-1.

lidad. Esto es necesario porque para que la predicción del movimiento sea satisfactoria desde un punto de vista subjetivo del espectador, ésta debe ser precisa, de forma que no se solapen distintos elementos de la imagen. En esta línea esta demostrado que el mejor método de precisar el origen y destino de un macro-bloque en una secuencia de imágenes consiste precisamente en determinar el elemento al que pertenece dicho macro-bloque y después ir precisando su ubicación a nivel de píxeles, y no directamente esto último. Como vemos, de nuevo este proceso viene dado de forma inherente al empezar a trabajar con las imágenes en niveles de resolución inferiores. Aquí nos podemos percatar de la gran utilidad que tiene el trabajar sobre todo el dominio del espacio.

3. **Búsqueda telescópica:** No sólo podemos reducir el área de búsqueda, reduciendo la resolución de cada imagen, sino siendo conscientes de las características inherentes que todo vídeo tiene (probabilísticamente hablando). Esto es, que los campos de movimiento de dos imágenes consecutivas probablemente sean muy similares. Y aprovechando esta similitud, el campo de movimiento del segundo frame se pueda calcular mas rápidamente si partimos del campo de la imagen anterior, para posteriormente ir especificando con mas exactitud el vector de movimiento concreto para dicho frame en búsquedas muy localizadas. Así las técnicas vistas anteriormente pueden ser mas rápidas si la búsqueda se realiza en un área reducida y suponiendo que los vectores de movimiento no van a cambiar bruscamente de ubicación entre frames.

III-C3. El predictor ($\mathcal{P}$): El predictor usado en MPEG-1 está basado en la copia (P) o promediado (B) de macro-bloques extraidos de la(s) imágen(es) de referencia a partir de los vectores de movimiento determinados en la fase ME (véase la Sección III-C2). En la Figura 9 se muestra un ejemplo de este proceso.

III-C4. La transformada discreta del coseno ($\mathcal{T}$): La transformación es idéntica a la usada en JPEG (véase la Sección III-A4).

III-C5. El cuantificador ($\mathcal{Q}^q$): Se trata de un proceso de cuantificación escalar idéntico al usado en JPEG (véase la Sección III-A5).

III-C6. Codificación de las texturas residuo (VLC): De nuevo, el MPEG-1 hereda esta funcionalidad del JPEG (véase la Sección III-A6).

III-C7. Codificación de los vectores de movimiento (VLC): En MPEG-1, los vectores de movimiento se codifican usando el código de Huffman mostrado en la Figura 10. Dichos vectores son representados como una tupla *(x,y)*, donde *x* e *y* siguen una distribución de probabilidad de Laplace con un rango de $0 \leq x, y < 16$. Se trata de un código de longitud variable para el desplazamiento del movimiento en otras palabras, el código de compresión para un determinado vector de movimiento.

III-C8. En controlador del bit-rate (BC): MPEG-1 posee básicamente dos modos de funcionamiento:

1. **Modo CBR (Constant Bit-Rate):** En este modo, el módulo BC decide qué niveles de cuantificación deben ser utilizados para que la tasa de bits de salida del compresor permanezca constante a lo largo del tiempo.
2. **Modo VBR (Variable Bit-Rate):** Por contrapartida, en este modo el sistema BC no funciona. El nivel de cuantificación se mantiene constante y por tanto, la cantidad de bits a la salida es proporcional al contenido visual de la secuencia de vídeo.

III-C9. El multiplexor de texturas/movimiento (Mux): Una vez que los code-streams de las texturas y el movimiento han sido generados, llega el momento se entrelazarlos para que durante la reproducción, tras un cierto retardo generalmente inapreciable, ambos streams puedan ser descomprimidos

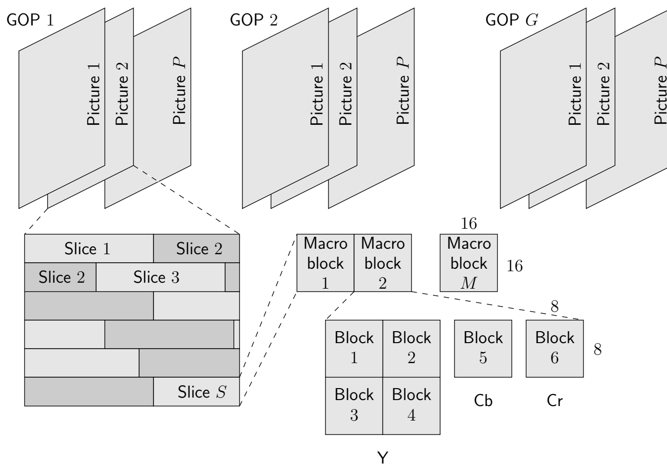

Figura 11. Distribución de los datos en MPEG1.

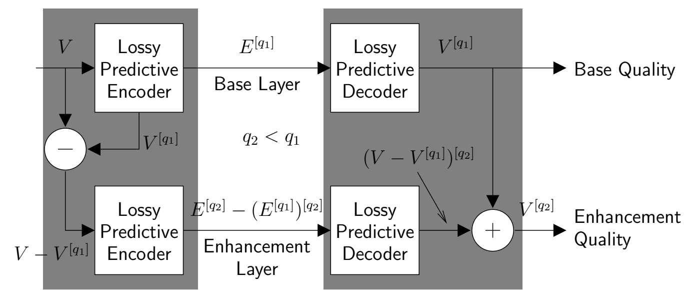

Figura 12. Escalabilidad en calidad MPEG-2.

simultáneamente. Esta tarea de entrelazamiento la realiza el multiplexor.

III-C10. Distribución de los datos en MPEG-1: Cada una de las capas (slice en la imagen) es codificada independientemente quedando comprimida y es etiquetada con un código especial en cadena. Esto aumenta la capacidad de recuperación de errores del código. El tamaño mínimo de una capa es un macro-bloque y el máximo es un frame completo.

Como se observa en la Figura 11 cada macro-bloque contiene bloques de datos de luminancia y crominancia (cuatro bloques de luminancia, Y1, Y2, Y3, Y4, y dos de crominancia, U y V).

III-D. MPEG-2

El nombre de estándar es ”Generic Coding of Moving Pictures and Associated Audio” o *ISO/IEC 13818-1*. Creado en 1994, se masificó su uso en DVDs (Digital Versatile Disc), televisión digital y DVB-T (Digital Video Broadcasting Terrestrial). Constituye una evolución de MPEG-1, el cual, se puede considerar como un subconjunto del MPEG-2 siendo compatible con éste. Sus ventajas son: el soporte para audio multicanal, vídeo entrelazado (el formato usado por las televisiones) y progresivo (el usado en los motitores de ordenador, cámaras fotográficas y de vídeo, etc.).

La sintaxis del MPEG-2 tiene dos categorías:

1. Una sintaxis *no escalable*, la cual incluye a la sintaxis del MPEG-1, con extensiones adicionales para soportar vídeo entrelazado.
2. Una sintaxis *escalable*, que permite una codificación por capas de la señal de vídeo, mediante la cual, el descodificador puede descodificar:
   - Sólo la capa básica para obtener una señal con calidad mínima, o
   - utilizar capas adicionales para incrementar la calidad de la señal. Lo que lo hace ideal para entornos de visualización remota, ya que se puede adaptar el peso de la señal enviada según este parámetro.

En comparación MPEG-2 es mejor que su antecesor en que:

1. **Posee un mayor rango de bit-rates.** Concretamente entre 4 Mbps y 30 Mbps. MPEG-2 no esta optimizado para bajas tasas de bits (menores que 1 Mbit/s), pero supera en desempeño a MPEG-1 a 3 Mbit/s y superiores.
2. **Soporta mayor resolución.** Es el estándar de televisión en alta definición (HDTV).
3. **Posee una resistencia mejorada a los errores.** MPEG-2 introduce y define *Flujos de Transporte*, los cuales son diseñados para transportar vídeo y audio digital a través de medios impredecibles e inestables (como es la TDT o la TV por satélite). Además como se expondrá en el siguiente punto la escalabilidad SNR también contribuye a esta mejora.[2]
4. **Es escalable a nivel SNR y espacial.**
   a) En la Figura 12 se ilustra el mecanismo que utiliza MPEG-2 para llevar a cabo la **escalabilidad SNR**, donde se dibuja a grandes rasgos el proceso de codificación y decodificación con pérdida, propio de MPEG-2. La notación usada es la misma que en los anteriores gráficos. La ”cantidad de calidad” de la imagen viene determinada por el bit-rate que se le asigna y por el número de capas que la compongan. Se representa por un superíndice en la Figura 12, en el que a mayor índice, menor bit-rate, de tal forma que $V_i^{[0]} = V_i$ (siendo V_i la i-ésima imagen de la secuencia de vídeo).
   Continuando con la Figura 12, se observa que el flujo de bits es dividido en dos capas. La idea esta basada en el hecho de que el sistema visual humano es más insensible a las componentes de alta frecuencia de las señales de vídeo. Las semejanzas entre capas consisten en que tienen la misma resolución espacial para diferentes calidades de vídeo. Así como que cada capa contiene todos los coeficientes DCT, pero con la particularidad de que cada capa puede tener un valor diferente

[2]El hecho de que el esquema de codificación de vídeo MPEG-2 utilice técnicas de eliminación de la redundancia o compresión hace que cualquier aplicación de vídeo MPEG-2 sea muy vulnerable ante la pérdida de información y errores en la decodificación o en la transmisión. Su impacto negativo sobre la calidad de vídeo se ve amplificado debido a los fenómenos de propagación espacial y temporal de errores presentes en el estándar MPEG-2. Puesto que no hay un esquema adicional que controle la propagación de las pérdidas o los errores. Por tanto, la escalabilidad permite introducir códigos de detección y corrección de errors de forma desigual, haciendo mucho más resistente la capa básica que la de mejora y de esta forma, el bit-rate efectivo no se ve incrementado de forma significativa.

del factor de escala. A continuación se describen los tipos de capas: La llamada "capa base" o **base layer** transmite con un nivel de prioridad alta: los coeficientes de baja frecuencia, los modos de codificación y los vectores de movimiento. La "capa de mejora" o **enhancement layer**, transmite en un nivel de prioridad mas bajo donde la probabilidad de pérdidas es mayor: Los coeficientes de alta frecuencia, entre otra información menos importante, proporcionando un mayor refinamiento de los coeficientes de la DCT de la capa básica, mejorando la calidad proporcionada por ésta. Así la escalabilidad SNR proporciona también una mejor resistencia a errores de transmisión. Por ejemplo, las capas de mejora pueden transmitirse sobre canales con peores prestaciones, mientras que la capa básica (sobre mejores medios) provee siempre de un mínimo de la señal.

Este modo de escalabilidad es idóneo en aplicaciones de transmisión de vídeo, cuando se tienen redes de transmisión que soportan dos niveles de prioridad como las redes con tecnología ATM [9].

b) La **escalabilidad espacial** fue diseñada para ser utilizada como herramienta que permitiese la interoperatividad entre diferentes estándares y entre diferentes sistemas con diferentes resoluciones como dispositivos que trabajan a bajas resoluciones (un móvil, por ejemplo), resoluciones medias (una TV, por ejemplo), y altas resoluciones (la HDTV, por ejemplo), con un mismo flujo de bits (pudiendo tener diferentes: tamaños de cuadro, caudal de cuadros y formatos de muestro).

El *flujo de bits* se divide en capas de diferente resolución espacial. La capa básica es codificada por si misma y proporciona la resolución espacial básica, mientras que la capa de mejora utiliza la capa básica, para proporcionar la resolución espacial completa de la señal de vídeo.

III-E. MPEG-4

La cuarta y última versión de MPEG esta disponible desde 1999 y es muy usado en contenido DivX. Su nombre técnico es *Coding of audio-visual objetcs* [10]. Como su anterior versión también esta basado en una codificación híbrida, pero ha sido optimizado para trabajar con niveles de bit-rate muy bajos [11], hasta 5 Kbps.

Los medios audio-visuales son descompuestos en objetos (objetos visuales y de audio), donde cada objeto puede ser sintético o natural, y puede ser codificado usando diferentes técnicas como sprites, codificación híbrida, mallas 2D ó 3D con texturas, etc.

Las ventajas de MPEG-4 frente a la versión anterior (MPEG-2) se resumen básicamente en refinar la precisión a 1/4 de sub-pixel. Aumentar el número de vetores de movimiento o macro-bloques así como de referencias a imágenes hasta 4. Y mejorar la corrección de errores en el código.

III-E1. Niveles y perfiles: La mayoría de las características que conforman el estándar MPEG-4 no tienen que estar disponibles en todas las implementaciones, al punto que no existen actualmente implementaciones completas del estándar MPEG-4 [12]. Para manejar esta variedad, el estándar incluye el concepto de *perfil* (profile) y *nivel*, lo que permite definir conjuntos específicos de capacidades que pueden ser implementados para cumplir con objetivos particulares. Los principales parámetros de cada perfil son:

1. Tamaño típico de visionado.
2. Máximo número de objetos.
3. Máximo bit-rate (Kbps).
4. Máximo número de capas de mejora.

III-E2. Mayor precisión en la escalabilidad de la granularidad: Debido a la amplia variación de ancho de banda disponible en intervalos cortos para sesiones inalámbricas, hay una necesidad de métodos de codificación de vídeo escalable que permitan al flujo de datos adaptarse flexiblemente a las condiciones de red cambiante en tiempo real. Una técnica es la *Escalabilidad Granular Fina (FGS)* de MPEG-4, la cual puede adaptarse en tiempo real a variaciones de ancho de banda mientras usa el mismo flujo de datos pre-codificado. Por tanto, las ventajas clave del entramado de MPEG-4 FGS (elasticidad y flexibilidad) vienen a expensas de una calidad de vídeo sensiblemente inferior.

III-F. H.264/AVC

El estándard H.264/AVC [13], es también llamado *Estandar ISO/IEC 14496-10*. Constituye una evolución más de MPEG-4, siendo de hecho su parte número 10. Es una norma que define un códec de vídeo de alta compresión, desarrollada conjuntamente por el ITU-T Video Coding Experts Group (VCEG) y el ISO/IEC Moving Picture Experts Group (MPEG). La intención del proyecto H.264/AVC fue la de crear un estándar capaz de proporcionar una buena calidad de imagen con tasas binarias notablemente inferiores a los estándares previos (MPEG-2, H.263 o MPEG-4 parte 2), además de no incrementar la complejidad de su diseño.

III-F1. Diferencias con los codecs anteriores: Para empezar a programar el código del nuevo estándar se adoptaron las siguientes premisas:

1. La estructura DCT + Compensación de Movimiento de las versiones anteriores era superior a otros estándares y por esto no había ninguna necesidad de hacer cambios fundamentales en la estructura.
2. Algunas formas de codificación de vídeo que habían sido excluidas en el pasado debido a su complejidad y su alto coste de implementación se volvieron a examinar para su inclusión puesto que la tecnología VLSI *Very Large Scale Integration* había sufrido un adelanto considerable y una bajada de costes de implementación.
3. Para permitir una libertad máxima en la codificación y evitar restricciones que comprometan la eficiencia, no se contempla mantener la compatibilidad con normas anteriores.
4. El tamaño de los macro-bloques puede ser de 16, 8 y 4, en cualquier combinación.

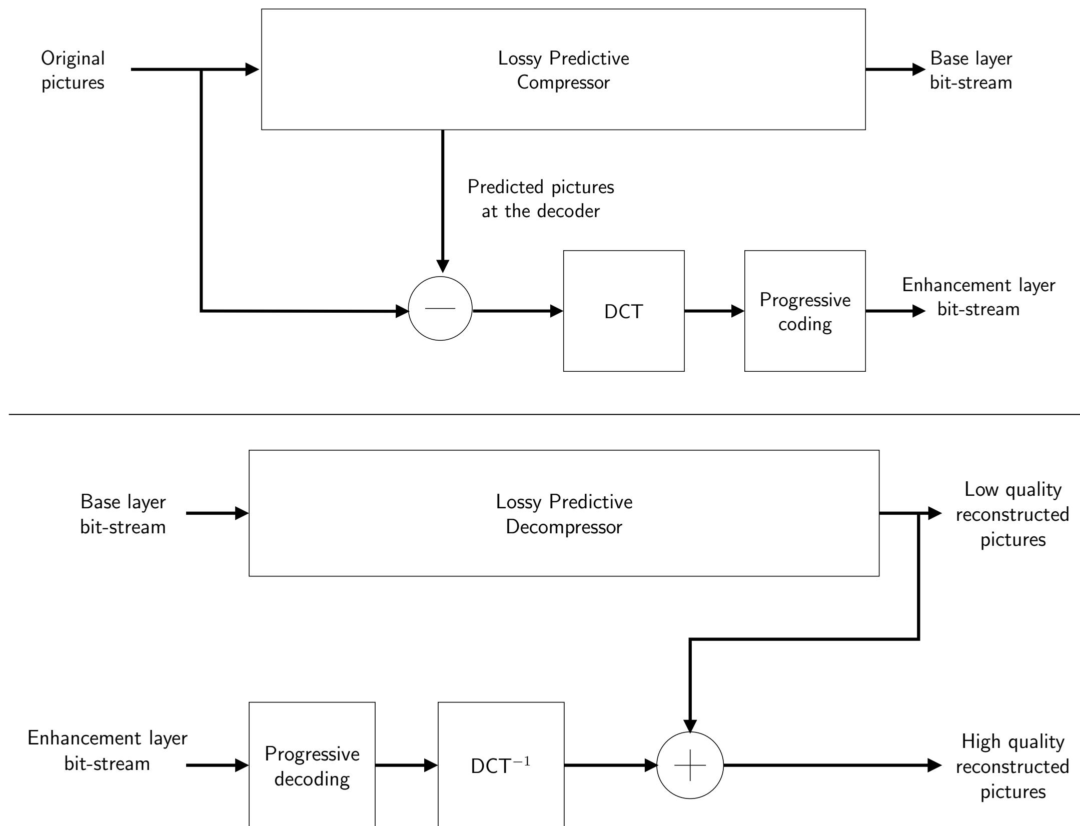

Figura 13. El codec MPEG-4 escalable en calidad. Arriba el compresor y debajo el descompresor.

5. Codificación mejorada de la entropía:
   a) CAVLC (Coded according to the context adaptive variable length) [14]. El objetivo de esta codificación es procesar la información que se quiere transmitir o almacenar en un dispositivo de forma que ocupe el mínimo espacio posible. De esta manera, con el uso de la CAVLC sera posible transmitir una imagen en menos tiempo o hacer que ocupe menos espacio en el dispositivo de almacenamiento. Una característica importante de esta codificación es que no tiene pérdidas y que por lo tanto se podrá recuperar la información original al aplicar el proceso inverso. Se emplea para codificar y comprimir la información que resulta de la aplicación de la transformación y cuantificación de un bloque de luminancia de tamaño 4x4 píxeles.
   b) CABAC, en inglés Context Adaptative Binary Arithmetic Coder. Presenta una mejora en la dimensión del bitstream respecto a CAVLC, dado que consigue bitstreams un 10 por ciento mas pequeños. La arquitectura de CABAC demuestra que es bastante buena ya que puede comprimir eficientemente la señal original. Sin embargo, el rendimiento se podría mejorar considerando que cada coeficiente es estadísticamente dependiente de sus vecinos.
6. Soporte de hasta 16 referencias a imágenes en muestras de macro-bloques tipo B.
7. Post-compresión que mejora los ratios conseguidos inicialmente.
8. Compresión sin pérdida, si no se lleva a cabo la cuantificación.

III-F2. Perfiles: El uso inicial del MPEG-4/AVC estuvo enfocado hacia el vídeo de baja calidad (ej. videoconferencia) basado en 8 bits/muestra y con un muestreo ortogonal de 4:2:0. Esto no daba salida al uso de este códec en ambientes profesionales que exigen resoluciones más elevadas, necesitan más de 8 bits/muestra y un muestreo de 4:4:4 ó 4:2:2, funciones para la mezcla de escenas, tasas binarias mas elevadas, poder representar algunas partes de vídeo sin pérdidas y utilizar el sistema de color por componentes RGB. Por este motivo surgió la necesidad de una extensiones, llamadas ”extensiones de gama de fidelidad” *(FRExt)* que incluían soporte para:

1. Un tamaño de transformada adaptativo,
2. una cuantificación con matrices escaladas y
3. una representación eficiente sin pérdidas de regiones específicas. El conjunto de extensiones denominadas de ”perfil alto” soportan 4:2:0 con hasta 8 bits/muestra, 4:2:0 ó 4:2:2 con hasta 10 bits/muestra y, por último, 4:4:4 con 12 bits/muestra y la codificación de regiones sin pérdidas.

III-G. H.264/SVC

También referenciado como *Estandar ISO/IEC 14496-10*, continua en desarrollo. Consiste en la versión escalable de H.264/AVC, siendo compatible con éste. En otras palabras, la

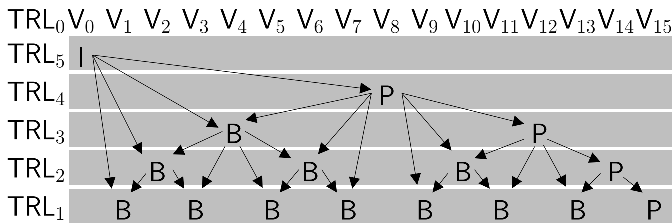

Figura 14. Escalabilidad temporal usando MCTF en H.264/SVC.

capa base dada por el codec H.264/SVC puede se descomprimida por H.264/AVC.

III-G1. Diferencias con los codecs anteriores: Aunque H.264/SVC es compatible hacia atrás con H.264/AVC, define un nuevo esquema de descorrelación temporal llamado MCTF [15], en inglés *Motion Compensated Temporal Filtering*. Es más eficiente que su homólogo anterior (recordemos el esquema IPPP...), además de permitir la escalabilidad en el tiempo.

H.264/SVC no sólo aporta esta nueva forma de escalabilidad, sino que también las otras dos, la espacial y en calidad. Así el diseño del SVC apoya la creación de flujos de bits de calidad *escalable* que se pueden convertir en flujos de bits que se ajustan a uno de los perfiles H.264/AVC *no escalable* mediante una reescritura del proceso a baja complejidad.

III-G1a. Escalabilidad temporal: La escalabilidad temporal provee de un acceso rápido a cualquier frame del vídeo. Concretamente en H.264/SVC se implementa la escalabilidad temporal por medio de MCTF. En la Figura 14 se representan las dependencias entre distintos tipos de imágenes que se crean para permitir la escalabilidad temporal en este codec.

En el *eje X* figuran los tipos de imágenes (I, P y B). Recordemos que V_0 se corresponde con una imagen *intra (I)* la cual tiene la misma codificación que la imagen original sin comprimir y sin vectores de movimiento. En el *eje Y*, TRL, representa los *niveles de resolución temporales*, o en otras palabras, los frames por segundo. Es interesante observar que, según la capacidad del medio o por petición expresa, el codec es capaz de mostrar el número de imágenes por segundo adecuado, de tal forma que, si se desea sólo una imagen, se muestra la intra, si se desean 2, se muestra la I y la P (la imagen P depende de la I), si se desean 3, se muestran la I, dos P y una B (con sus respectivas dependencias). Según el esquema del ejemplo presentado, la escalabilidad temporal podría proveer desde 1 hasta 8 imágenes por segundo.

III-G2. Escalabilidad espacial: Como se observa en la Figura 23 en comparación con la anterior (Figura 14). Las dependencias entre imágenes para las escalabilidades temporal y espacial, son evidentemente distintas. Por poner un ejemplo: la imagen intra (I) sirve de base para dos B y una P. Si bien se exponen las dependencias entre imágenes en un mismo rango en el dominio del espacio menor (abajo), también vemos que cada imagen se corresponde de forma directa con su versión en un rango del espacio mayor (arriba).

III-G3. Escalabilidad en calidad: Las capas son identificadas por su dependencia con el identificador D (véase le Figura 16). Cada capa contiene una o mas capas de calidad, que representan la fuente de vídeo para un instante de tiempo con una resolución espacial y una fidelidad (calidad)

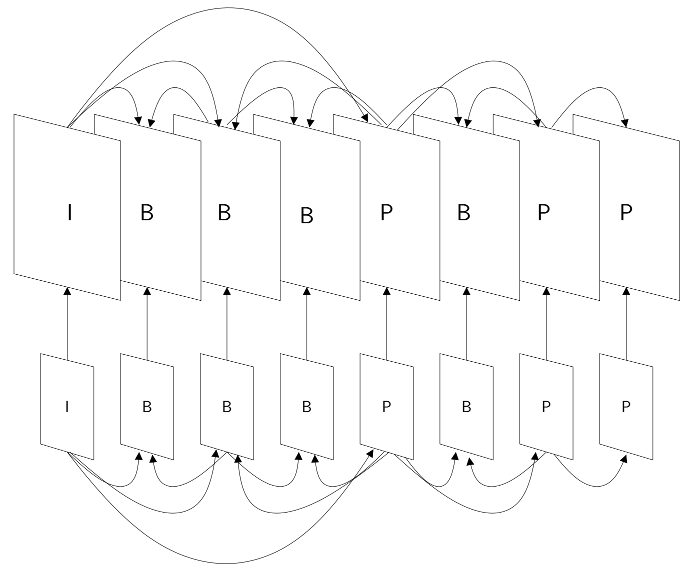

Figura 15. Escalabilidad espacial en H.264/SVC.

Capa (D)	Resolución	Calidad	Ratio de la imagen	(S,T,Q)
0	176x144	0	3.75	(0,0,0)
1	176x144	0	7.5	(0,1,0)
2	176x144	0	15	(0,2,0)
3	176x144	0	30	(0,3,0)
4	176x144	1	3.75	(0,0,1)
5	176x144	1	7.5	(0,1,1)
6	176x144	1	15	(0,2,1)
7	176x144	1	30	(0,3,1)
8	352x288	0	3.75	(1,0,0)
9	352x288	0	7.5	(1,1,0)
10	352x288	0	15	(1,2,0)
11	352x288	0	30	(1,3,0)
12	352x288	1	3.75	(1,0,1)
13	352x288	1	7.5	(1,1,1)
14	352x288	1	15	(1,2,1)
15	352x288	1	30	(1,3,1)

Figura 16. Ejemplo de generación de capas en H.264/SVC.

específica. Éstas se representan en la última columna de la tabla con las siglas T, S y Q respectivamente.

El conjunto de capas de calidad dependientes entre sí corresponden a la misma resolución espacial y se identifican por un identificador de calidad Q. Las capas de calidad son creadas progresivamente [16] según un nivel de cuantificación decreciente. De este modo, para las capas de refinamiento de la calidad con valor de $Q > 0$ indica que siempre la capa anterior de calidad, con un identificador de calidad Q - 1, se emplea para la capa intra de predicción. Y para el valor de $Q = 0$, se puede seleccionar como capa de referencia para la predicción cualquier capa anterior. De ella se extraen además las siguientes afirmaciones:

1. Si la capa de mayor calidad, la número 15, no puede ser descomprimida por cualquier motivo, por ejemplo, falta de ancho de banda del medio, entonces las imágenes impares deberán ser descomprimidas con la mínima calidad.
2. Si la capa número 14 se pierde, entonces sólo las imágenes cuyo índice sea divisible entre 4 se obtendrán con la máxima calidad.

Figura 17. Las tres formas de escalabilidad en JPEG 2000.

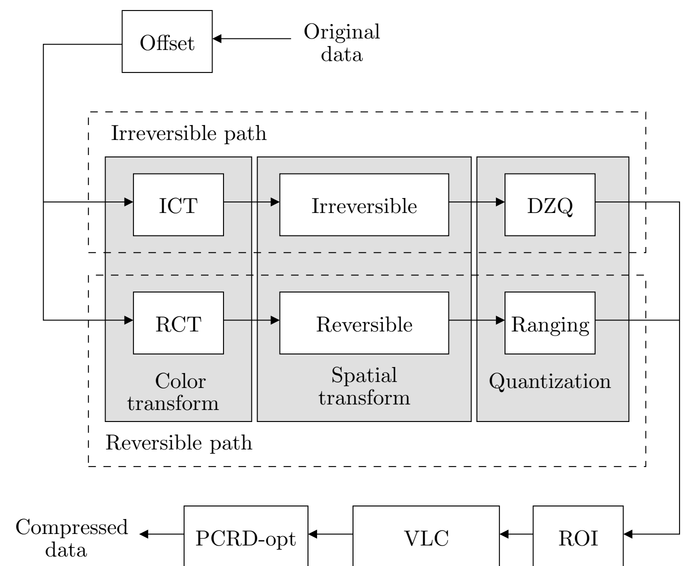

Figura 18. Arquitectura del compresor JPEG 2000.

Figura 19. La imagen Lena comprimida a 0.1 bits/pixel usando JPEG (izquierda) y JPEG 2000 (derecha). La calidad visual se ha expresado además usando la métrica PSNR[dB].

3. En caso de que las capas 12 a 15 se pierdan, entonces todas las imágenes se obtendrán con calidad mínima.
4. Si la capa 11 se pierde, entonces las imágenes impares se obtendrán con calidad 1 pero con una resolución de 176x144.
5. Si sólo se descomprime la capa base, la 0, entonces sólo una imagen de cada 8 se visualizará, con calidad 0 y un tamaño de 176x144.

III-G4. Número de capas: Bajo un escenario real de transmisión, como puede ser Internet, la cantidad de datos que se pueden transmitir dependen de múltiples factores impredecibles y esto debe prevenirse en la compresión del vídeo. Para maneja esta situación, el número de capas de calidad debe ser lo mas alto posible para así poder adaptarse a la red y aprovechar al máximo el ancho de banda. Ya que no se puede transmitir media capa, en cuantas mas capas se divida la calidad, con mayor precisión se ajustara a los requisitos de la red. Sin embargo, se debe llegar a un compromiso para dicho número, puesto que a mayor nº de capas menor ratio de compresión.

III-H. JPEG 2000 (J2K)

JPEG 2000 es el último estándar de compresión de imágenes desarrollado por la ISO [17]. Fue creado con la idea de reemplazar al omnipresente estándar JPEG. Como puede apreciarse en la Figura 18, el compresor JPEG 2000 (sólo J2K en adelante) está formado por una serie de etapas secuenciales que son:

1. La transformada de color, que eliminar la correlación ínter-componente.
2. La transformada espacial (una trasformada wavelet), que elimina la correlación intra-componente.
3. La fase de cuantificación que elimina la información visualmente menos relevante.
4. La etapa de definición de la(s) ROI(s) (Regions Of Interest).
5. La codificación de longitud variable o VLC (Variable Length Coding) que hace efectiva la compresión de los datos.
6. La etapa PCRD-opt [18] (Post Compression Rate-Distortion Optimization), que se utiliza para organizar los datos en el code-stream con el objeto de conseguir

las diferentes formas de escalabilidad.

J2K presenta sobre JPEG las siguientes ventajas:

1. Es más eficiente en términos R/D (véase la Figura 19).
2. Pueden realizarse tanto compresiones *lossy* (irreversibles) como *lossless* (totalmente reversibles), dependiendo de los requerimientos del usuario. La versión lossy arroja tasas de compresión ligeramente superiores a las de la versión lossless.
3. Es escalable en calidad, en resolución y en región y ventana[3] de interés. En la Figura 17 se muestra un ejemplo de cada tipo de escalabilidad. Para conseguir estas tres formas de escalabilidad a partir de un único code-stream, el estándar define (entre otras) tres ordenaciones diferentes de los paquetes de datos JPEG 2000 [19]. La progresión LRCP (Layer-Resolution-Component-Precinct) permite una descodificación progresiva en calidad al ordenar los datos referentes a los coeficientes wavelet por planos de bits. La progresión RLCP produce una descodificación progresiva por niveles de resolución al ordenar los datos referentes a los coeficientes wavelet por subbandas de frecuencia espacial incremental y finalmente, la progresión por WOI (Window Of Interest) se genera al poder seleccionar el subconjunto correspondiente de paquetes que hacen referencia a dicha ventana.

III-I. Motion JPEG 2000 (MJ2K)

Motion JPEG 2000 [20] se define en la norma *ISO / IEC 15444-3* y en la *UIT-T T.802*. En ella se especifica el uso del codec JPEG 2000 para las secuencias de imágenes programadas (secuencias de movimiento), combinado con audio, y compuesto en una presentación global. Las extensiones de archivos para archivos de vídeo Motion JPEG 2000 son `.mj2` y `.mjp2` de acuerdo con la norma *RFC 3745*. Es el principal soporte de cine digital estándar en la actualidad con el apoyo de las *Digital Cinema Initiatives*, un consorcio de la mayoría de los principales estudios y proveedores, para el almacenamiento, distribución y exhibición de películas cinematográficas. También se está considerando como un formato de archivo digital por la Biblioteca del Congreso.

A diferencia de los codecs de vídeo comunes, como MPEG-4, MJ2K no emplea la compresión de estructura temporal. En su lugar, cada fotograma es una entidad independiente codificada por una variante con o sin pérdidas de JPEG 2000.

III-J. Motion Compensated Temporal Filtering (MCTF)

El estudio de nuevas formas de mejorar la codificación de vídeo y desarrollo de las ya existentes es muy prolífico. Cada uno de esos trabajos consiguen buenos resultados para un subconjunto de características que debería tener la codificación ideal para un vídeo. Y aunque en algunos trabajos se encuentren formas de mejorar alguna característica concreta, como por ejemplo, una rápida accesibilidad aleatoria al vídeo [21], [22], o simplemente hacer el vídeo mas portable en una transmisión por un canal estrecho [23], [24]. Sin embargo, es evidente que la compensación temporal de las imágenes de un vídeo reúne el mayor número de ventajas, entre ellas, reduce notablemente los Kbps necesarios para codificarlo. Por ello es un planteamiento muy usado. Sin embargo, aun no existe una metodología que estandarice todo el proceso. De este modo existen numerosos estudios que aplican algún tipo de predicción del movimiento, con técnicas del tipo MCTF, como LIMAT, en inglés *Lifting-based Invertible Motion Adaptive Transform* y que presentan resultados interesantes.

En investigaciones sobre LIMAT [25] se ha trabajado con mallas deformables que pueden mejorar la compensación de movimiento debido a su adaptabilidad a las condiciones de su contexto (particularidades de la secuencia de imágenes), gracias a las expansiones y contracciones en el marco de seguimiento de los movimientos en la imagen, en comparación con mantener un campo de movimiento continuo. Y se concluye que el entorno de trabajo de LIMAT, con el uso de la DWT tiene un rendimiento superior, al menos de 5/3 sobre sólo 1/3 utilizando la transformada de Haar. En esta misma línea de estructuras más flexibles, se estudió la aplicación de DWT para la escalabilidad espacial [7], [26], [27] y como aprovecharlo con el fin de minimizar cálculos para la predicción. Así como también, llevar a cabo un aumento progresivo de la precisión de las referencias para la predicción del movimiento [8].

En el Instituto de Heinrich Hertz (Alemania) se desarrolló la norma H.264, ya comentada, y con ella, el sistema de estimación del movimiento de MCTF. Existen implementaciones de MCTF [28] que usan múltiples frames para el seguimiento del movimiento. De modo que el margen en que se pueden detectar macro-bloques similares se mayor y, por tanto, la predicción es más amplia.

Pero este planteamiento no carece de por menores y algunos trabajos se enfocan sobre los problemas que conlleva aplicar la correlación temporal de imágenes, buscando un compromiso entre eficiencia y calidad de visualización. Uno de los problemas más llamativos es una distorsión visualmente parecida al llamado *efecto fantasma* (ver Figura 20), producido precisamente por el simple hecho de relacionar dos o más imágenes entre sí y dar como resultado fotogramas en los que se muestra información de varios instantes de tiempo a la vez. Esta línea se toma en trabajos donde se consigue reducir este ruido mediante técnicas aplicadas al paso baja de subbandas temporales [29], mejorando además las escalabilidades temporal y espacial.

Otro planteamiento consiste en el concepto de imágenes clave [16] para controlar eficazmente la deriva de la calidad en la codificación basada en paquetes escalables con las estructuras jerárquicas de predicción. O incluso ampliando el procesado a las partes de la imagen que no se encuentren en movimiento, usando un método de adaptación de contenido para compensación de movimiento en tres dimensiones [30].

Como este tipo de trabajos, muchos otros consisten en aumentar la calidad de imagen mediante procesos adicionales al compensado temporal. Desde otro punto de vista, existen los trabajos que mediante una *Highly scalable video compression* [31] consiguen optimizar cuantitativamente el número

[3]Cuando el área de la imagen a descomprimir con mayor calidad se especifica durante la compresión se habla de región de interés o ROI (Region Of Interest). Una ROI puede tener cualquier forma posible. Sin embargo, cuando el área se define durante la descompresión, nos referimos a una ventana o WOI (Window Of Interest), que sólo puede tener forma rectangular.

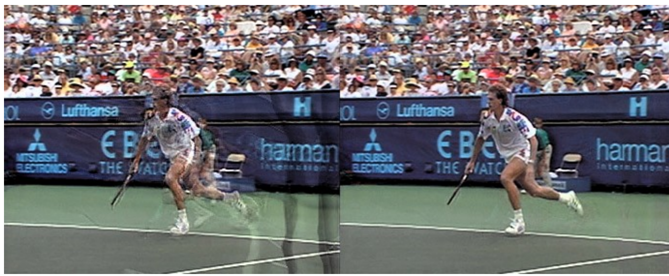

Figura 20. Datos cruzados de varias imágenes.

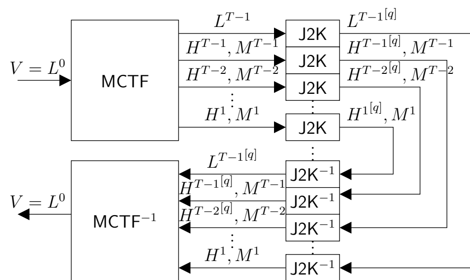

Figura 21. Estructura básica de MCJ2K.

de cálculos de los procesos basados en la Transformada Wavelet. Aumentando significativamente el rendimiento con un detrimento relativamente pequeño en calidad.

III-K. *Motion Compensated JPEG 2000 (MCJ2K)*

III-K1. El papel de MCTF en MCJ2K: Como se ha visto MCTF elimina eficientemente la redundancia temporal de una secuencia de imágenes [32] y provee de una escalabilidad temporal y controles para minimizar el impacto de errores y pérdidas en los datos.

De igual modo, como todos sabemos, J2K es un buen compresor de imágenes y puede crear codificaciones escalables en espacio y calidad. Es posible que ambos codecs pueden utilizarse juntos para diseñar un compresor escalable en espacio, tiempo y calidad. El esquema presentado en la Figura 21 es simple y constituye el esqueleto de MCJ2K.

III-K2. MCTF en MCJ2K: En MCJ2K los GOPs son siempre abiertos y simétricos en cada resolución temporal. Como puede verse en la Figura 22 se presentan 2 GOPs, el primero esta compuesto únicamente por V_0 y el segundo lo constituyen de V_1 a V_{16}.

En el caso en que no se encuentre suficiente correlación temporal, las imágenes B son reemplazadas por imágenes I (y las referencias de los vectores de movimiento se eliminan). Tal

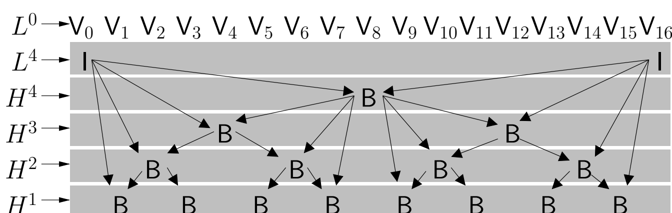

Figura 22. MCTF con suficiente correlación temporal.

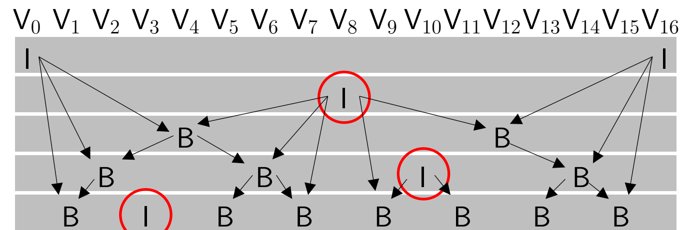

Figura 23. MCTF sin suficiente correlación temporal.

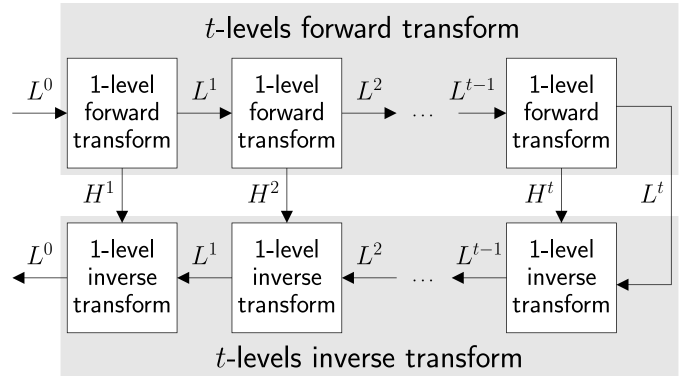

Figura 24. Transformación en el nivel temporal.

y como se observa en la Figura 23. Dicho con otras palabras, una imagen B puede finalmente sustituirse por una INTRA si la entropía de B es demasiado grande (siendo igual o mayor). Es decir, la imagen que se desea predecir mediante vectores de movimiento, es tan distinta a la anterior, que es menos costoso codificar la imagen directamente como INTRA, en lugar de enumerar sus diferencias.

La implementación de MCTF debe sufrir ciertas modificaciones para adaptarse al esquema inicial en la Figura 21. Aquí MCTF sigue una típica implementación recursiva ”filter bank”, con transformaciones tanto hacia adelante como hacia atrás, representada en la Figura 24.

Podemos ver en detalle en la Figura 25 uno de los compenentes de transformación de nivel temporal directo e inverso, propio de las cajas de la Figura 24 desde un punto de vista a más alto nivel.

En la Figura 25, el superíndice t representa el nivel de transformación Wavelet, y el subíndice i el número de fotograma al que pertenece la muestra. En esta línea observamos que se da un trato distinto a los fotogramas pares e impares, donde en los primeros se lleva a cabo la predicción y en los segundos se compara la diferencia entre lo predicho y lo que realmente ocurre.

El mecanismo de una transformada Wavelet consiste en dividir recursivamente los promedios y las diferencias, llevar a cabo una predicción del movimiento (mediante el establecimiento de referencias o vectores de movimiento) y finalmente una actualización o refinamiento de dichas referencias. La transformada inversa consiste en llevar a cabo los pasos en orden inverso, para finalmente unir las medias y las diferencias. Por ejemplo, sean dos señales A y B, promedio S y diferencia D. Se puede recuperar A y B a partir de S y D,

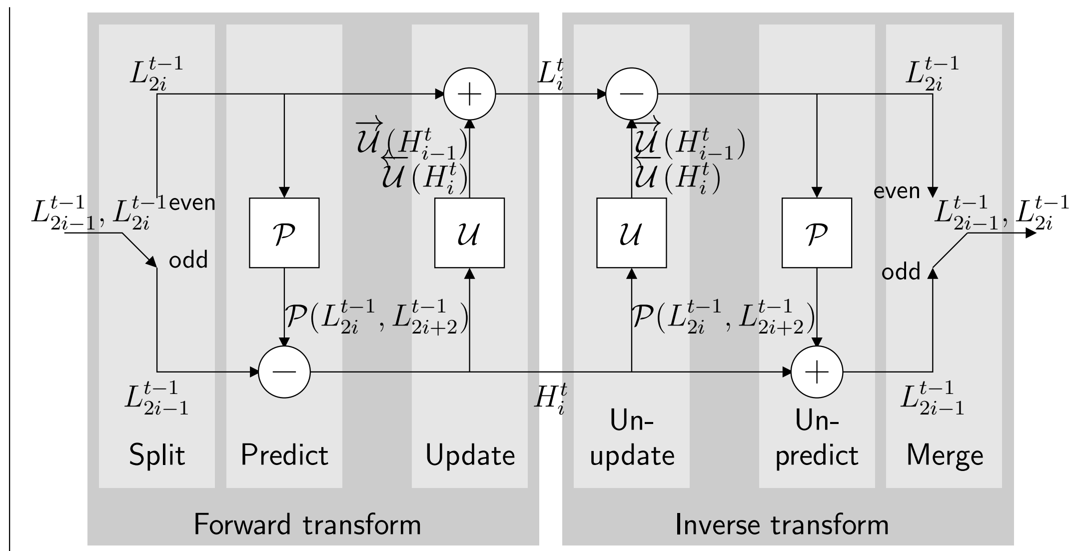

Figura 25. Nivel de transformación directo (izquierda) e inverso (derecha).

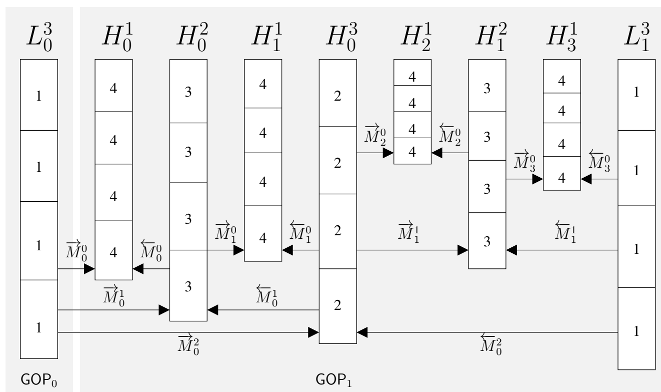

Figura 26. Escalabilidad temporal en MCJ2K.

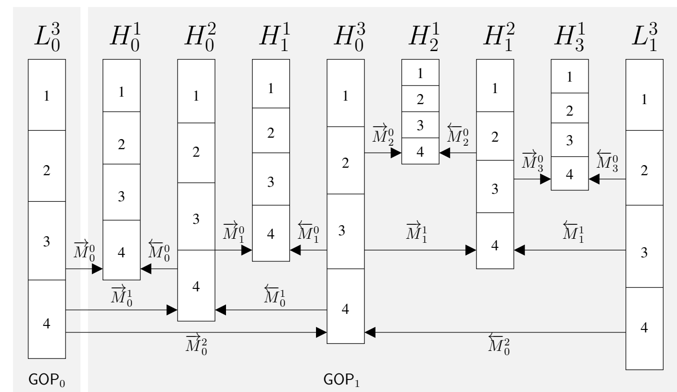

Figura 27. Escalabilidad espacial y en calidad de MCJ2K.

como ecuación. También se puede invirtiendo una matriz 2x2, de ahí la llamada transformada Wavelet de Haar [33]. A mayor correlación entre las dos señales mas pequeña sera D (la resta entre ambas, su diferencia) y se podra representar con menos bits.

III-K3. Escalabilidad temporal, espacial y en calidad: Lo que hace posible estas tres escalabilidades es la división de los datos en capas y la organización de éstas. Existen 5 tipos de ordenaciones (dos muy utilizados) para almacenar los paquetes que componen un fotograma o imagen de un vídeo. Dado que los paquetes en sí no modifican, su ordenación se puede cambiar de un tipo a otro. Claro esta, siempre y cuando dispongamos de dichos paquetes.

La **escalabilidad temporal** representa el número de imágenes por segundo que se transmiten, es decir, la cantidad de paquetes enviados para la resolución y calidad. Para conseguir que funcione correctamente las sub-bandas temporales deben ser decodificadas en orden, para visualizar la secuencia de imágenes de forma real y cronológica.

Como se puede observar en la Figura 26 las primeras imágenes del vídeo son decodificadas a partir de 2 GOP (GOPs abiertos) tras lo que se decodifican las capas en el orden indicado por las flechas.

Para la **escalabilidad en calidad** (Figura 27) las sub-bandas temporales deben ser decodificadas usando el orden **LRCP**. Éste orden se organiza por capas de calidad, resolución, componentes y finalmente precintos. Es el tipo de orden que guarda el mejor compromiso entre la resolución y la calidad en la imagen que realmente es visualizada. De manera, se cuida que no se de el caso de ver una imagen demasiado pequeña con mucha calidad, o demasiado grande con una calidad muy deficiente. Recordemos que realmente la calidad de una imagen consiste en la cantidad de altas frecuencias que transmitimos de ella. A mayor cantidad de éstas, mas detalle tiene el vídeo y, a menor cantidad, se presenta mas suavizado, como desenfocado. En la practica se suele dejar que el propio ancho de banda disponible trunque este número de paquetes.

Se muestra la misma gráfica para la **escalabilidad en calidad y espacial** porque, ambas necesitan la misma ordenación de datos. Eso quiere decir que, si deseamos una reconstrucción por niveles de resolución lo que hay que hacer es usar el orden que genera la escalabilidad en calidad y descartar aquellos datos que pertenecen a subbandas DWT que generan una imagen de mayor resolución que la que se esta reconstruyendo

en ese momento. El orden usado se denomina **RLCP** de las siglas en inglés de *Resolution Layer Component Precint* este tipo de ordenación consiste en que el primer paquete tiene toda la imagen a baja resolución, el segundo y sucesivos continen información que amplia la resolución, hasta llegar a la máxima.

III-K4. Organización de stream: Para concretar la sintaxis de las Figuras 26 y 27 observemos que cada sub-banda temporal ($L^t, H^t, H^{t-1}, \cdots, H^1$) es un stream diferente, comprimido con Motion JPEG 2000 de forma independiente.

Aunque se conoce el número de imágenes en cada sub-banda, cada imagen puede tener distinta peso (Kb). Por lo tanto, con el fin de decodificar sólo una imagen dada, es necesario descomprimir diferentes sub-bandas.

IV. ARQUITECTURA CLIENTE/SERVIDOR PARA EL STREAMING DE VÍDEO

IV-A. La arquitectura cliente/servidor

La arquitectura cliente/servidor (en adelante, arquitectura C/S) es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un cliente realiza peticiones a otro programa, el servidor, que le da respuesta.

En esta arquitectura la capacidad de proceso está repartida entre los clientes y los servidores, aunque son más importantes las ventajas de tipo organizativo debidas a la centralización de la gestión de la información y la separación de responsabilidades, lo que facilita y clarifica el diseño del sistema.

La arquitectura C/S sustituye a la arquitectura monolítica en la que no hay distribución, tanto a nivel físico como a nivel lógico. En ella, el remitente de una solicitud es conocido como cliente. Sus características más importantes podrían resumirse en que:

- Es quien inicia solicitudes o peticiones, tienen por tanto un papel activo en la comunicación (dispositivo maestro o amo).
- Espera y recibe las respuestas del servidor.
- Por lo general, puede conectarse a varios servidores a la vez.
- Normalmente interactúa directamente con los usuarios finales mediante una interfaz gráfica de usuario.
- El cliente presenta una velociad de conexión con el servidor, que estará limitada por la contratación de red que haya echo el usuario con una compañía y por las limitaciones propias del enlace. Por ejemplo: Si el cliente ha contratado un servicio de 6 Mbps este será la capacidad máxima del cliente para recibir datos, además aunque se le proporcionara un ancho de banda superior no se podría alcanzar una velocidad mayor de 10Mb si el cable utilizado para el enlace es, por ejemplo, cable coaxial.

Por otra parte, al receptor de la solicitud enviada por el cliente se conoce como servidor. Sus principales características son:

- Al iniciarse esperan a que lleguen las solicitudes de los clientes. Desempeñan entonces un papel pasivo en la comunicación (dispositivo esclavo).
- Tras la recepción de una solicitud, la procesan y luego envían la respuesta al cliente.
- Por lo general, aceptan conexiones desde un gran número de clientes (aunque en ciertos casos el número máximo de peticiones puede estar limitado por motivos de rendimiento y seguridad).
- No es frecuente que interactúen directamente con los usuarios finales (sin un cliente típico).

Como es lógico la utilización de este tipo de arquitectura presenta una serie de ventajas y de inconvenientes. Entre las primeras destacaríamos:

- **Centralización del control:** los accesos, recursos y la integridad de los datos son controlados por el servidor de forma que un programa cliente defectuoso o no autorizado no pueda dañar el sistema. Esta centralización también facilita la tarea de poner al día datos u otros recursos.
  La centralización del sistemas tiene la cualidad de permitir un control más sencillo, ya que es la mejor forma de captar, manipular y usar la información cuando es necesario que un gran número de usuarios puedan acceder a ella. Otra de sus ventajas es que evita la inconsistencia de las aplicaciones. De este modo, se obtiene una forma sencilla y que requiere poco recursos, para controlar un sistema.
- **Cierto grado de escalabilidad:** se puede aumentar la capacidad de clientes y servidores por separado. Cualquier elemento puede ser aumentado (o mejorado) en cualquier momento, o se pueden añadir nuevos nodos a la red (clientes y/o servidores).
- **Fácil mantenimiento:** al estar distribuidas las funciones y responsabilidades entre varias computadoras independientes, es posible reemplazar, reparar, actualizar, o incluso trasladar un servidor, mientras que sus clientes no se verán afectados por ese cambio (o se afectarán mínimamente). Por lo tanto, podríamos decir que hay una independencia de los cambios.
- **Se trata de una arquitectura muy desarrollada y extensivamente probada:** existen tecnologías, suficientemente desarrolladas, diseñadas para el paradigma de C/S que aseguran la seguridad en las transacciones, la amigabilidad de la interfaz, y la facilidad de empleo.

Por otra parte, es posible encontrar algunas desventajas, que son comentadas a continuación:

- **Existencia de cuellos de botella:** la congestión del tráfico ha sido siempre un problema en el paradigma de C/S. Cuando una gran cantidad de clientes envían peticiones simultaneas al mismo servidor, puede ser que cause muchos problemas para éste (a mayor número de clientes, más problemas para el servidor en relación a consumo de ancho de banda, memoria y CPU). De ahí, que surgieran los sistemas descentralizados.
- **Débil tolerancia a fallos del servidor:** cuando un servidor está caído, las peticiones de los clientes no pueden ser satisfechas. Este problema se soluciona, teniendo un

grupo de servidores en lugar de uno solo.

- **Altos recursos computacionales y de ancho de banda en el lado servidor:** el software y el hardware de un servidor son generalmente muy determinantes. Un hardware regular de un ordenador personal puede no poder servir a cierta cantidad de clientes. Normalmente se necesita software y hardware específico, sobre todo en el lado del servidor, para satisfacer el trabajo. Por supuesto, esto aumentará el coste.
- **Efectos colaterales causados por la seguridad:** el cliente no dispone de los recursos que puedan existir en el servidor. Por ejemplo, si la aplicación es una página Web, no podemos escribir en el disco duro del cliente o imprimir directamente sobre las impresoras sin sacar antes la ventana previa de impresión de los navegadores.

La mayoría de los servicios de Internet son tipo de cliente-servidor. La acción de visitar un sitio Web requiere una arquitectura cliente-servidor, ya que el servidor web sirve las páginas Web al navegador (al cliente). Por ejemplo, al leer un artículo del portal de un periódico de Internet, la computadora y el navegador Web del usuario serían considerados un cliente; y las computadoras, las bases de datos, y los usos que componen el portal serían considerados el servidor. Cuando el navegador Web del usuario solicita un artículo particular del periódico, el servidor Web recopila toda la información a mostrar en la base de datos, la articula en una página Web, y la envía de nuevo al navegador Web del cliente.

IV-B. Hilos, servidores concurrentes e iterativos

Un servidor concurrente atiende a varios clientes al mismo tiempo, más aún, mientras está atendiendo sigue escuchando. El problema es que todo cliente tiene que esperar su turno para ser atendido, si uno de ellos pide un archivo muy grande los demás tienen que esperar. Por lo tanto, podríamos decir que el procedimiento que sigue el servidor es el siguiente:

1. Espera por la llegada de un cliente.
2. Inicia un servidor para manejar los requerimientos del cliente. Esto involucra la creación de un nuevo proceso o hilo. Cuando el cliente se va (termina) el proceso o hilo también termina.
3. Regresa al primer paso.

En el anterior esquema entendemos que un hilo es un flujo de control independiente sobre una sección de código dentro de un proceso que puede ser planificado por el sistema operativo para ejecutarse. Existen dentro de un proceso y para que puedan ser planificadas deben mantener información sobre el flujo de control: puntero a la pila, registros, conjunto de señales pendientes y bloqueadas, propiedades de planificación (como la política y la prioridad) y otros datos específicos de la hebra.

IV-C. La pila de protocolos TCP/IP

Típicamente, el protocolo utilizado en las aplicaciones cliente/servidor el es TCP/IP. Se le denomina conjunto de protocolos TCP/IP, en referencia a los dos protocolos más importantes que la componen: Protocolo de Control de Transmisión (TCP) y Protocolo de Internet (IP), que fueron dos de los primeros en definirse, y que son los más utilizados de la familia.

En la pila de protocolos TCP/IP, TCP es la capa intermedia entre el protocolo de Internet (IP) y la aplicación. Habitualmente, las aplicaciones necesitan que la comunicación sea fiable y, dado que la capa IP aporta un servicio de datagramas no fiable (sin confirmación), TCP añade las funciones necesarias para prestar un servicio que permita que la comunicación entre dos sistemas se efectúe libre de errores, sin pérdidas y con seguridad.

Los servicios provistos por TCP corren en el anfitrión (host) de cualquiera de los extremos de una conexión, no en la red. Por lo tanto, TCP es un protocolo para manejar conexiones de extremo a extremo. Tales conexiones pueden existir a través de una serie de conexiones punto a punto, por lo que estas conexiones extremo-extremo son llamadas en ocasiones, y por simulitud a los circuitos creados en la técnica de conmutación de circuitos, circuitos virtuales.

Las aplicaciones envían flujos de bytes a la capa TCP para ser enviados a la red. TCP divide el flujo de bytes llegado de la aplicación en segmentos de tamaño apropiado (normalmente esta limitación viene impuesta por la unidad máxima de transferencia o MTU (Maximum Transfer Unit) del nivel de enlace de datos de la red a la que la entidad está asociada) y le añade sus cabeceras. Entonces, TCP pasa el segmento resultante a la capa IP, donde a través de la red, llega a la capa TCP de la entidad destino. El TCP comprueba que ningún segmento se ha perdido asignando a cada uno un número de secuencia, que es también usado para asegurarse de que los paquetes han llegado a la entidad destino en el orden correcto. El TCP devuelve un asentimiento (ACK) por bytes que han sido recibidos correctamente; un temporizador en la entidad origen del envío causará un timeout si el asentimiento no es recibido en un tiempo razonable, y el (presuntamente desaparecido) paquete será entonces retransmitido. TCP revisa que no haya bytes dañados durante el envío usando un checksum; es calculado por el emisor en cada paquete antes de ser enviado, y comprobado por el receptor.

Las conexiones TCP se componen de tres etapas: establecimiento de conexión, transferencia de datos y fin de la conexión. Para establecer la conexión se usa el procedimiento llamado negociación en tres pasos (3-way handshake). Una negociación en cuatro pasos (4-way handshake) es usada para la desconexión. Durante el establecimiento de la conexión, algunos parámetros como el número de secuencia son configurados para asegurar la entrega ordenada de los datos y la robustez de la comunicación.

Aunque es posible que un par de entidades finales comiencen una conexión entre ellas simultáneamente, normalmente una de ellas abre un socket en un determinado puerto TCP y se queda a la escucha de nuevas conexiones. Es común referirse a ésto como apertura pasiva, y determina el lado servidor de una conexión. El lado cliente de una conexión realiza una apertura activa de un puerto enviando un paquete SYN inicial al servidor como parte de la negociación en tres pasos. En el lado del servidor se comprueba si el puerto está abierto, es decir, si existe algún proceso escuchando en ese puerto.

En caso de no estarlo, se envía al cliente un paquete de respuesta con el bit RST activado, lo que significa el rechazo del intento de conexión. En caso de que sí se encuentre abierto el puerto, el lado servidor respondería a la petición SYN válida con un paquete SYN/ACK. Finalmente, el cliente debería responderle al servidor con un ACK, completando así la negociación en tres pasos (SYN, SYN/ACK y ACK) y la fase de establecimiento de conexión.

Durante el establecimiento de conexión TCP, los números iniciales de secuencia son intercambiados entre las dos entidades TCP. Estos números de secuencia son usados para identificar los datos dentro del flujo de bytes, y poder identificar (y contar) los bytes de los datos de la aplicación. Siempre hay un par de números de secuencia incluidos en todo segmento TCP, referidos al número de secuencia y al número de asentimiento. Un emisor TCP se refiere a su propio número de secuencia cuando habla de número de secuencia, mientras que con el número de asentimiento se refiere al número de secuencia del receptor. Para mantener la fiabilidad, un receptor asiente los segmentos TCP indicando que ha recibido una parte del flujo continuo de bytes. Una mejora de TCP, llamada asentimiento selectivo (SACK, Selective Acknowledgement) permite a un receptor TCP asentir los datos que se han recibido de tal forma que el remitente solo retransmita los segmentos de datos que faltan.

A través del uso de números de secuencia y asentimiento, TCP puede pasar los segmentos recibidos en el orden correcto dentro del flujo de bytes a la aplicación receptora. Los números de secuencia son de 32 bits (sin signo), que vuelve a cero tras el siguiente byte después del $2^{32} - 1$. Una de las claves para mantener la robustez y la seguridad de las conexiones TCP es la selección del número inicial de secuencia (ISN, Initial Sequence Number).

Los asentimientos (ACKs o Acknowledgments) de los datos enviados o la falta de ellos, son usados por los emisores para interpretar las condiciones de la red entre el emisor y receptor TCP. Unido a los temporizadores, los emisores y receptores TCP pueden alterar el comportamiento del movimiento de datos. El TCP usa una serie de mecanismos para conseguir un alto rendimiento y evitar la congestión de la red (la idea es enviar tan rápido como el receptor pueda recibir). Estos mecanismos incluyen el uso de ventana deslizante, que controla que el transmisor mande información dentro de los límites del buffer del receptor, y algoritmos de control de flujo, tales como el algoritmo de evitación de la congestión (congestion avoidance), el de comienzo lento (slow-start), el de retransmisión rápidda, el de recuperación rápida (Fast Recovery), y otros.

La fase de finalización de la conexión usa una negociación en cuatro pasos (four-way handshake), terminando la conexión desde cada lado independientemente. Cuando uno de los dos extremos de la conexión desea parar su ”mitad” de conexión transmite un paquete FIN, que el otro interlocutor asentirá con un ACK. Por tanto, una desconexión típica requiere un par de segmentos FIN y ACK desde cada lado de la conexión.

A la luz de toda esta información, se ha elegido trabajar con este protocolo principalmente porque:

1. Ofrece una transferencia fiable de datos (control de errores y de flujo).
2. Realiza el proceso de multiplexación, es decir, el TCP distingue procesos dentro del mismo host a partir del puerto en el que escuchan y a quién están escuchando.
3. Es orientado a conexión (antes de transmitir esta se establece).
4. Aunque las aplicaciones leen y escriben un flujo de bytes, el TCP agrupa los bytes en segmentos que son transmitidos como los datagramas UDP. Por tanto, se trata de un protocolo eficiente desde el punto de vista el overhead de las cabeceras.

IV-D. Código Implementado

El lenguaje de programación que hemos utilizado para implementar este sistema, es el lenguaje C. C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B, a su vez basado en BCPL. Se trata de un lenguaje débilmente tipificado de medio nivel pero con muchas características de bajo nivel. Dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel.

Uno de los objetivos de diseño del lenguaje C es que sólo sean necesarias unas pocas instrucciones en lenguaje máquina para traducir cada elemento del lenguaje, sin que haga falta un soporte intenso en tiempo de ejecución. Es muy posible escribir C a bajo nivel de abstracción; de hecho, C se usó como intermediario entre diferentes lenguajes.

En parte a causa de ser de relativamente bajo nivel y de tener un modesto conjunto de características, se pueden desarrollar compiladores de C fácilmente. En consecuencia, el lenguaje C está disponible en un amplio abanico de plataformas (seguramente más que cualquier otro lenguaje). Además, a pesar de su naturaleza de bajo nivel, el lenguaje se desarrolló para incentivar la programación independiente de la máquina. Un programa escrito cumpliendo los estándares e intentando que sea portátil puede compilarse en muchos computadores.

C se desarrolló originalmente (conjuntamente con el sistema operativo Unix, con el que ha estado asociado mucho tiempo) por programadores para programadores. Sin embargo, ha alcanzado una popularidad enorme, y se ha usado en contextos muy alejados de la programación de sistemas, para la que se diseñó originalmente.

La implementación de sistemas empleando lenguaje de programación C, tiene una serie de ventajas para el programador:

1. Un núcleo del lenguaje simple, con funcionalidades añadidas importantes, como funciones matemáticas y de manejo de archivos, proporcionadas por bibliotecas.
2. Acceso a memoria de bajo nivel mediante el uso de punteros.
3. Punteros a funciones y variables estáticas, que permiten una forma rudimentaria de encapsulado y polimorfismo.
4. Un sistema de tipos que impide operaciones sin sentido.

Como es lógico, también C presenta una serie de carencias e incovenientes tales como:

1. El mayor problema que presenta el lenguaje C frente a los lenguajes de tipo de dato dinámico es la gran diferencia en velocidad de desarrollo: es más lento programar en C. La razón estriba en que el compilador de C se limita a traducir código sin apenas añadir nada.
2. En C el programador ha de reservar y liberar la memoria explícitamente. En otros lenguajes (como BASIC, Matlab o Java) la memoria es gestionada de forma transparente para el programador.
3. El mantenimiento también es más difícil y costoso que con lenguajes de más alto nivel. El código en C se presta a sentencias cortas y enrevesadas de difícil interpretación.
4. C no dispone de sistemas de control automáticos y la seguridad depende casi exclusivamente de la experiencia del programador.
5. No existe recolección de basura nativa.

Por supuesto, las ventajas de la implementación de código C superan sus inconvenientes, principalmente porque es un lenguaje muy eficiente, puesto que es posible utilizar sus características de bajo nivel para realizar implementaciones óptimas. Además, a pesar de su bajo nivel es el lenguaje más portado en existencia y proporciona facilidades para realizar programas modulares y/o utilizar código o bibliotecas existentes.

V. PROPUESTA

Ahora se va a explicar como se ha llegado a la solución de este proyecto. El cliente, a la hora de ejecutarse para establecer la conexión con el servidor, debe de introducir por teclado la IP del servidor, por lo que se ha guardado en la variable `server_host` la IP. Una vez introducido la IP, se le pregunta al DNS por la dirección de destino, en este caso el servidor. En el caso de que el DNS nos devuelva una dirección nula se interrumpe la conexión. El código que realiza esta funcionalidad se muestra a continuación:

```
1  /* Preguntamos al DNS por la dir IP destino. */
2  struct hostent *receiver_IP;
3  if ((receiver_IP = gethostbyname(server_host)) == ↩
       NULL) {
4    perror("gethostbyname");
5    exit(EXIT_FAILURE);
6  }
```

Una vez que sepamos la dirección de destino calculamos el número del protocolo del tcp con la función "getprotobyname". Ya, con el valor del protocolo, pasamos a crear el socket descriptor. Para ello usamos la función "socket", donde los parámetros que se les pasa son los siguientes:

- AF_INET: Protocolo de Internet IPv4
- SOCK_STREAM: Proporciona secuenciado, confiable, de dos vías, basado en la conexión secuencias de bytes. Un mecanismo fuera de banda de transmisión de datos puede ser apoyada.
- ptrp->_proto: Número de protocolo de tcp.

Si no se crea el socket descriptor el programa deja de ejecutarse. Con el socket descriptor creado ahora pasamos a especificar el host y el puerto de destino de los segmentos. Creamos una variable de tipo estructura "sockaddr_in", borramos los datos de la estructura y añadimos los siguientes valores:

- Internet.
- IP de destino de los segmentos.
- Puerto de destino de los segmentos.

```
1  int sd; /* Socket descriptor */ {
2
3    /* Obtenemos el n'umero del protocolo. */
4    struct protoent *ptrp; /* Pointer to a protocol ↩
         table entry. */
5    ptrp = getprotobyname("tcp");
6    if(ptrp == NULL) {
7      perror("getprotobyname");
8      exit(EXIT_FAILURE);
9    }
10
11   /* Creamos el socket descriptor. */
12   sd = socket(AF_INET, SOCK_STREAM, ptrp->p_proto);
13   if(sd < 0) {
14     perror("socket");
15     exit(EXIT_FAILURE);
16  }
17 }
18
19 /* Si pulsamos CRTL+C, el programa acaba ejecutando ↩
       la funci'on end(). */
20 void end() {
21   fprintf(stderr," %s:␣CTRL+C␣detected.␣Exiting␣...␣"↩
         , argv[0]);
22   close(sd);
23   fprintf(stderr,"done\n");
24   exit(EXIT_SUCCESS);
25 }
26
27 /* Activamos la senal SIGINT. */
28 signal(SIGINT, end);
29
30 /* Especificamos el host y puerto destino de los ↩
       segmentos. */
31 struct sockaddr_in socket_addr; {
32   /* Borramos la estructura. */
33   memset((char  *)&socket_addr,0,sizeof(socket_addr)↩
         );
34   /* Usaremos Internet. */
35   socket_addr.sin_family = AF_INET;
36   /* IP destino de los segmentos */
37   socket_addr.sin_addr = *((struct in_addr *)↩
         receiver_IP->h_addr);
38   /* Puerto destino de los segmentos */
39   socket_addr.sin_port = htons(server_port);
40 }
41
42 /* Nos conectamos al receptor. */
43 connect(sd, (const struct sockaddr *)&socket_addr, ↩
       sizeof(socket_addr));
```

El código que realiza esta funcionalidad se muestra a continuación:

Ahora el cliente puede establecer la conexión con el servidor mediante la función "connect", siendo los parámetros de entrada:

- El socket descriptor.
- La estructura con el host y el puerto de destino.
- El tamaño de la estructura anterior.

Una vez que el cliente haya establecido la conexión con el servidor solo tenemos que recibir los datos que este nos envíe. Si recibimos todas las capas solo es necesario escribir en disco lo que recibamos, pero en el caso de que no tengamos suficiente ancho de banda para recibir una capa entonces el cliente rellenará esa/s capa/s con imágenes en grises. Primero vamos a comentar las variables que han sido necesarias para gestionar el control de las capas.

- *buffer: Cadena donde se almacena los datos que recibimos del servidor.
- tamano_nulo: Entero donde se guarda el total de bytes que ocupa la imagen nula de los vídeos High.
- tamano_motion_nulo: Entero donde se guarda el total de bytes que ocupa la imagen nula de los vídeos de movimiento.
- bytes_read: Entero que controla el número de bytes que recibimos del servidor.
- contador_capa, contador_maximo, contador_aux: Enteros que gestionan el control de las capas de las que se compone el vídeo que el cliente va a recibir y las capas que es capaz de recibir.
- salir: Variable que avisa al cliente cuando ha recibido todos los vídeos por parte del servidor.
- tamano_vídeo: Entero que almacena el total de bytes que ocupa el vídeo que va a recibir..
- i: Entero usado para el control de bucles.
- j: Entero para controlar el número de GOPs.
- comienzo, final, tiempo_1, tiempo_2, ini, fin: Variables para calcular el tiempo y controlar la gestión de capas.
- diferencia: Número real de doble precisión para guardar la diferencia de tiempo.
- repetir: Entero que guarda el número de repeticiones para incrustar imágenes nulas en todos los vídeos.
- ruptura: Entero que finaliza el código cuando vale 1 en la gestión de imágenes nulas.

El código que realiza esta funcionalidad se muestra a continuación:

```
1   /* Asignamos memoria para almacenar los datos ↩
        entrantes. */
2
3   char *buffer = (char *)malloc(payload_size*sizeof(↩
        char));            //Guardamos el video que ↩
        recibimos del servidor
4   int tamano_nulo, tamano_motion_nulo; //Guarda el ↩
        tamano de la imagen nula y de la imagen nula ↩
        de los movimientos
5   int bytes_read = 0; //Se guardan el numero de ↩
        bytes que recibimos del servidor
6   int contador_capa, contador_maximo, contador_aux;
7   int salir = 0;        //Cuando el servidor haya ↩
        terminado de enviar todos los videos nos ↩
        devuelve un valor indicando que terminemos la ↩
        conexion
8   int tamano_video = 0;
9   int i;
10  int j = 0;
11  struct timeval comienzo, final, tiempo_1, tiempo_2↩
        , ini, fin;
12  double diferencia;
13  int repetir;
14  int ruptura = 0; //Variable para finalizar el ↩
        programa en la incrustacion de imagenes nulas
```

Lo primero que realiza el cliente una vez establecida la conexión es recibir del servidor el número de capas del que se compone el vídeo que queremos recibir. Para ello usamos la función recv, que se compone de las siguientes variables:

- El descriptor del socket creado anteriormente.
- El buffer que contiene el valor del total de capas.
- El número de bytes a escribir en el socket.
- La opción de envío. Normalmente se usa el valor 0.

Una vez que hayamos recibido el valor del total de capas guardamos ese valor en las variables ”contador_maximo” y ”contador_capa” para usarlos mas adelantes. El código que realiza esta funcionalidad se muestra a continuación:

```
1   // Primero recibimos el numero de capas del que se↩
         compone el video que queremos reproducir
2   recv (sd, (void *)&contador_aux, sizeof(↩
        contador_aux), 0);
3   contador_maximo = contador_aux;
4   contador_capa = contador_aux;
```

Ahora crearemos unas cadenas que contienen los nombres básicos que contienen cada capa de vídeo, una cadena que guarda el nombre completo del vídeo y el numero del vídeo con el que estamos trabajando. El código que realiza esta funcionalidad se muestra a continuación:

```
1   /* Creamos las cadenas con los nombres de los ↩
        archivos */
2   char fr[] = "frame_types_";
3   char mr[] = "motion_residue_";
4   char high[] = "high_";
5   char mjc[] = ".mjc";
6   char nombre[22];
7   char num_video;
8   int numero = 0;              //Se guarda el ↩
        numero del archivo
```

Para la creación de los nombres se usa la misma cadena, y como cada nombre de los vídeos poseen diferentes longitudes se usará una función que vaciará dicha cadena para evitar introducir basura en el nombre y que se pueda producir algún error. El nombre de dicha función es ”Vacia_Cadena_Peq” donde le pasamos como parámetro de entrada la cadena. La función recorre toda la cadena insertando en cada posición el carácter vacío ”\0”. El código que realiza esta funcionalidad se muestra a continuación:

```
1  //Funcion para vaciar la cadena con el nombre del ↩
       archivo
2  void Vacia_Cadena_Peq (char *cadena){
3    int i;
4
5    for (i=0;i<22;i++){
6      cadena[i]='\0';
7    }
8  }
```

Con los nombres básicos de los archivos pasamos a crear todos los archivos necesarios para guardar los vídeos que queremos recibir del servidor. Para ello se ha desarrollado el siguiente método:

- Creamos una cadena de archivos con un tamaño igual al multiplicar el total de capas menos 1 por 3 (cada capa se compone de 3 vídeos) y sumándole 1 (el primer vídeo que recibimos, GOP 0). Hay que restarle 1 debido a que la capa 1 (low_X.mjc) solo se compone de un archivo mientras que el resto se compone de 3 archivos.
- Vaciamos la cadena donde se almacena el nombre total del archivo para evitar almacenar un dato no deseado mediante la función ”Vacia_Cadena_Peq”.
- En el primer archivo se almacenará el archivo de vídeo ”low_X.mjc”, abierto como archivo binario en escritura. Después se creara un bucle for de tamaño igual al total de capas a almacenar ya que en cada capa los 3 vídeos que se guardan tienen el mismo nombre excepto cambiando el número al final.
- En la primera parte del bucle se crearan los archivos con nombre ”frame_types_X”, donde el valor de X se

calculará del siguiente modo: 1 + 3*i. En la primera vuelta se almacenara el valor 1, en la siguiente el valor 4, y así sucesivamente hasta llegar al final del bucle. Después vaciamos los datos que contenga la cadena del nombre del archivo.

- En la segunda parte del bucle se crearán los nombres de los archivos referidos a ”motion_residue_X.mjc”, donde el valor de X se calculará de igual manera que en caso anterior excepto que el número calculado se obtiene del siguiente modo: 2+3*i. Después vaciamos los datos que contenga la cadena del nombre del archivo.
- En la tercera parte del bucle se crearán los nombres de los archivos referidos a ”high_X.mjc”, donde el valor de X se calculará de igual manera que en caso anterior excepto que el número calculado se obtiene del siguiente modo: 3+3*i. Después vaciamos los datos que contenga la cadena del nombre del archivo.

El código que realiza esta funcionalidad se muestra a continuación:

```
1  /* Creamos y abrimos los archivos */
2    FILE *archivo[(contador_maximo-1)*3+1];
3    Vacia_Cadena_Peq(nombre);
4    sprintf(nombre, "low_%d%s", contador_maximo-1, mjc↩
         );
5    archivo[0]= fopen(nombre,"wb");
6    Vacia_Cadena_Peq(nombre);
7    for (i=0; i<contador_maximo-1; i++){
8      sprintf(nombre, "%s%d", fr, contador_capa-i-1);
9      numero = 1 + 3*i;
10     archivo[numero] = fopen(nombre,"wb");
11     if (archivo[numero] == NULL){
12       printf ("Ha fallado el archivo %s.", nombre);
13       abort;
14     }
15     Vacia_Cadena_Peq(nombre);
16     sprintf(nombre, "%s%d%s", mr, contador_capa-i-1,↩
           mjc);
17     numero = 2 + 3*i;
18     archivo[numero] = fopen(nombre, "wb");
19     if (archivo[numero] == NULL){
20       printf ("Ha fallado el archivo %s.", nombre);
21       abort;
22     }
23     Vacia_Cadena_Peq(nombre);
24     sprintf(nombre, "%s%d%s", high, contador_capa-i↩
          -1, mjc);
25     numero = 3 + 3*i;
26     archivo[numero] = fopen(nombre, "wb");
27     if (archivo[numero] == NULL){
28       printf ("Ha fallado el archivo %s.", nombre);
29       abort;
30     }
31     Vacia_Cadena_Peq(nombre);
32   }
```

Vamos a crear también un archivo ”tiempo.txt” que se abrirá en modo escritura binaria (wb) donde se va a guardar el tiempo que a tardado en recibirse cada GOP. Declaramos 2 cadenas, una de tamaño 9 donde se guardará el tiempo y un carácter que contiene el símbolo de salto de línea que se añade al final de la cadena anterior.

El código que realiza esta funcionalidad se muestra a continuación:

```
1  FILE *tiempo;
2    tiempo=fopen("tiempo.txt","wb");
3    char cadena[9];
4    char salto='\n';
```

Con el número máximo de capas que podemos recibir y con los archivos necesarios creados el cliente puede empezar a recibir los vídeos del servidor y almacenarlos en memoria. Antes de nada explicar que se ha implementado una función genérica que según el vídeo que queramos almacenar en disco se encarga de recibir sus datos y luego almacenarlos. Esta función se llama ”Guarda_Imagen” y los datos usados para la función son los siguientes:

- Un buffer donde se almacenará los datos del vídeo.
- El archivo donde queremos que se guarden los datos del vídeo.
- El socket que nos enlaza al servidor.

Ahora vamos a ver por dentro como funciona la función: Primero se declaran 5 variables como enteros, que inicializaremos a 0:

- total_bytes: Se almacena los bytes que ocupa el vídeo que vamos a recibir.
- bytes_read: Se almacena los bytes del vídeo para luego ir escribiéndolo en disco. Iremos escribiendo en disco partes de 512 bytes del vídeo y el último sera igual o menor de 512.
- datos: Variable que controla el bucle donde el cliente se queda recibiendo un paquete de datos, ya sea del tamaño del payload o el tamaño restante del vídeo.
- tamano_max: Se guarda el tamaño en bytes de lo que el cliente va a recibir.
- tamano: Se guarda el total de bytes que ocupa el vídeo para luego devolverlo.

Con las variables inicializadas el cliente recibe los datos del vídeo mediante la función recv y se almacenan en la variable ”total_bytes”, y estos datos se guardaran también en la variable ”tamano”. Mostramos por pantalla el total de bytes que vamos a recibir para así poder comprobar que vamos a recibir el tamaño correcto, combprobándolo con el servidor. Ahora entramos en un bucle while donde se va a repetir mientras que hayamos recibido todos los datos del vídeo. Al comienzo de este bucle se va a llamar a una condición ”si ... sino ...” donde se dan 2 casos:

- tamano $<$ payload_size: Si el tamaño que queda por recibir es menor que el tamaño del payload (512 bytes) quiere decir que en tamano_max se almacena el valor de tamano, ya que este va a ser la última recepción.
- tamano $>$ payload_size : Si el tamaño que queda por recibir es mayor que el tamaño del payload (512 bytes) entonces en tamano_max se almacena el valor del payload.

Después de saber la cantidad de datos que se van a recibir inicializamos la variable ”datos” a 0, y entonces entramos en un bucle while donde se va a controlar la recepción de datos. La condición que se establece es la siguiente: datos $<$ tamano_max, es decir, que mientras no hayamos recibido los datos esperados repita el bucle. Lo primero que hacemos dentro del bucle es recibir los datos mediante la función recv donde los parámetros usados son:

- sd: el socket
- (void *)buffer+datos: el buffer donde se van a almacenar los datos recibidos, aparte se le suma la variable datos para posicionarse en el lugar del buffer donde tiene que almacenar los datos.

- tamano_max-datos: el total de datos que se van a almacenar en el buffer y le restamos la cantidad de datos que hemos recibido.
- 0: El valor de la bandera.

Una vez recibido los datos los mostramos por pantalla para comprobar que cantidad hemos recibido. Después le sumamos a la variable "datos" el total de bytes que hemos recibido.

El motivo por el que se le suma al buffer la variable "datos" y se le resta a la variable "tamano_max" se debe a la siguiente caso:

Supongamos que el valor de tamano_max es de 512 bytes, y al inicio la variable datos vale 0. En el caso de que recibamos los 512 bytes solo se repetiría el bucle una vez, pero en el caso de que recibamos menos datos, por ejemplo 400 bytes, tenemos que tener en cuenta que los datos que recibamos se van a almacenar a partir de la posición 401 del buffer, ya que los bytes anteriores ya se habían escrito en el archivo, aparte, en vez de almacenar 512 bytes se van a almacenar 112 bytes (512 - 400).

Cuando se halla recibido el paquete pasamos entonces a escribirlo en el archivo mediante la función "fwrite", donde los parámetros de entrada son los siguientes:

- El buffer que almacena los datos recibidos.
- El tamaño de los datos que recibimos.
- El número de bytes que hemos recibido.
- El archivo en el que se va a escribir.

Una vez que haya terminado este bucle a la variable tamano se le restan los datos recibidos y luego vuelve al comienzo del primer bucle donde se comprobará si se han recibido o no todos los datos. En el caso de que se haya recibido todo limpiamos el buffer.

El código que realiza esta funcionalidad se muestra a continuación:

```
1  //Funcion para recibir y escribir en disco una ↩
       imagen
2  int Guarda_Imagen(char *buffer, FILE *archivo, int ↩
       sd){
3    int total_bytes = 0;
4    int bytes_read = 0;
5    int datos;
6    int tamano_max;       //Tamano del paquete a ↩
       recibir
7    int tamano;   //Tamano total del archivo a recibir
8
9    recv(sd, (void *) &total_bytes, sizeof(total_bytes↩
       ), 0);
10   tamano = total_bytes;
11   fprintf(stderr, "␣Total␣bytes:␣%d\n", total_bytes)↩
       ;
12   fflush(stderr);
13   while(tamano > 0){
14     if (tamano < payload_size){
15       tamano_max = tamano;
16     }else{
17       tamano_max = payload_size;
18     }
19     datos = 0;
20     while (datos < tamano_max){
21       bytes_read = recv(sd, (void *)buffer+datos, ↩
           tamano_max-datos, 0);
22       fprintf(stderr,"␣␣␣Bytes␣recibidos:␣%d\n", ↩
           bytes_read);
23       datos = datos + bytes_read;
24     }
25     fwrite((void *)buffer, sizeof(char), tamano_max,↩
         archivo);  //Se escribe en disco el ↩
         segmento final del video
26     tamano = tamano - datos;
27   }
28   Vacia_Cadena(buffer);
29   return tamano;
30 }
```

Lo siguiente que vamos a explicar es la creación de los buffers que contienen las imágenes nulas de los archivos. Las variables creadas son las siguientes:

- nombre_nulo: Cadena que contiene el nombre del archivo con la imagen nula de los vídeos low_X y high_X. El nombre de dicho archivo es "nula.j2c".
- nombre_motion_nulo: Cadena que contiene el nombre del archivo con la imagen nula de los vídeos motion_residue_X. El nombre de dicho archivo es "motion_nula.j2c".
- buffer_nulo: Buffer que contiene la imagen nula de los vídeos low_X y high_X.
- buffer_motion_nulo: Buffer que contiene la imagen nula de los vídeos motion_residue_X.

Lo siguiente a explicar son las funciones usadas para guardar las imágenes nulas en sus respectivos buffers. Primero debemos obtener el tamaño que ocupan dichas imágenes nulas, por lo que llamamos a la función "Tamano_Nulo" cuyo parámetro de entrada es la imagen nula de la cual queramos saber su tamaño. Dentro de la función declaramos la variable "tamaño" como entero, que va a ser el que nos va a devolver el tamaño de la imagen nula. Después declaramos la variable "archivo" para abrir el archivo con el nombre del archivo con el que llamamos a la función en modo lectura binaria (rb). Después usamos la función "fseek" para posicionarnos al final del archivo y al final guardamos el tamaño de la imagen con ftell, ya que esta nos devuelve la posición actual del archivo. Para finalizar devolvemos dicho tamaño. El código que realiza esta funcionalidad se muestra a continuación:

```
1  //Funcion para obtener el tamano de la imagen nula
2  int Tamano_Nulo(char cadena[16]){
3    int tamano;
4
5    FILE *archivo;
6    archivo = fopen(cadena,"rb");
7    fseek(archivo, 0, SEEK_END);
8    tamano = ftell(archivo);
9
10   return tamano;
11 }
```

Esta función la llamamos para los archivos que contienen la imagen nula. Cuando sepamos el tamaño de cada una lo siguiente que debemos realizar es guardar cada una de las imágenes en su respectivo buffer. La función que se encarga de guardar la imagen en el buffer se llama "Crear_Nulo", donde los parámetros de entrada son el buffer donde vamos a guardar la imagen y el nombre de archivo.

Dentro de la función declaramos la variable archivo que abrirá el archivo y la variable "i" como entero para el control del bucle, inicializado a 0. Abrimos de nuevo el archivo en modo lectura binaria (rb) y nos posicionamos al principio del archivo con la función fseek. Cogemos el primer carácter del archivo con getc y lo guardamos en el buffer y entra en un bucle while donde estará cogiendo el siguiente carácter del archivo y guardándolo en el buffer hasta llegar al final. Cuando finalice el bucle salimos de la función. El código que realiza esta funcionalidad se muestra a continuación:

```c
//Funcion para crear el buffer con una imagen nula
void Crear_Nulo(char *buffer, char cadena[16]){
  FILE *archivo;
  int i = 0;

  archivo = fopen(cadena,"rb");
  fseek(archivo, 0, SEEK_SET);
  buffer[i] = getc(archivo);
  while (!feof(archivo)){
    i++;
    buffer[i] = getc(archivo);
  }
}
```

La explicación de la creación de los buffers con las imágenes nulas se ve en el siguiente código:

```c
        //En esta parte vamos a crear el buffer que va a contener la imagen nula y la imagen nula de los archivos de movimiento (motion)
        char nombre_nulo[] = "nula.j2c";
        char nombre_motion_nulo[] = "motion_nula.j2c";
        tamano_nulo = Tamano_Nulo(nombre_nulo);
        char *buffer_nulo = (char *)malloc(tamano_nulo*sizeof(char));            //Guardamos una imagen nula
        Crear_Nulo (buffer_nulo, nombre_nulo); //Se crea una imagen nula antes de empezar ha recibir, donde se usara en aquellas capas que el cliente no pueda recibir
        tamano_motion_nulo = Tamano_Nulo(nombre_motion_nulo);
        char *buffer_motion_nulo = (char *)malloc(tamano_motion_nulo*sizeof(char));             //Guardamos una imagen nula de los archivos de movimiento
        Crear_Nulo (buffer_motion_nulo, nombre_motion_nulo); //Se crea una imagen nula de los movimientos antes de empezar ha recibir, donde se usara en aquellas capas que     el cliente no pueda recibir
```

Por último se va a explicar el funcionamiento de las capas. Según el tipo de vídeo que queramos recibir puede tener x capas, aunque por defecto MCJ2K comprime el vídeo en 6 capas, que son con las cuales vamos a trabajar. El vídeo lo vamos a recibir, para dividirlo, en 2 partes:

- La primera parte representa la primera imagen que vamos a recibir (GOP 0), perteneciente al vídeo "Low_X.mjc" Aquí se hará la primera gestión del ancho de banda disponible.
- La segunda parte representa a las capas. Según la fórmula con la que calculamos el número de capas que el cliente va a poder recibir a continuación de su recepción actual se enviaran todas las capas, algunas intermedias o ninguna capa, según el ancho de banda del que se disponga. En este ejemplo se trabajar con 6 capas, siendo mas importantes de enviar los vídeos referentes a "high_5" hasta llegar a los vídeos de "high_1".

Antes vamos a explicar las funciones que se encargan de incrustar las imágenes nulas en los vídeos. Existen 2 casos:

- Lo que recibe el cliente son o los vídeos low o high o los vectores de movimiento, donde en estos la función que usaremos se llama "Escribir_Nulo", donde los parámetros de entrada son los siguientes:
  - El archivo del vídeo en el cual queremos insertar la imagen nula.
  - El buffer que contiene la imagen nula.
  - El número de imágenes nulas que queremos incrustar.
  - El tamaño de la imagen nula.
- Lo que recibe el cliente son los frame_types. Estos archivos contienen carácteres, que pueden ser el carácter "I" o "B", indicando si existe o no movimiento entre las imágenes respectivamente, pero cuando el cliente no puede recibir nada insertará el carácter "B". La función que usaremos se llama "Escribir_Frames", donde los parámetros de entrada son los siguientes:
  - El archivo del frame_types en el que se inserta los carácteres.
  - El buffer donde se insertan los carácteres.
  - El número de carácteres a insertar en el buffer.

En la función "Escribir_Nulo" declaramos la variable "i" como entero para el control del bucle for, donde llamará a la función "fwrite" tantas veces como imágenes queramos insertar. A dicha función le pasamos los siguientes parámetros:

- El buffer que contiene la imagen nula.
- El tipo de dato, en este caso es de tipo carácter.
- El tamaño de la imagen nula.
- El archivo en el que queremos incrustar la imagen nula.

El código que realiza esta funcionalidad se muestra a continuación:

```c
//Funcion para escribir x imagenes nulas en un buffer
void Escribir_Nulo(FILE *archivo, char *buffer, int rep, int num){
  int i;

  for (i=0; i<rep; i++){
    fwrite((void *)buffer, sizeof(char), num, archivo);
  }
}
```

En la función "Escribir_Frames" declaramos la variable "i" como entero para el control del bucle for, donde insertara en el buffer tantas letras queramos insertar. Una vez tenga el buffer los datos llamamos escribimos el buffer en su respectivo archivo con "fwrite" usando como parámetros de entrada:

- El buffer que contiene los carácteres.
- El tipo de dato, en este caso es de tipo carácter.
- El tamaño del buffer, en este caso el número de carácteres insertadas.
- El archivo en el que queremos incrustar los carácteres.

El código que realiza esta funcionalidad se muestra a continuación:

```c
//Funcion para escribir x caracteres en los frame_types_X
void Escribir_Frames(FILE *archivo, char *buffer, int num){
  int i;

  for (i=0; i<num; i++){
    buffer[i] = 'B';
  }
  fwrite((void *)buffer, sizeof(char), num, archivo);
}
```

Con todas las funciones explicadas pasamos a la explicación de la recepción de los vídeos. Lo primero que vamos a realizar será tomar el tiempo en el que comienza la recepción total del

cliente y el tiempo el que comienza a recibirse la primera imagen, cuyos tiempo los guardamos en las variables ”ini” y ”tiempo_1” respectivamente. Dichos tiempos se obtienen con la función ”gettimeofday” que nos devuelve el tiempo en segundos y milisegundos. Después llamamos a la función ”Guarda_Imagen” donde vamos donde vamos a recibir los datos de low_X perteneciente al GOP 0. Una vez recibido el vídeo y escrito en su respectivo archivo tomamos el tiempo que ha tardado en recibirse en la variable ”tiempo_2”. Imprimimos por pantalla ”rlX”, que quiere decir que se ha recibido el vídeo ”Low_X”. Luego calculamos la diferencia de tiempo y lo guardamos en la variable ”diferencia”. Hay que tener en cuenta que, aunque la función ”gettimeofday” nos devuelva los milisegundos tenemos que pasarlo a dicho tamaño por lo que, en el caso de tiempo_2, el tiempo en milisegundos que se guarda en ”tiempo_2.tv_usec” se multiplica por 0.000001. Cuando tengamos la diferencia debemos dividirla entre el tiempo que tarda teóricamente en recibirse una imagen, que es 1/30 segundos. Si dicha diferencia es mayor que 1 es que nuestro ancho de banda era insuficiente como para recibir la primera imagen, por lo que vamos a tener que insertar imágenes nulas en todos los vídeos. Para ello la diferencia la vamos a redondear a lo bajo mediante la función ”floor” ya que dicho redondeo nos devuelve el número de veces que tenemos que insertar imágenes nulas en todos los vídeos. Inicializamos contador_capa a 0 y nos metemos en un bucle while donde se repetirá mientras que el valor de diferencia sea mayor que 0. Dentro del bucle lo primero que hacemos es enviarle el valor de contador_capa al servidor para decirle que no tenemos suficiente ancho de banda para recibir los vídeos y ahora entramos en un bucle for que recorrerá todos los vídeos. En la primera vuelta solo insertará una imagen nula en el vídeo ”Low_X.mjc”, ya que puede ser el caso en el que el vídeo solo haya sido comprimido con 1 capa y solo se vaya a recibir ”Low_0.mjc”. Primero limpiamos el buffer con la función ”Vacia_Cadena”, que es similar a la función ”Vacia_Cadena_Peq”, pero en este caso limpia el buffer de las imágenes que posee un tamaño de 512. Luego llamamos a la función ”Escribir_Nulo” pasándole como entrada el archivo[0] (Low_X.mcj), el buffer con la imagen nula, el número de imágenes nulas que queremos, en este caso solo 1, y el tamaño de la imagen nula, escribimos por pantalla que se ha escrito una imagen nula. En el caso de que tengamos más de una capa entonces debemos insertar imágenes nulas en los frame_types, motion_residue y high.
El orden para incrustar las imágenes nulas en el resto de los vídeos es el siguiente:

- Limpiamos el buffer.
- Insertamos los carácteres en los frame_types_X, por lo que llamamos a la función ”Escribir_Frames” como parámetro de entrada ”archivo[1+3*(i-1)]”, que son los archivos de los frame_types, el buffer y el número de carácteres que queremos incrustar ”$(1 << (i-1))$”, esto es lo mismo que $2^{(i-1)}$.
- Limpiamos el buffer.
- Insertamos las imágenes nulas en los vídeos motion_residue_X.mjc, por lo que llamamos a la función ”Escribir_Frames” como parámetros de entrada ”archivo[2+3*(i-1)]”, el buffer con la imagen nula del vector de movimiento, el número de imágenes nulas que queremos incrustar ”$(1 << (i-1))$” y el tamaño de la imagen nula.
- Insertamos las imágenes nulas en los vídeos high_X.mjc, por lo que llamamos a la función ”Escribir_Frames” como parámetros de entrada ”archivo[3+3*(i-1)]”, el buffer con la imagen nula de los vídeos normales, el número de imágenes nulas que queremos incrustar ”$(1 << (i-1))$” y el tamaño de la imagen nula.

Cuando se hayan insertado las imágenes nulas en todos los vídeos decrementamos el valor de diferencia a 1 e incrementamos el valor de j a 1 (el número de GOPs que vamos a recibir) y volvemos a repetir el bucle hasta que diferencia llegue a 0. Al salir del bucle while incrementamos el valor de contador_capa a 1 y le enviamos dicho valor al servidor mediante un send para indicarle de que volvemos a tener ancho de banda para poder recibir la capa 1 (Low_X.mjc”).
En el caso de que si tuviésemos suficiente ancho de banda entonces le enviamos al servidor el valor inicial de contador_capa, es decir, el número ma’ximo de capas del que se compone el vídeo e incrementamos el valor de j a 1. Para finalizar mostramos por pantalla el tiempo que ha tardado en recibirse el GOP 0.

El código que realiza esta funcionalidad se muestra a continuación:

```
gettimeofday(&ini, NULL);
/* Recibimos un segmento que representa al GOP 0. */
gettimeofday(&tiempo_1, NULL);
tamano_video = Guarda_Imagen(buffer, archivo[0], sd);
gettimeofday(&tiempo_2, NULL);
fprintf(stderr,"  rl %d\n",contador_capa);
fflush(stderr);
diferencia = (tiempo_2.tv_sec+tiempo_2.tv_usec*0.000001)-(tiempo_1.tv_sec+tiempo_1.tv_usec*0.000001);
diferencia = diferencia/((1<<0)/30.000000); //El GOP 0 solo tiene una imagen, por lo que siempre se divide entre 1/30 de segundo
if (diferencia > 1){    //No hemos tenido suficiente ancho de banda para recibir el video low_X, por lo que se debe meter tantas imagenes nulas
 diferencia = floorf(diferencia);
 contador_capa = 0;
 while (diferencia > 0){
  send (sd, (void *)&contador_capa, sizeof(contador_capa), 0);
  for (i=0; i<contador_maximo; i++){
    if (i == 0){
      Vacia_Cadena(buffer);
      Escribir_Nulo(archivo[0], buffer_nulo, 1, tamano_nulo);
      fprintf(stderr,"\nSe ha impreso una imagen nula.");
    }else{
      Vacia_Cadena(buffer);
      Escribir_Frames(archivo[1+3*(i-1)], buffer, 1<<(i-1));
      Vacia_Cadena(buffer);
      Escribir_Nulo(archivo[2+3*(i-1)], buffer_motion_nulo, 1<<(i-1), tamano_motion_nulo);
      Escribir_Nulo(archivo[3+3*(i-1)], buffer_nulo, 1<<(i-1), tamano_nulo);
      fprintf(stderr,"\nSe ha impreso una imagen nula.");
    }
  }
  diferencia--;
  j++;
```

```
31     }
32     //Ya se ha incrustado todas las imagenes nulas, por↩
           lo que se le dice al servidor que nos envie ↩
           una capa.
33     contador_capa++;
34     send (sd, (void *)&contador_capa, sizeof(↩
           contador_capa), 0);
35    }else{
36     send (sd, (void *)&contador_capa, sizeof(↩
           contador_capa), 0);
37     j = 1;         //Esta variable se usa para ↩
           controlar el Gop que se esta enviando
38    }
39   printf("\nEl Gop 0 ha tardado en recibirse %lf\n", ↩
          diferencia);
```

Una vez recibido el GOP 0 pasamos a explicar el bucle que gestiona el control de las capas, donde se entrara a un bucle for hasta que no recibamos todos los vídeos. Lo primero que hacemos antes de recibir los vídeos es obtener el tiempo de inicio de las transmisiones en la variable ”comienzo”.

Para la gestión de las capas tenemos que tener en cuenta dos casos, según el ancho de banda del que disponga el cliente:

- Las capas que el cliente pueda recibir mediante la comunicación con el servidor.
- Las capas que el servidor no va a enviarnos por la falta de ancho de banda, donde el cliente deberá rellenar esas capas con imágenes nulas.

Para ambos casos se van a usar un bucle ”for”. El primer bucle va desde 0 hasta la capa que el cliente pueda recibir, mientras que el segundo va con las capas restantes.

El orden en el que vamos a recibir las capas siempre es el mismo:

- low_X.mjc: El vídeo principal. Este vídeo se recibe al principio del bucle, es decir, cuando i vale 0. Para el resto de valores de i se gestionan el resto de archivos.
- frame_types_X: se compone de una o varias letras que indican si el vídeo que vamos a recibir contiene movimiento (I) o no (B).
- motion_residue_X.mjc: Contiene los vectores de movimiento.
- high_X.mjc: Los vídeos restantes. Según sea mayor el valor de la X peor será la calidad del vídeo.

Para escribir los datos recibidos en sus respectivos archivos vamos a usar el mismo método con el que creamos los archivos:

- low_X: Sus datos se guardan en archivo[0].
- frame_types_X: Sus datos se guardan en el archivo [1+3*(i-1)].
- motion_residue_X.mjc: Sus datos se guardan en el archivo [2+3*(i-1)].
- high_X.mjc: Sus datos se guardan en el archivo [3+3*(i-1)].

La razón por la cual al valor de i se le resta 1 es debido a que el bucle comienza con la recepción del vídeo ”Low_x.mjc” y este se recibe solo, el resto comienza con i valiendo 1. En la 2º vuelta del bucle for se posiciona en los vídeos 1, 2 y 3 respectivamente y así con el resto.

Cuando i vale 0 entonces tomamos el 1º tiempo en la variable ”tiempo_1”, llamamos a la función ”Guarda_Imagen” usando como parámetros de entrada el buffer donde se guarda los datos, el archivo[0] y el socket. Una vez que recibamos el vídeo volvemos a tomar el tiempo en la variable ”tiempo_2”, calculamos la diferencia de tiempo y la mostramos por pantalla.

Cuando i vale 1 o más entonces el procedimiento es el siguiente:

- Frame_Types: Tomamos el tiempo de inicio en ”tiempo_1”, llamamos a la función ”Guarda_Imagen” usando como parámetros de entrada el buffer, el archivo [1+3*(i-1)] y el socket. Cuando recibamos el archivo volvemos a tomar el tiempo en la variable ”tiempo_2” y mostramos por pantalla el nombre del archivo recibido, el GOP actual y la diferencia de tiempo.
- Motion_Residue: Tomamos el tiempo de inicio en ”tiempo_1”, llamamos a la función ”Guarda_Imagen” usando como parámetros de entrada el buffer, el archivo [2+3*(i-1)] y el socket. Cuando recibamos el archivo volvemos a tomar el tiempo en la variable ”tiempo_2” y mostramos por pantalla el nombre del archivo recibido, el GOP actual y la diferencia de tiempo.
- High: Tomamos el tiempo de inicio en ”tiempo_1”, llamamos a la función ”Guarda_Imagen” usando como parámetros de entrada el buffer, el archivo [3+3*(i-1)] y el socket. Cuando recibamos el archivo volvemos a tomar el tiempo en la variable ”tiempo_2” y mostramos por pantalla el nombre del archivo recibido, el GOP actual y la diferencia de tiempo.

Este bucle se repite hasta que se reciban todas las capas según nuestro ancho de banda. Tener en cuenta de que si el GOP 0 se transmitió bien el GOP 1 se recibe completo, en caso contrario solo se recibe una capa.

El código que realiza esta funcionalidad se muestra a continuación:

```
1   for(;;) {
2    /* En este bucle se gestiona la recepcion de los ↩
         videos y la escritura en disco */
3    gettimeofday(&comienzo, NULL);
4    for (i=0; i<contador_capa; i++){
5     if (i == 0){
6     /* Recibimos la capa perteneciente a Low_5. */
7     gettimeofday(&tiempo_1, NULL);
8     tamano_video = Guarda_Imagen(buffer, archivo[0], ↩
          sd);
9     gettimeofday(&tiempo_2, NULL);
10    diferencia = (tiempo_2.tv_sec+tiempo_2.tv_usec↩
          *0.000001)-(tiempo_1.tv_sec+tiempo_1.tv_usec↩
          *0.000001);
11    fprintf(stderr,"\nEl video Low %d del Gop %d ha ↩
          tardado en recibirse %lf\n", contador_maximo↩
          -1, j, diferencia);
12    fprintf(stderr,"  rl %d\n",contador_maximo);
13    fflush(stderr);
14   }else{
15    //Pedir los archivos de High. Peticiones se ↩
          convierte en procedimiento
16    gettimeofday(&tiempo_1, NULL);
17    tamano_video = Guarda_Imagen(buffer, archivo[1+3*(↩
          i-1)], sd);             //Recibe segmento ↩
          de frame_types
18    gettimeofday(&tiempo_2, NULL);
19    diferencia = (tiempo_2.tv_sec+tiempo_2.tv_usec↩
          *0.000001)-(tiempo_1.tv_sec+tiempo_1.tv_usec↩
          *0.000001);
20    fprintf(stderr,"\nEl video Frame_Types %d del Gop ↩
           %d ha tardado en recibirse %lf\n", ↩
          contador_maximo-i, j, diferencia);
21    fprintf(stderr,"  ft %d\n",contador_maximo-i-1);
22    fflush(stderr);
23
24    gettimeofday(&tiempo_1, NULL);
```

```
25     tamano_video = Guarda_Imagen(buffer, archivo[2+3*(i-1)], sd);             //Recibe segmento de motion_residues
26     gettimeofday(&tiempo_2, NULL);
27     diferencia = (tiempo_2.tv_sec+tiempo_2.tv_usec*0.000001)-(tiempo_1.tv_sec+tiempo_1.tv_usec*0.000001);
28     fprintf(stderr,"\nEl video Motion_Residue_ %d del Gop_ %d ha tardado en recibirse_ %lf\n", contador_maximo-i, j, diferencia);
29     fprintf(stderr,"  mr %d\n",contador_maximo-i-1);
30     fflush(stderr);
31 
32     gettimeofday(&tiempo_1, NULL);
33     tamano_video = Guarda_Imagen(buffer, archivo[3+3*(i-1)], sd);             //Recibe segmento de High
34     gettimeofday(&tiempo_2, NULL);
35     diferencia = (tiempo_2.tv_sec+tiempo_2.tv_usec*0.000001)-(tiempo_1.tv_sec+tiempo_1.tv_usec*0.000001);
36     fprintf(stderr,"\nEl video High_ %d del Gop_ %d ha tardado en recibirse_ %lf\n", contador_maximo-i, j, diferencia);
37     fprintf(stderr,"  rh %d\n",contador_maximo-i-1);
38     fflush(stderr);
39    }
40   }
```

Al comienzo del programa se enviarán todas las capas, pero según el ancho de banda las capas que vayamos recibiendo irán variando.

Una vez que hayamos recibido los vídeos tomamos el tiempo en la variable ”final” para comprobar el tiempo que ha tardado la recepción, ya que este tiempo es importante para saber si el ancho de banda es suficiente o no. También vamos a guardar dicho tiempo en la cadena ”cadena” para escribirlo en el archivo ”tiempo.txt” para tener un detalle guardado del tiempo que tardó en recibirse el GOP. El código que realiza esta funcionalidad se muestra a continuación:

```
1       gettimeofday(&final, NULL);
2       diferencia = (final.tv_sec+final.tv_usec*0.000001)-(comienzo.tv_sec+comienzo.tv_usec*0.000001);
3       fprintf(stderr, "\nEl tiempo total de recepcion ha sido de:_ %lf\n",diferencia);
4       fflush(stderr);
5       sprintf(cadena,"%lf %c",diferencia,salto);
6       fwrite(cadena, sizeof(char),9,tiempo );
```

El funcionamiento del segundo bucle más sencillo que el anterior ya que no hay que recibir datos del servidor sino que el cliente rellena los archivos que no hemos podido recibir con imágenes nulas. Este bucle recorre desde la capa posterior recibida hasta la última capa.

Este bucle posee una ideología similar al anterior ya que tiene en cuenta 2 casos:

- i = 0. Este caso se dará cuando no se pueda recibir ninguna capa, por lo que debemos insertar una imagen nula en el vídeo ”Low_X.mcj”. Limpiamos el buffer con la función ”Vacia_Cadena”, llamamos a la función ”Escribir_Nulo”, cuyos parámetros de entrada son el archivo[0], el buffer de la imagen nula, 1 sola imagen y el tamaño de la imagen nula.
- El resto. Debemos insertar las imágenes nulas en el resto de los archivos del siguiente modo:
  - Limpiamos el buffer.
  - Insertamos los carácteres en el vídeo ”Frame_Types_X” con la función ”Escribir_Frames”, cuyos parámetros de entrada son el archivo [1+3*(i-1)], el buffer y el número de carácteres a insertar, dado por $1 << (i-1)$.
  - Limpiamos el buffer.
  - Insertamos las imágenes nulas en el vídeo ”Motion_Residue_X” con la función ”Escribir_Nulo”, cuyos parámetros de entrada son el archivo [2+3*(i-1)], el buffer con la imagen nula del vector de movimiento, el número de imágenes a insertar dado por $1 << (i-1)$ y el tamaño de la imagen nula.
  - Insertamos las imágenes nulas en el vídeo ”High_X” con la función ”Escribir_Nulo”, cuyos parámetros de entrada son el archivo [3+3*(i-1)], el buffer con la imagen nula del vídeo, el número de imágenes a insertar dado por $1 << (i-1)$ y el tamaño de la imagen nula.

El código que realiza esta funcionalidad se muestra a continuación:

```
1   /* En el caso de que no se hayan enviado todas las capas, las capas restantes deben rellenarse con imagenes nulas */
2   for (i=contador_capa; i<contador_maximo; i++){
3    if (i == 0){
4      Vacia_Cadena(buffer);
5      Escribir_Nulo(archivo[0], buffer_nulo, 1, tamano_nulo);
6      fprintf(stderr,"\nSe ha impreso una imagen nula.");
7    }else{
8      Vacia_Cadena(buffer);
9      Escribir_Frames(archivo[1+3*(i-1)], buffer, 1<<(i-1));
10     Vacia_Cadena(buffer);
11     Escribir_Nulo(archivo[2+3*(i-1)], buffer_motion_nulo, 1<<(i-1), tamano_motion_nulo);
12     Escribir_Nulo(archivo[3+3*(i-1)], buffer_nulo, 1<<(i-1), tamano_nulo);
13     fprintf(stderr,"\nSe ha impreso una imagen nula.");
14   }
15  }
```

Como ya sabemos ambos valores de tiempo pasamos a calcular el número de capas que el cliente va a poder recibir en el siguiente envío.

Primero calculamos la diferencia de tiempo entre las variables ”final” y ”comienzo” y dicho resultado se divide entre 2 elevado al número de capas total del vídeo menos 1 partido 30 como podemos ver en el siguiente ejemplo:

$$diferencia = \frac{diferencia}{((1<<(contador_maximo-1))/30.000000)}$$

La razón por la que la diferencia de tiempo se divide de esta manera es que según el número de capas con el que haya sido comprimido el tiempo, excepto con el GOP 0 que contiene una sola imagen, con el resto en cada GOP se recibe un número de imágenes múltiplo de 2 como se puede observar en el siguiente ejemplo:

- 1 capa: $2^0/30 = 1/30 segundos$
- 2 capa: $2^1/30 = 2/30 segundos$
- 3 capa: $2^2/30 = 4/30 segundos$
- 4 capa: $2^3/30 = 8/30 segundos$
- 5 capa: $2^4/30 = 16/30 segundos$
- 6 capa: $2^5/30 = 32/30 segundos$

Por lo que se según el número de capas haya sido comprimido el tiempo cada GOP, excepto el 0, tiene que tardar en recibirse dicho tiempo, ya se reciban o no todas las capas.

Si la diferencia es menor de 1 segundo entonces el cliente tiene suficiente ancho de banda como para enviar más capas o todas ellas. Esto se calcula haciendo la comprobación entre el número máximo de capas existentes y el número de capas que hemos recibido en ese momento.

- Si los valores son iguales es porque estamos recibiendo todas las capas y no es necesario hacer ningún cambio.
- En el caso de que sean distintos es porque no estamos recibiendo todas las capas, por lo que en el siguiente envío vamos a poder recibir una capa más.

Al finalizar se incrementa en 1 el valor de j.

En el caso de que la diferencia fuese mayor de un segundo se pueden dar 2 casos:

- Si hemos 2 o más capas entonces debemos decrementar el número de capas a 1 e incrementamos el valor de j a 1.
- Si solo hemos recibido 1 capa es porque hemos recibido solo la capa 1 y no hemos tenido suficiente ancho de banda, por lo que se va a realizar el mismo procedimiento que con el GOP 0. Realizamos el redondeo a lo bajo de la diferencia mediante la función ”floor” y lo guardamos en la variable ”repetir”, pasamos el valor de contador_capa a 0 y el siguiente paso es el mismo que con el GOP 0 cuando no tuvimos suficiente ancho de banda. Cada vez que se inserta imágenes nulas en un GOP completo se incrementa en 1 el valor de j. Aquí la única diferencia respecto al principio es que cuando incrustamos las imágenes nulas en un GOP se puede dar el caso de que el servidor haya llegado al final de los vídeos, por lo que realizamos una recepción mediante un recv donde si dicho valor es 1 es debido a que el servidor a llegado al final de los archivos y entonces se incrementaría a 1 el valor de ruptura y saldríamos del bucle. En caso contrario se repite de nuevo el bucle y se decrementa el valor de repetir.

Para finalizar mostramos por pantalla el número de capas que el cliente va a poder recibir en el siguiente envío. El código que realiza esta funcionalidad se muestra a continuación:

```
1   //En el caso de que no haya terminado la transmision calcular el numero de capas para el siguiente envio
2    diferencia = diferencia/((1<<(contador_maximo-1))/30.000000);
3    if(diferencia <= 1){ //Tenemos suficiente ancho de banda
4    if (contador_aux == contador_maximo) {
5    contador_capa = contador_maximo;
6
7    }else{
8           contador_aux++;
9           contador_capa = contador_aux;
10   }
11   j++;   //Se incrementa el valor del Gop
12   }else{ //No tenemos suficiente ancho de banda. Aqui se dan 2 casos
13    if (contador_capa >=2){ //Hemos recibido mas de 2 capas, por lo que decrementamos una capa.
14      contador_capa--;
15      j++;         //Se incrementa el valor del Gop
16    }else{ //No tenemos suficiente ancho de banda con una capa, por lo que tenemos que calcular cuantas imagenes nulas tenemos que anadir
17      repetir = floor(diferencia); //Se redondea a lo bajo para obtener el numero de repeticiones
18      contador_capa = 0;
19      while (repetir >0){
20        send (sd, (void *)&contador_capa, sizeof(contador_capa), 0);
21        for (i=0; i<contador_maximo; i++){
22        if (i == 0){
23         Vacia_Cadena(buffer);
24         Escribir_Nulo(archivo[0], buffer_nulo, 1, tamano_nulo);
25         fprintf(stderr,"\nSe ha impreso una imagen nula.");
26        }else{
27         Vacia_Cadena(buffer);
28         Escribir_Frames(archivo[1+3*(i-1)], buffer, 1<<(i-1));
29         Vacia_Cadena(buffer);
30         Escribir_Nulo(archivo[2+3*(i-1)], buffer_motion_nulo, 1<<(i-1), tamano_motion_nulo);
31         Escribir_Nulo(archivo[3+3*(i-1)], buffer_nulo, 1<<(i-1), tamano_nulo);
32         fprintf(stderr,"\nSe ha impreso una imagen nula.");
33         }
34        }
35        j++;
36        //Aqui se recibe tambien el valor de salir por 2 razones: por el caso de que el servidor nos haya enviado todo y para que este no pierda el control
37        repetir--;
38        recv (sd, (void *)&salir, sizeof(salir), 0);
39        if (salir == 1){
40         ruptura = 1; //El servidor ha llegado al final del archivo, por lo que incrementamos el valor de la variable a 1 para finalizar
41         break;
42        }
43      }
44      contador_capa++;
45    }
46   }
47  fprintf(stderr, "\nEn el siguiente envio se reciben %d capas\n", contador_capa);
```

Una vez que el cliente sabe el número de capas que puede recibir después envía ese dato al servidor mediante la función ”send” usando como datos de entrada:

- El valor del socket.
- En número de capas que vamos a poder recibir.
- El tamaño de la variable.
- La bandera, por defecto 0.

Por último se va a comprobar si el servidor ha llegado al final de los archivos. En el caso de que en las incrustaciones de las imágenes nulas en los GOPs no llegara al final de los vídeos entonces recibiríamos del servidor un valor mediante recv. Si dicho valor es 1 o en el caso de que sí terminara durante las imágenes nulas entonces saldremos del bucle general para finalizar el programa. El código que realiza esta funcionalidad se muestra a continuación:

```
1   //Una vez calculado las capas que podemos recibir enviamos el valor al servidor
2   //contador_capa = 5;
3   send(sd, (void *) &contador_capa, sizeof(contador_capa), 0);
4
5   //Comprobamos que el servidor haya terminado de enviar todos los videos
6   if (ruptura == 0) recv (sd, (void *)&salir, sizeof(salir), 0);
7   if ((salir==1)||(ruptura==1)) break; //El programa finaliza cuando el servidor llegue al final del archivo, ya sea recibiendo datos o insertando imagenes nulas
8  }
```

Para finalizar el proceso del cliente vamos a guardar el tiempo de finalización en la variable ”fin” para realizar la

diferencia y mostrar el tiempo total que ha transcurrido en todo el proceso y el total de GOPs que se han enviado (hay que restarle un 1 al valor de j) y luego pasamos a cerrar todos los archivos que se han abierto. En el caso del primer archivo se hace normal, pero para los archivos de las capas es necesario usar un bucle for, como podemos comprobar en el código a continuación:

```
1    gettimeofday(&fin, NULL);
2    diferencia = (fin.tv_sec+fin.tv_usec*0.000001)-(
         ini.tv_sec+ini.tv_usec*0.000001);
3    printf("\nEl tiempo total de recepcion es de %lf\n
         ", diferencia);
4    printf("\nEl total de Gop es de %d\n", j-1);
5    sprintf(cadena,"%lf%c",diferencia,salto);
6    fwrite(cadena, sizeof(char),9,tiempo );
7
8    //Cerramos los archivos
9    fclose(archivo[0]);
10   for (i=0; i<contador_maximo-1; i++){
11     fclose(archivo[1+3*i]);
12     fclose(archivo[2+3*i]);
13     fclose(archivo[3+3*i]);
14   }
15   printf("\n");
16   fclose(tiempo);
17 }
```

También cerramos el archivo donde se guardaban los tiempos.

VI. Evaluación

VI-A. Resultados de recepción con 6 capas

Vamos a comentar las pruebas que se han realizado para comprobar los resultados obtenidos con la recepción de los vídeos comprimidos con MCJ2K. El vídeo que se ha usado se llama ”coastguard.yuv”, el cual posee un tamaño de imagen de 352x288 y el tamaño total del vídeo es de 45619200 bytes. Tenemos que calcular el número de imágenes total que posee el vídeo que vamos a descomprimir, ya que vamos a coger un número de imágenes que sea múltiplo de 32 más 1. Esta es una regla establecida por el compresor. Para calcular el total de imágenes que posee un vídeo usaremos la siguiente fórmula:

$$Total_Imagenes = \frac{Tamano_v'ideo}{Tamano_imagen * 1.5} \quad (8)$$

Como ya sabemos el tamaño de la imagen y del vídeo obtenemos el total de imágenes:

$$Total_Imagenes = \frac{45619200}{352 * 288 * 1.5} = 300 \quad (9)$$

El vídeo ”coastguard.yuv” se compone de un total de 300 imágenes. Ahora vamos a coger un número de imágenes que sea múltiplo de 32 más 1. En este caso vamos a coger 289 imágenes ya que:

$$289 = 288 + 1 = 2*2*2*2*2*3*3+1 = 32*9+1 \quad (10)$$

Una vez que sepamos el número de imágenes con el cual el vídeo ha sido comprimido podemos pasar a descomprimirlo.

El vídeo ha sido recibido en diferentes capas de compresión y a diferentes velocidades de recepción. Las velocidades de recepción usadas son las siguientes:

- 100 kBs = 12.5 kbs
- 200 kBs = 25 kbs
- 300 kBs = 37.5 kbs
- 400 kBs = 50 kbs
- 500 kBs = 62.5 kbs
- 600 kBs = 75 kbs

Primero vamos a explicar las recepciones de los vídeos con 6 capas. Para ello vamos a obtener una tabla que contiene la información de la cantidad de GOPs de los que se compone el vídeo comprimido y el ancho de banda necesario para la recepción. Para obtener dicha tabla usamos el siguiente comando en la carpeta donde se encuentre el vídeo comprimido con 6 capas:

```
mcj2k info --pictures=289
```

La tabla que obtenemos se muestra en la Tabla I.

En la tabla se muestra el total de GOPs de los que se compone el vídeo y para cada archivo la cantidad de kbs necesarios para recibir dichos archivos.
En el GOP 0 se observa que solo se va a recibir una imagen de low_5.mjc. Esta es la primera imagen que se recibe y se hace sola y no influye el ancho de banda que tengamos ya que se tiene que recibir siempre. A partir del GOP 1 se empiezan a recibir todos los vídeos, en el caso de tener suficiente ancho de banda. Vemos que para cada capa se muestra el ancho de banda necesario para recibir cada archivo, a la derecha del todo se muestra el ancho de banda total necesario y abajo del todo se muestra la media.

Una vez tengamos los vídeos recibidos con las diferentes velocidades vamos a pasar a descomprimirlos, para ello tenemos que usar el siguiente comando:

```
mcj2k expand --pictures=289
```

Cuando estén los 6 vídeos descomprimidos se van a comparar con el vídeo original para comprobar las diferencias existentes en cada vídeo, por lo que vamos a usar el siguiente comando:

```
snr --file_A="ruta_v'ideo_original \
  --file_B="ruta_v'ideo_recibido" \
  2> "velocidad_recepci'on
```

Este comando lo que hace es generar comprobar las diferencias entre 2 vídeos y las guardamos en un archivo para luego generar una gráfica con dichos resultados. Una vez que se tengan los datos de los 6 vídeos pasamos a generar una gráfica uniendo todos los vídeos para comprobar las diferencias entre si. Para generar la gráfica nos ubicamos en la ruta donde se encuentren los archivos y escribimos en la terminal ”gnuplot” más el archivo ”crear_imagen”, que contiene los siguientes comandos:

```
1 set terminal fig textspecial color
2
3 set output "grafica_capa_6.fig"
4
5 plot "./100k-6" with lines title "100k-6", "./300k-6"
       with lines title "300k-6", "./600k-6" with lines
       title "600k-6"
```

Con este comando se genera la siguiente gráfica para 6 capas:

```
/home/cesar/MCTF/bin/info_j2k: temporal_levels=6
/home/cesar/MCTF/bin/info_j2k: pictures=289
/home/cesar/MCTF/bin/info_j2k: pictures_per_second=30
/home/cesar/MCTF/bin/info_j2k: GOP_size=32
/home/cesar/MCTF/bin/info_j2k: number_of_GOPs=9
/home/cesar/MCTF/bin/info_j2k: GOP_time=1.06666666667
/home/cesar/MCTF/bin/info_j2k: All the values are given in thousands of bits per second (Kbps).
```

GOP#	TRL5 low_5		motion_5	TRL4 high_5		motion_4	TRL3 high_4		motion_3	TRL2 high_3		motion_2	TRL1 high_2		motion_1	TRL0 high_1	Total
0000	52	-	0	0	--	0	0	----	0	0	--------	0	0	----------------	0	0	52
0001	59	B	6	55	BB	10	95	BBBB	15	125	BBBBBBBB	20	98	BBBBBBBBBBBBBBBB	27	53	567
0002	59	B	6	64	BB	11	113	BBBB	17	157	BBBBBBBB	22	153	BBBBBBBBBBBBBBBB	32	112	752
0003	62	B	7	71	BB	12	88	BBBB	19	149	BBBBBBBB	31	236	BBBBBBBBBBBBBBBB	45	250	976
0004	57	B	6	57	BB	11	90	BBBB	17	132	BBBBBBBB	24	151	BBBBBBBBBBBBBBBB	33	132	716
0005	57	B	6	58	BB	10	82	BBBB	15	109	BBBBBBBB	21	138	BBBBBBBBBBBBBBBB	30	96	626
0006	53	B	5	56	BB	11	90	BBBB	17	122	BBBBBBBB	22	127	BBBBBBBBBBBBBBBB	29	91	629
0007	54	B	6	53	BB	10	86	BBBB	16	104	BBBBBBBB	20	117	BBBBBBBBBBBBBBBB	27	93	591
0008	52	B	5	51	BB	10	81	BBBB	15	104	BBBBBBBB	20	124	BBBBBBBBBBBBBBBB	26	97	592
0009	48	B	5	50	BB	10	81	BBBB	15	98	BBBBBBBB	19	114	BBBBBBBBBBBBBBBB	29	109	584
Average	56		6	57		11	90		16	122		22	140		31	115	670

Total average: 670.72 Kbps (notice that the first GOP is not included)

Cuadro I
COMPOSICIÓN DEL CODE-STREAM.

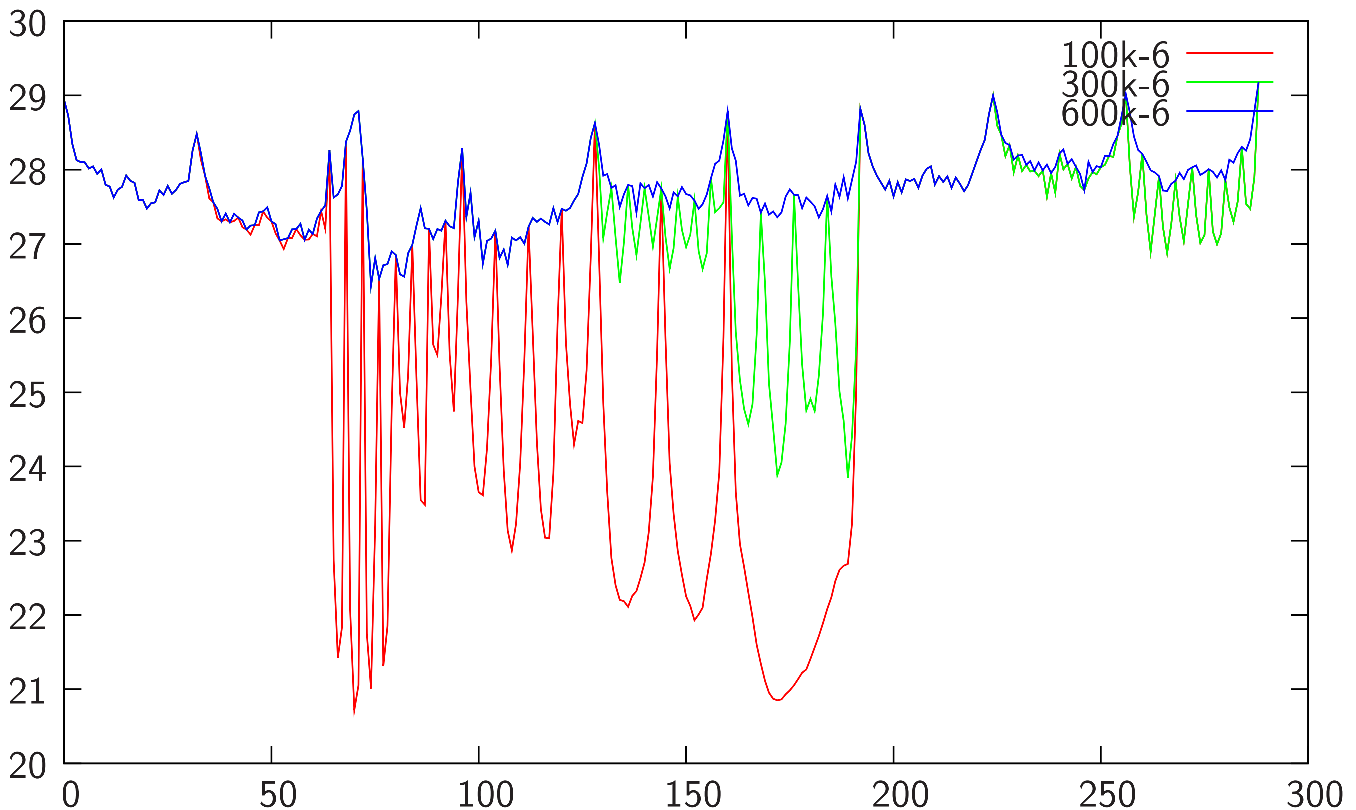

Figura 28. Recepción del vídeo con 6 capas.

En dicha gráfica podemos observar que con un ancho de banda de 600 KBs podemos recibir el vídeo casi por completo, ya que cuando se reciben todas las capas el cliente no tiene suficiente ancho de banda para recibir todas las capas, por lo que deja de recibir la capa 6 y en el siguiente GOP como posee ancho de banda de sobra para recibir 5 capas vuelve a pedir al servidor que le envíe todas las capas, pero le vuelve a suceder lo mismo, así hasta recibir todas. Esto se observa en los picos de la gráfica de 600k. Con un ancho de banda de 300 KBs obtenemos los mismos resultado que con 600k, pero en la mitad de la recepción el cliente empieza a pedir que el servidor le envíe menos capas, como se observa. Más adelante ve que vuelve a tener suficiente ancho de banda para pedir más capas, pero de nuevo no tiene ancho de banda para recibir esas capas por lo que le pide al servidor que le envíe menos capas de nuevo. Con 100 KBs obtenemos peores resultados, ya que solo es capaz de recibir de 1 a 2 capas, por lo que, como podemos observar en la gráfica, comienza a pedir menos capas. Cuando solo está recibiendo una sola capa tiene ancho de banda de sobra para pedir una más, y así estará pidiendo entre 1 a 2 capas.

Lo que podemos detallar en esta gráfica es que cuanto mayor sea el ancho de banda, mas se van a poder transmitir todas las capas y no se tengan que incrustar imágenes nulas, mostrando

una calidad similar al del vídeo original. Ahora vamos a seguir comprobando los resultados con las pruebas restantes.

VI-B. Resultados de recepción con 5 capas

Si nos vamos a la tabla del "Apéndice C" podemos ver la información del vídeo con 5 capas. Para obtener esta tabla nos vamos a la carpeta donde se encuentre el vídeo y usamos el siguiente comando:

```
mcj2k info --temporal\_levels=5
--pictures=289
```

Un detalle importante que se ve ahora en el comando es el uso de "–temporal_levels". Este comando sirve para indicarle al programa de compresión el número de capas del que se compone el vídeo. Por defecto su valor es 6 (el número de capas de los que se compone el vídeo, por defecto 5, más 1), por lo que para obtener la información con 5 capas se pone 5. Lo que destaca ahora en esta tabla es lo siguiente:

- El vídeo posee el doble de GOPs respecto al caso anterior (9 GOPs).
- El ancho de banda, tanto del vídeo "low_4.mcj" como de la capa 4 (high_4.mjc"), a incrementado, teniendo una media de ancho de banda de 112 y 90 Kbs respectivamente.

La Figura 29 contiene los resultados de la comparación del vídeo original con los vídeos recibidos a las distintas velocidades.
En este caso al principio se observa que se tienen los mismos resultados que con 6 capas, pero mas adelante, se supone que en el GOP 2, no hay ancho de banda disponible, por lo que se insertan imágenes nulas. Con 600 KBs se pueden enviar casi todas las capas, aunque al principio podemos observar de que es incapaz de servir todas y pide una capa menos, y en la siguiente recepción le vuelve a pedir todas las capas. Con 300 KBs se observa que hay momentos donde puede recibir todas las capas, pero en el siguiente envío le pedirá una capa menos. Aquí ya se observa que con 100k no tenemos suficiente ancho de banda para recibir todas las capas, por lo que cuando recibe una imagen de la capa 1 (Low_X) vuelve a insertar en todas las capas imágenes nulas. También la necesidad de un mayor ancho de banda repercute en el resto de resultados, llegando momentos donde no se recibe una capa pero después el cliente le vuelve a pedir dicha capa.

VI-C. Resultados de recepción con 4 capas

La Figura 30 contiene los resultados de la comparación del vídeo original con los vídeos recibidos a las distintas velocidades. Aquí se puede observar que con 100 KBs se obtienen pulsos hacia abajo, esto nos quiere decir que se envían en uno o varios GOPs varias imágenes nulas en todos los vídeos y con 200 KBs también empezamos a obtener peores resultados. Con 300 KBs y 600 KBs obtenemos unos resultados similares que en el caso anterior, aún así se empieza a notar la insuficiencia de ancho de banda.

VI-D. Resultados de recepción con 3 capas

La Figura 31 contiene los resultados de la comparación del vídeo original con los vídeos recibidos a las distintas velocidades. Ahora con 100 KBs las veces que se insertar imágenes nulas en todos los archivos es mayor. Con 300 KBs comenzamos a ver que es incapaz de recibir todas las capas, por lo que en un envío recibirá una imagen y en el siguiente incrustará imágenes nulas en todos los árchivos. Con 600 KBs comenzamos a ver la insuficiencia de ancho de banda para recibir todas las capas. Al comienzo se observa que pasará de recibir 3 capas a ir bajando hasta recibir 1 capa y más adelante se irá recuperando, donde pide de 2 a 3 capas y a mitad de la recepción tiene ancho de banda para recibir todas las capas.

VI-E. Resultados de recepción con 2 capas

La Figura 32 contiene los resultados de la comparación del vídeo original con los vídeos recibidos a las distintas velocidades. Con 2 capas se observa que con ninguna de las pruebas no tenemos ancho de banda para más de una capa, por lo que estarán insertando en varios GOPs imágenes nulas en todos los vídeos, donde con 100 KBs y 300 KBs se incrustarán seguidamente imágenes nulas en todos los archivos y con 600 KBs estará recibiendo una imagen y en la siguiente recepción no pide nada.

VI-F. Resultados de recepción con 1 capas

La Figura 33 contiene los resultados de la comparación del vídeo original con los vídeos recibidos a las distintas velocidades. Aquí ya se puede observar que deberíamos tener un ancho de banda bastante alto para poder recibir el vídeo.

VI-G. Comparación de resultados con 100 KBs en 1 capa, 3 capas y 6 capas.

En la Figura 34 podemos ver los resultados de la recepción de los vídeos comprimidos con 1 capa, 3 capas y 6 capas con un ancho de banda de 100 KBs. Aquí podemos ver mejor estos resultados donde con 1 capas, como se explicó anteriormente, es incapaz de enviar el vídeo estando insertando durante varios GOPs imágenes nulas. Con 3 capas al principio tenemos ancho de banda disponible para recibir todas las capas, pero al poco tiempo volvemos a quedarnos sin ancho de banda repitiendo el caso anterior. Con 6 capas obtenemos unos mejores resultados, aunque en mitad de la transmisión comience a pedir menos capas comienza a pedirle más capas, obteniendo unos resultados casi similares al vídeo original.

VI-H. Comparación de resultados con 600k en 1 capa, 3 capas y 6 capas.

En la Figura 35 podemos ver los resultados de la recepción de los vídeos comprimidos con 1 capa, 3 capas y 6 capas con un ancho de banda de 600 KBs. Con este ancho de banda los resultados obtenidos son distintos al caso anterior. En el caso de 1 capa, aunque tampoco tengamos ancho de banda para recibir la capa, las veces que se inserta una imagen nula en el vídeo es menor respecto al usar un ancho de banda de 100KBs.

En el caso de 3 capas hay momentos donde el cliente le pide al servidor una capa menos pero después vuelve a pedírsela. Al final con 6 capas se observa que no posee suficiente ancho de banda para recibir todas las capas, pero si que puede recibir 5 capas y así le sobra ancho de banda como para pedirle de nuevo que le envíe todas las capas, y cuando recibe todas las capas vuelve a tener una carencia de ancho de banda y pedirle de nuevo una menos.

VII. Conclusiones y trabajo futuro

Se ha conseguido los objetivos planteados inicialmente en este proyecto, que eran la implementación de un cliente de vídeo escalable que adaptase la tasa de transmisión de datos al ancho de banda de recepción del cliente. El programa puede mejorarse incluyendo a la hora de ejecutarse parámetros como el puerto por el que escucha, el tamaño del paquete (payload) y la implementación de la reproducción del vídeo a la misma vez que estamos recibiendo los archivos. Otros posible trabajos es la creación de un cliente de vídeo escalable que adaptase la calidad de vídeo.

Apéndice A
Generación de una imagen nula

Por un requerimiento del software de descompresión, todos los niveles de resolución deben ser reconstruidos aunque no sean recibidos. Cuando el cliente no recibe todas las imágenes de un nivel de resolución temporal o ni tan siquiera recibe un nivel completo, genera en disco tantas imágenes en el dominio JPEG 2000 como imágenes ausentes existen. Dichas imágenes son siempre imágenes nulas (iguales al tono de color RGB (128,128,128) OJO CON ESTO, PUEDE SER QUE SEA (0,0,0)) con el objeto de que no generen perturbaciones visualmente desagradables en las reconstrucciones.

Para genera una imagen nula se realizan los siguientes pasos:

1. Crear una secuencia YUV 4:2:2 con una imagen nula (en realidad una imagen TYV 4:2:0 con todos los pixels a [0,0,0]). Sea dicha secuencia `nula.yuv`.
   ```
   dd if=/dev/zero of=nula.yuv \
   count=$[352*288+(352*288)/2]
   ```
2. Sumarle 128 a cada componente de dicha imagen:
   ```
   add char 128 < nula.yuv > 128.yuv
   ```
   Donde el programa `add.c` simplemente suma el valor 128 a cada entero del fichero de entrada y genera dicho resultado en el dichero de salida.
3. Comprimirla:
   ```
   rm -f low_0
   ln -s 128.yuv low_0
   mcj2k compress --temporal_levels=1 \
    --pictures=1
   ```
   Sea dicha secuencia comprimida `1.mjc`:
   ```
   mv low_0.mjc 1.mjc
   ```
   y
   $$\texttt{tam1} = \#\texttt{1.mjc} \tag{11}$$
   su longitud.
4. Añadir una imagen nula a la secuencia anterior:
   ```
   cp 128.yuv 1
   cat 1 >> 128.yuv
   rm 1
   ```
5. Comprimirla:
   ```
   rm -f low_0
   ln -s 128.yuv low_0
   mcj2k compress --temporal_levels=1 \
    --pictures=2
   ```
   Sea dicha secuencia comprimida `2.mjc`:
   ```
   mv low_0.mjc 2.mjc
   ```
   y
   $$\texttt{tam2} = \#\texttt{2.mjc} \tag{12}$$
   su longitud.
6. Calcular la diferencia entre las longitudes de `2.mjc` y `1.mjc`. Es decir, calcular
   $$\texttt{dif} = \texttt{tam2} - \texttt{tam1}. \tag{13}$$
7. Extraer de `2.mjc` la última imagen del code-stream. Dicha imagen va desde el byte que está en la posición `tam1` hasta el final del archivo. Esto se puede hacer con el comando:
   ```
   dd if=2.mjc of=128-sin-cabecera.mjc \
     bs=1 count=dif skip=tam1
   ```

Cada vez que el descompresor encuentre la secuencia de bytes contenida en `nula-sin-cabecera.mjc` tras una cabecera válida[4], generará a la salida una imagen nula en el formato YUV 4:2:2.

Apéndice B
Generación de campo de movimiento nulo

De nuevo, por un mero requerimiento del software de descompresión, todos los campos de movimiento deben ser reconstruidos aunque no sean recibidos. Cuando esto ocurra, se supondrá que los campos de movimiento no recibidos son nulos (lo que implica un modelo de movimiento lineal).

Para generar un capo de movimiento nulo se realizan los siguientes pasos:

1. Crear una secuencia YUV 4:2:2 con tres imágenes nulas. Sea dicha secuencia `3nula.yuv`.
   ```
   dd if=/dev/zero of=nula.yuv \
     count=$[352*288+(352*288)/2]
   cp nula.yuv 3nula.yuv
   cat nula.yuv >> 3nula.yuv
   cat nula.yuv >> 3nula.yuv
   ```
2. Comprimirla, usando dos niveles de resolución temporal:
   ```
   rm -f low_0
   ln -s 3nula.yuv low_0
   mcj2k compress --temporal_levels=2 \
     --pictures=3
   ```
   Sea el único campo de movimiento comprimido `3motion.mjc`:

[4]La cabecera de todas las secuencias `.mjc` de todos los niveles de resolución temporal es idéntica puesto que los parámetros de compresión se conservan entre subbandas temporales.

```
mv motion_residue_1.mjc 3motion.mjc
```

y

$$\texttt{tam3} = \#\texttt{3motion.mjc} \tag{14}$$

su longitud.

3. Añadir 2 imágenes nulas a la secuencia anterior:

```
cp 3nula.yuv 5nula,yuv
cat nula.yuv >> 5nula.yuv
cat nula.yuv >> 5nula.yuv
```

4. Comprimirla:

```
rm -f low_0
ln -s 5nula.yuv low_0
mcj2k compress --temporal_levels=2 \
  --pictures=5
```

Sea dicha secuencia comprimida `5motion.mjc`:

```
mv motion_residue_1.mjc 5motion.mjc
```

y

$$\texttt{tam5} = \#\texttt{5motion.mjc} \tag{15}$$

su longitud.

5. Calcular la diferencia entre las longitudes de `3motion.mjc` y `5motion.mjc`. Es decir, calcular

$$\texttt{dif} = \texttt{tam5} - \texttt{tam3}. \tag{16}$$

6. Extraer de `5motion.mjc` el último campo del codestream. Dicha campo va desde el byte que está en la posición `tam3` hasta el final de `5motion.mjc`. Esto se puede hacer con el comando:

```
dd if=5motion.mjc \
  of=motion\_nulo.mjc bs=1 \
  count=dif skip=tam3
```

Cada vez que el descompresor encuentre la secuencia de bytes contenida en `nula-sin-cabecera.mjc` tras una cabecera válida[5], generará a la salida una imagen nula en el formato YUV 4:2:2.

APÉNDICE C
RESULTADOS DE LA RECEPCIÓN DE UN VÍDEO COMPRIMIDO CON 5 CAPAS

Esta es la tabla de la información del vídeo comprimido con 5 capas:

APÉNDICE D
RESULTADOS DE LA RECEPCIÓN DE UN VÍDEO COMPRIMIDO CON 4 CAPAS

Esta es la tabla de la información del vídeo comprimido con 4 capas:

```
/home/cesar/MCTF/bin/info_j2k: temporal_levels=4
/home/cesar/MCTF/bin/info_j2k: pictures=289
/home/cesar/MCTF/bin/info_j2k: pictures_per_second=30
/home/cesar/MCTF/bin/info_j2k: GOP_size=8
/home/cesar/MCTF/bin/info_j2k: number_of_GOPs=36
/home/cesar/MCTF/bin/info_j2k: GOP_time=0.266666666667
/home/cesar/MCTF/bin/info_j2k: All the values are given in thousands of bits per ↩
      second (Kbps).
```

	TRL3			TRL2			TRL1			TRL0	
GOP#	low_3		motion_3	high_3		motion_2	high_2		motion_1	high_1	Total
0000	209	-	0	0	--	0	0	----	0	0	209
0001	210	B	16	122	BB	20	87	BBBB	25	46	530
0002	217	B	15	115	BB	19	92	BBBB	25	48	534
0003	220	B	16	135	BB	19	105	BBBB	30	54	583
0004	236	B	17	126	BB	20	106	BBBB	28	63	599
0005	233	B	16	140	BB	20	132	BBBB	30	72	646
0006	230	B	16	154	BB	21	138	BBBB	30	103	695
0007	238	B	17	159	BB	21	151	BBBB	30	94	714
0008	239	B	18	176	BB	27	192	BBBB	38	178	871
0009	135	B	23	155	BB	42	245	BBBB	63	397	1062
0010	250	B	19	162	BB	31	256	BBBB	48	244	1013
0011	263	B	15	145	BB	27	228	BBBB	36	193	910
0012	250	B	14	132	BB	24	217	BBBB	35	168	843
0013	238	B	15	140	BB	21	137	BBBB	30	124	709
0014	231	B	17	138	BB	24	180	BBBB	33	114	740
0015	234	B	18	129	BB	28	158	BBBB	36	177	783
0016	231	B	15	123	BB	23	130	BBBB	31	113	670
0017	227	B	14	116	BB	23	157	BBBB	32	108	680
0018	225	B	15	112	BB	22	156	BBBB	28	80	641
0019	223	B	14	101	BB	21	114	BBBB	30	88	594
0020	230	B	15	106	BB	20	124	BBBB	30	107	635
0021	227	B	16	120	BB	22	120	BBBB	28	90	626
0022	226	B	17	132	BB	26	141	BBBB	29	89	663
0023	224	B	15	111	BB	21	119	BBBB	30	98	621
0024	214	B	15	125	BB	20	130	BBBB	30	88	624
0025	220	B	15	104	BB	22	115	BBBB	30	91	599
0026	210	B	14	101	BB	20	123	BBBB	26	85	581
0027	222	B	15	107	BB	20	120	BBBB	26	95	607
0028	217	B	15	105	BB	19	110	BBBB	26	101	595
0029	210	B	14	104	BB	20	113	BBBB	26	99	591
0030	205	B	16	106	BB	21	134	BBBB	27	98	610
0031	207	B	16	105	BB	20	133	BBBB	26	94	604
0032	211	B	14	101	BB	19	117	BBBB	26	97	587
0033	207	B	14	98	BB	19	110	BBBB	29	114	593
0034	208	B	14	99	BB	18	116	BBBB	31	119	608
0035	207	B	14	97	BB	20	114	BBBB	27	97	580
0036	194	B	14	96	BB	19	114	BBBB	28	107	575
Average	221		16	122		22	140		31	115	670

```
Total average: 670.343333333 Kbps (notice that the first GOP is not included)
```

APÉNDICE E
RESULTADOS DE LA RECEPCIÓN DE UN VÍDEO COMPRIMIDO CON 3 CAPAS

Esta es la tabla de la información del vídeo comprimido con 3 capas:

```
/home/cesar/MCTF/bin/info_j2k: temporal_levels=3
/home/cesar/MCTF/bin/info_j2k: pictures=289
/home/cesar/MCTF/bin/info_j2k: pictures_per_second=30
/home/cesar/MCTF/bin/info_j2k: GOP_size=4
/home/cesar/MCTF/bin/info_j2k: number_of_GOPs=72
/home/cesar/MCTF/bin/info_j2k: GOP_time=0.133333333333
/home/cesar/MCTF/bin/info_j2k: All the values are given in ↩
      thousands of bits per second (Kbps).
```

	TRL2			TRL1			TRL0	
GOP#	low_2		motion_2	high_2		motion_1	high_1	Total
0000	418	-	0	0	--	0	0	418
0001	391	B	19	93	BB	25	45	576
0002	420	B	19	82	BB	25	47	595
0003	440	B	18	96	BB	25	46	628
0004	434	B	18	87	BB	25	49	616
0005	454	B	20	100	BB	29	51	656
0006	441	B	20	111	BB	30	57	661
0007	458	B	19	98	BB	28	60	665
0008	473	B	19	115	BB	27	66	702
0009	467	B	22	127	BB	30	75	722
0010	466	B	20	137	BB	30	68	724
0011	461	B	21	120	BB	30	82	716
0012	460	B	24	156	BB	31	124	796
0013	474	B	24	146	BB	29	98	773
0014	477	B	24	156	BB	31	91	781
0015	469	B	26	177	BB	33	96	803
0016	479	B	29	207	BB	42	260	1020
0017	318	B	28	191	BB	40	254	832
0018	270	B	49	298	BB	86	540	1244
0019	510	B	39	313	BB	60	325	1250
0020	500	B	25	200	BB	36	162	924
0021	484	B	28	272	BB	38	157	981
0022	526	B	29	183	BB	34	230	1005
0023	510	B	21	174	BB	36	197	939
0024	501	B	28	261	BB	33	138	964
0025	502	B	20	122	BB	32	152	830
0026	477	B	25	152	BB	28	96	781
0027	466	B	27	215	BB	33	110	853
0028	462	B	24	145	BB	33	118	785
0029	448	B	30	166	BB	33	116	795
0030	468	B	23	150	BB	39	239	920
0031	464	B	25	160	BB	33	139	823
0032	462	B	21	101	BB	30	88	702
0033	458	B	20	121	BB	29	88	717
0034	455	B	28	193	BB	35	128	841
0035	447	B	29	186	BB	29	83	777
0036	450	B	21	126	BB	27	77	702
0037	440	B	23	129	BB	30	88	711
0038	446	B	20	100	BB	30	89	686
0039	443	B	21	127	BB	32	125	750
0040	461	B	21	121	BB	27	89	721
0041	446	B	20	99	BB	28	86	681
0042	455	B	26	140	BB	28	94	746
0043	449	B	27	163	BB	32	93	767
0044	453	B	25	118	BB	27	85	709
0045	440	B	22	114	BB	30	91	699
0046	448	B	24	125	BB	30	104	733

[5]La cabecera de todas las secuencias `.mjc` de todos los niveles de resolución temporal es idéntica puesto que los parámetros de compresión se conservan entre subbandas temporales.

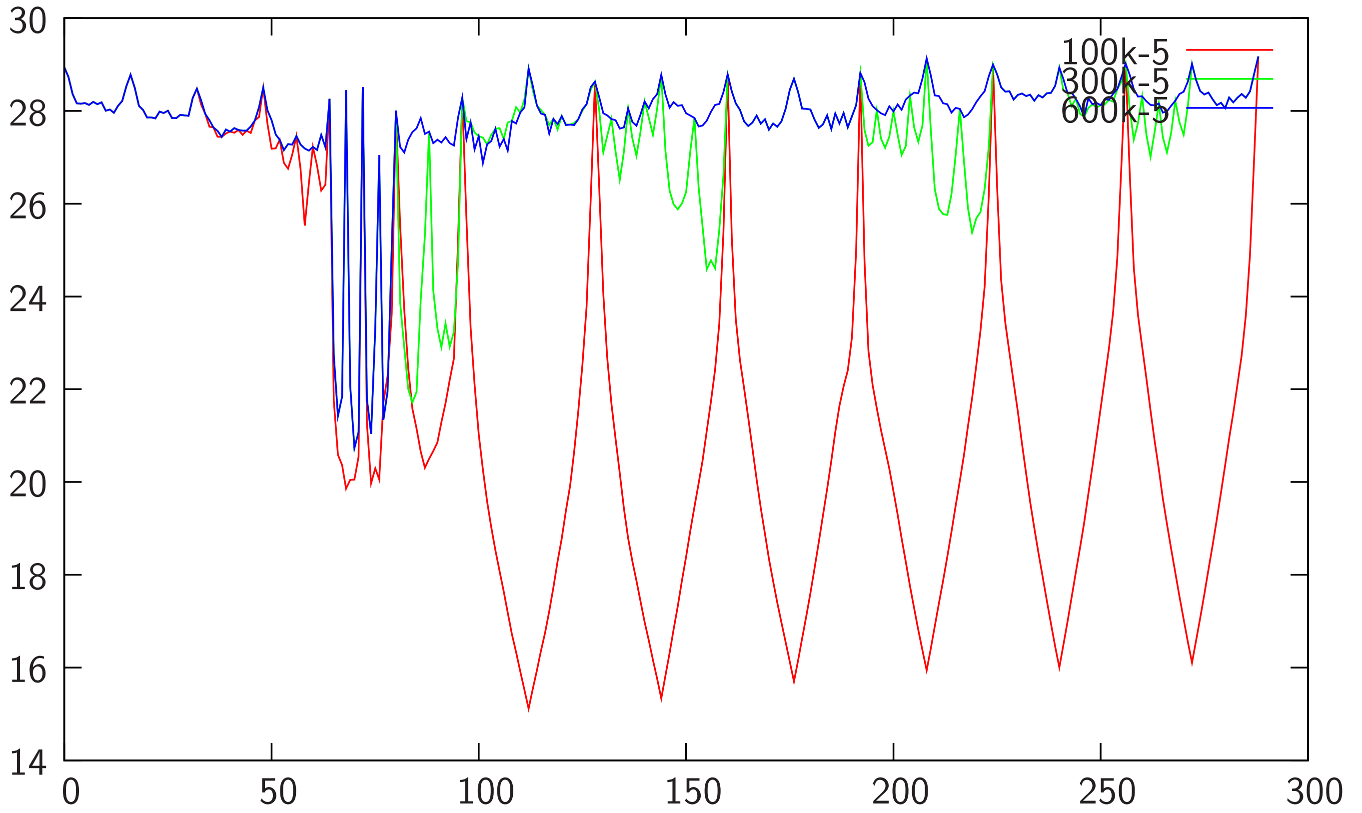

Figura 29. Recepción del vídeo con 5 capas.

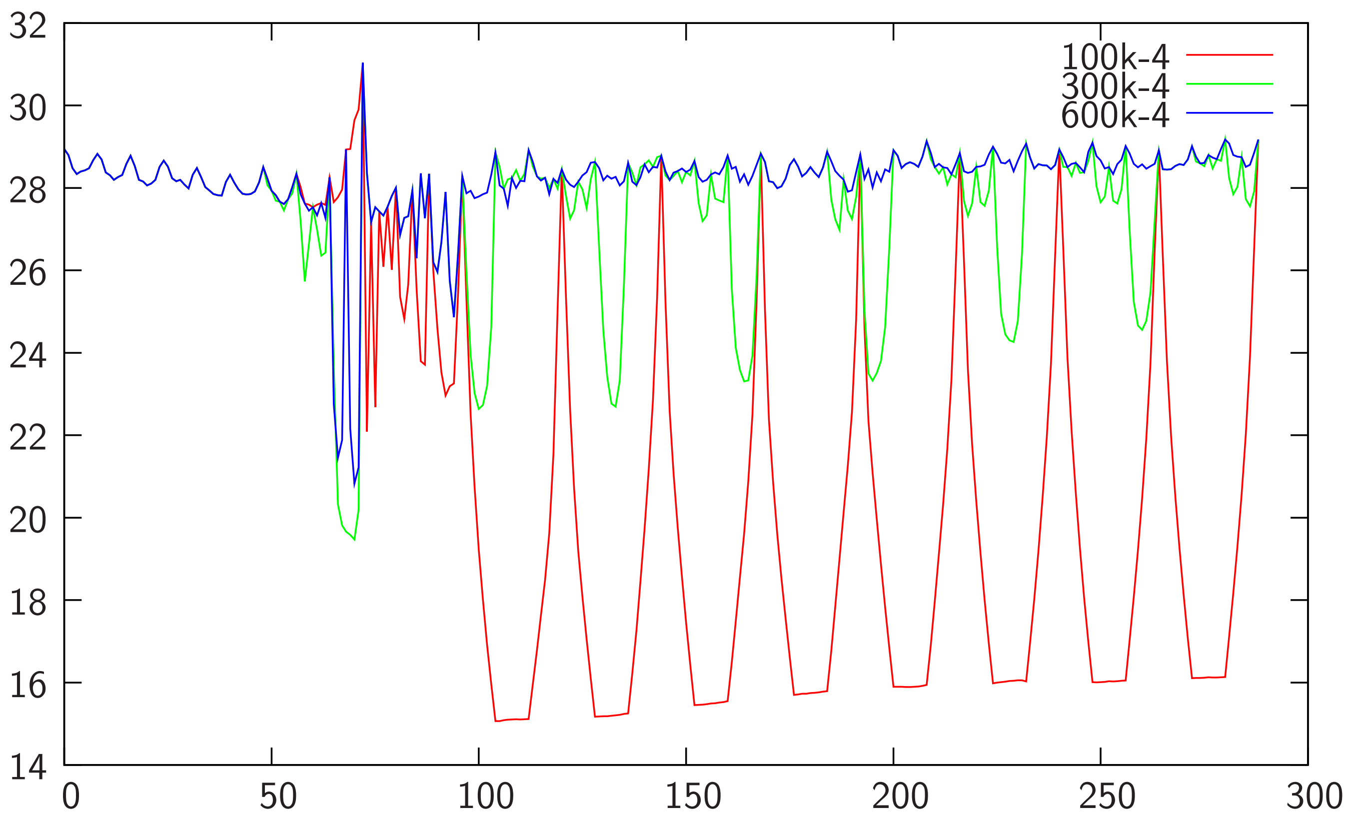

Figura 30. Recepción del vídeo con 4 capas.

```
/home/cesar/MCTF/bin/info_j2k: temporal_levels=5
/home/cesar/MCTF/bin/info_j2k: pictures=289
/home/cesar/MCTF/bin/info_j2k: pictures_per_second=30
/home/cesar/MCTF/bin/info_j2k: GOP_size=16
/home/cesar/MCTF/bin/info_j2k: number_of_GOPs=18
/home/cesar/MCTF/bin/info_j2k: GOP_time=0.533333333333
/home/cesar/MCTF/bin/info_j2k: All the values are given in thousands of bits per second (Kbps).
```

	TRL4			TRL3			TRL2			TRL1			TRL0	
GOP#	low_4		motion_4	high_4		motion_3	high_3		motion_2	high_2		motion_1	high_1	Total
0000	104	-	0	0	--	0	0	----	0	0	--------	0	0	104
0001	108	B	10	91	BB	15	119	BBBB	20	90	BBBBBBBB	25	47	528
0002	118	B	11	99	BB	16	131	BBBB	19	106	BBBBBBBB	29	58	590
0003	115	B	11	114	BB	16	147	BBBB	20	135	BBBBBBBB	30	87	679
0004	119	B	12	113	BB	18	167	BBBB	24	172	BBBBBBBB	34	136	799
0005	125	B	15	91	BB	22	159	BBBB	37	251	BBBBBBBB	55	320	1077
0006	125	B	9	85	BB	15	139	BBBB	25	222	BBBBBBBB	35	180	840
0007	115	B	10	95	BB	17	139	BBBB	23	159	BBBBBBBB	32	119	713
0008	115	B	10	85	BB	17	126	BBBB	26	144	BBBBBBBB	34	145	705
0009	112	B	10	81	BB	15	114	BBBB	23	156	BBBBBBBB	30	94	639
0010	115	B	10	83	BB	15	103	BBBB	20	119	BBBBBBBB	30	97	597
0011	113	B	11	94	BB	18	126	BBBB	24	130	BBBBBBBB	29	90	638
0012	107	B	10	85	BB	16	118	BBBB	20	124	BBBBBBBB	30	93	608
0013	105	B	10	83	BB	16	103	BBBB	21	119	BBBBBBBB	28	88	576
0014	108	B	11	89	BB	15	106	BBBB	20	115	BBBBBBBB	26	98	591
0015	102	B	10	81	BB	15	105	BBBB	20	123	BBBBBBBB	27	99	587
0016	105	B	10	81	BB	15	103	BBBB	20	125	BBBBBBBB	26	95	584
0017	104	B	9	75	BB	13	99	BBBB	18	113	BBBBBBBB	30	116	582
0018	97	B	10	88	BB	16	96	BBBB	19	114	BBBBBBBB	28	102	575
Average	112		11	90		16	122		22	140		31	115	662

Total average: 662.11 Kbps (notice that the first GOP is not included)

0047	453	B	21	105	BB	27	81	690
0048	429	B	23	154	BB	32	95	735
0049	443	B	22	121	BB	32	95	715
0050	440	B	21	108	BB	28	86	686
0051	439	B	22	122	BB	26	81	692
0052	421	B	21	123	BB	26	88	681
0053	452	B	21	116	BB	26	98	715
0054	444	B	21	123	BB	27	91	708
0055	441	B	22	112	BB	29	111	716
0056	434	B	20	108	BB	23	91	679
0057	422	B	20	111	BB	25	94	674
0058	421	B	22	116	BB	27	104	693
0059	422	B	21	132	BB	26	93	696
0060	411	B	21	135	BB	28	104	701
0061	418	B	21	129	BB	26	93	688
0062	415	B	20	137	BB	26	94	695
0063	431	B	21	117	BB	24	101	696
0064	423	B	19	116	BB	27	93	680
0065	428	B	20	108	BB	26	97	681
0066	415	B	20	113	BB	32	130	712
0067	420	B	20	122	BB	32	132	727
0068	417	B	21	111	BB	30	106	687
0069	421	B	20	112	BB	30	98	683
0070	415	B	21	115	BB	25	97	674
0071	421	B	21	112	BB	28	100	683
0072	389	B	20	116	BB	28	114	669
Average	445		23	140		31	115	755

Total average: 755.491666667 Kbps (notice that the first GOP is not↩ included)

APÉNDICE F
RESULTADOS DE LA RECEPCIÓN DE UN VÍDEO COMPRIMIDO CON 2 CAPAS

Esta es la tabla de la información del vídeo comprimido con 2 capas:

```
/home/cesar/MCTF/bin/info_j2k: temporal_levels=2
/home/cesar/MCTF/bin/info_j2k: pictures=289
/home/cesar/MCTF/bin/info_j2k: pictures_per_second=30
/home/cesar/MCTF/bin/info_j2k: GOP_size=2
/home/cesar/MCTF/bin/info_j2k: number_of_GOPs=144
/home/cesar/MCTF/bin/info_j2k: GOP_time=0.0666666666667
/home/cesar/MCTF/bin/info_j2k: All the values are given in ↩
     thousands of bits per second (Kbps).
```

	TRL1			TRL0	
GOP#	low_1		motion_1	high_1	Total
0000	836	-	0	0	836
0001	850	B	27	39	917
0002	783	B	24	51	860
0003	839	B	24	41	904
0004	840	B	26	53	920
0005	851	B	25	49	926
0006	880	B	26	44	951
0007	876	B	26	50	953
0008	869	B	26	49	944
0009	885	B	25	45	956
0010	909	B	32	58	1000
0011	862	B	30	55	948
0012	883	B	28	59	971
0013	894	B	33	54	982
0014	916	B	31	67	1015
0015	930	B	27	67	1025
0016	946	B	31	65	1042
0017	916	B	30	68	1015
0018	934	B	27	83	1045
0019	951	B	32	62	1045
0020	933	B	29	75	1037
0021	937	B	32	77	1047
0022	922	B	28	87	1038
0023	887	B	28	108	1024
0024	920	B	29	139	1089
0025	917	B	26	98	1041
0026	948	B	29	98	1076
0027	962	B	33	90	1086
0028	955	B	28	92	1076
0029	946	B	31	103	1081
0030	939	B	33	88	1062
0031	922	B	31	139	1093
0032	959	B	52	382	1394
0033	884	B	35	272	1192
0034	637	B	45	235	918
0035	464	B	82	577	1123
0036	540	B	85	502	1129
0037	998	B	75	522	1596
0038	1021	B	35	129	1185
0039	1037	B	34	176	1248
0040	1000	B	33	148	1181
0041	1012	B	44	198	1255
0042	969	B	36	115	1121
0043	910	B	35	295	1241
0044	1053	B	32	165	1251
0045	1026	B	30	174	1231
0046	1021	B	50	221	1293
0047	994	B	35	154	1185
0048	1003	B	36	121	1162
0049	984	B	31	184	1200
0050	1004	B	35	121	1161
0051	1011	B	30	86	1128
0052	955	B	34	107	1097
0053	973	B	35	114	1123
0054	933	B	29	106	1069
0055	927	B	27	88	1042
0056	924	B	36	149	1110
0057	903	B	35	112	1052
0058	897	B	34	120	1051
0059	917	B	45	279	1242
0060	936	B	50	198	1185
0061	943	B	35	135	1113
0062	928	B	36	143	1108
0063	933	B	33	91	1058
0064	924	B	32	84	1041
0065	917	B	32	81	1031
0066	916	B	30	94	1042
0067	913	B	35	85	1034
0068	910	B	46	170	1128
0069	890	B	31	90	1012
0070	895	B	32	76	1004
0071	912	B	30	86	1029
0072	900	B	29	68	998
0073	866	B	32	95	994
0074	880	B	33	81	994
0075	869	B	32	77	980
0076	893	B	30	100	1024
0077	889	B	33	122	1045
0078	887	B	33	127	1047
0079	889	B	28	81	999
0080	923	B	31	96	1051
0081	904	B	33	97	1034
0082	892	B	28	75	997
0083	898	B	31	102	1031
0084	911	B	31	87	1031
0085	915	B	35	97	1047
0086	899	B	33	89	1022
0087	890	B	27	86	1004
0088	906	B	28	84	1019
0089	872	B	29	78	979
0090	881	B	35	104	1021
0091	872	B	36	101	1011
0092	897	B	34	108	1039

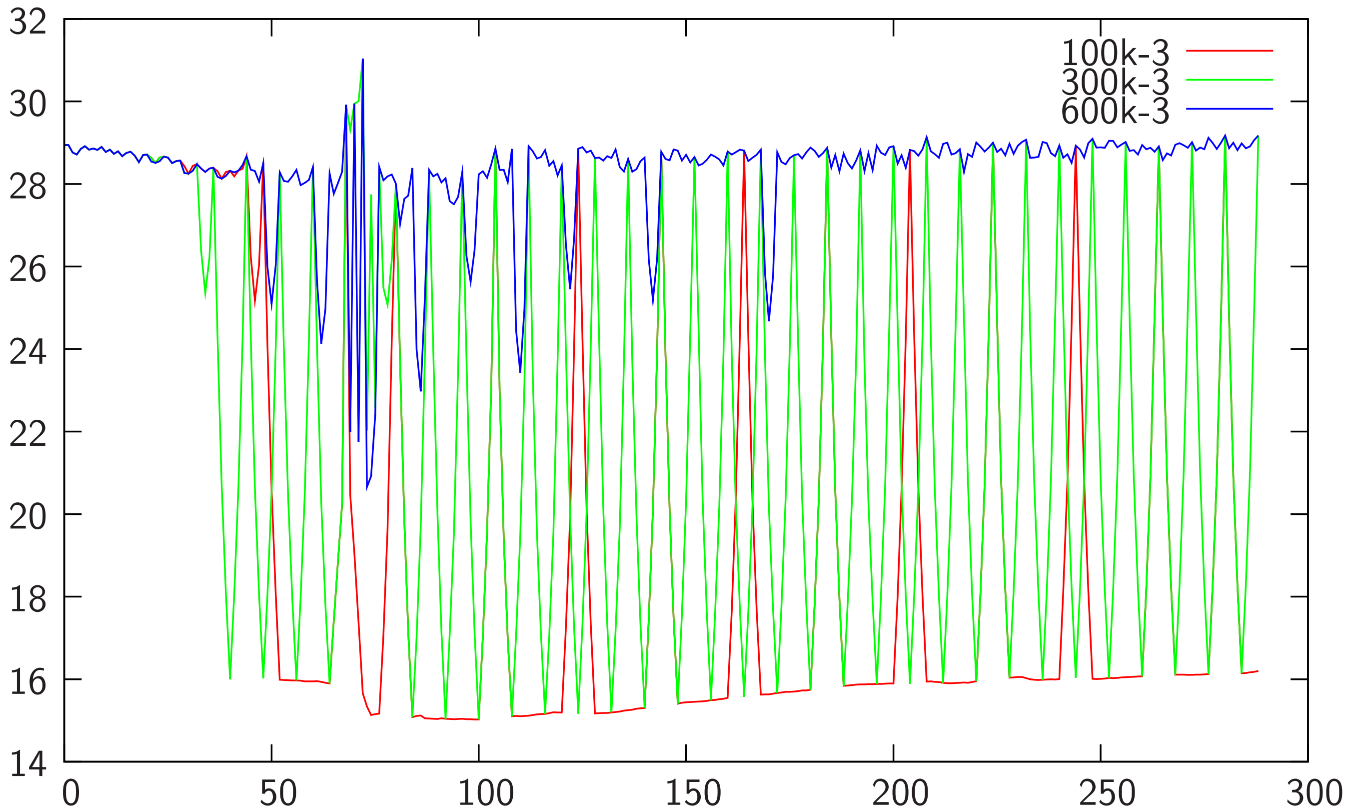

Figura 31. Recepción del vídeo con 3 capas.

```
0093        887 B     27       83       998
0094        906 B     29       79      1015
0095        906 B     32       91      1030
0096        859 B     34       98       993
0097        897 B     27       97      1022
0098        886 B     30       93      1010
0099        889 B     32      100      1022
0100        881 B     27       73       982
0101        887 B     28       68       983
0102        879 B     28       93      1002
0103        902 B     28       88      1018
0104        842 B     30       89       962
0105        872 B     32       98      1003
0106        905 B     27       98      1031
0107        877 B     27       90       995
0108        889 B     28       92      1010
0109        882 B     33      118      1035
0110        882 B     29      103      1014
0111        891 B     27       94      1013
0112        869 B     27       88       985
0113        870 B     27       90       989
0114        844 B     27       98       970
0115        827 B     29      122       979
0116        843 B     33       87       963
0117        867 B     29       95       993
0118        845 B     27       90       963
0119        825 B     29      111       966
0120        822 B     28       97       948
0121        833 B     28       86       949
0122        836 B     29      100       966
0123        806 B     28       93       928
0124        830 B     32       95       958
0125        841 B     28      100       970
0126        863 B     28      101       993
0127        829 B     27       92       949
0128        847 B     27       93       967
0129        844 B     27       95       966
0130        857 B     27      100       985
0131        846 B     32      133      1012
0132        830 B     32      127       990
0133        821 B     36      144      1002
0134        840 B     31      119       990
0135        825 B     28       96       950
0136        834 B     33      116       984
0137        833 B     34       94       962
0138        842 B     33      102       978
0139        826 B     28       97       952
0140        830 B     28       96       955
0141        803 B     33      101       937
0142        842 B     28       99       969
0143        837 B     31      123       992
0144        779 B     32      104       916
---- -------- --------------- --------
Average     890       32      115      1038

Total average: 1038.09916667 Kbps (notice that the first GOP is not↩
       included)
```

Apéndice G
Resultados de la recepción de un vídeo comprimido con 1 capas

Esta es la tabla de la información del vídeo comprimido con 1 capas:

```
/home/cesar/MCTF/bin/info_j2k: temporal_levels=1
/home/cesar/MCTF/bin/info_j2k: pictures=289
/home/cesar/MCTF/bin/info_j2k: pictures_per_second=30
/home/cesar/MCTF/bin/info_j2k: GOP_size=1
/home/cesar/MCTF/bin/info_j2k: number_of_GOPs=289
/home/cesar/MCTF/bin/info_j2k: GOP_time=0.0333333333333
/home/cesar/MCTF/bin/info_j2k: All the values are given in ↩
      thousands of bits per second (Kbps).

         TRL0
GOP#    low_0     Total
---- -------- --------
0000     1673      1673
0001     1673      1673
0002     1701      1701
0003     1661      1661
0004     1567      1567
0005     1687      1687
0006     1678      1678
0007     1725      1725
0008     1681      1681
0009     1716      1716
0010     1703      1703
0011     1696      1696
0012     1761      1761
0013     1764      1764
0014     1752      1752
0015     1750      1750
0016     1738      1738
0017     1729      1729
0018     1770      1770
0019     1779      1779
0020     1818      1818
0021     1773      1773
0022     1725      1725
0023     1732      1732
0024     1766      1766
0025     1772      1772
0026     1788      1788
0027     1809      1809
0028     1833      1833
0029     1854      1854
0030     1860      1860
0031     1903      1903
0032     1892      1892
0033     1803      1803
0034     1832      1832
```

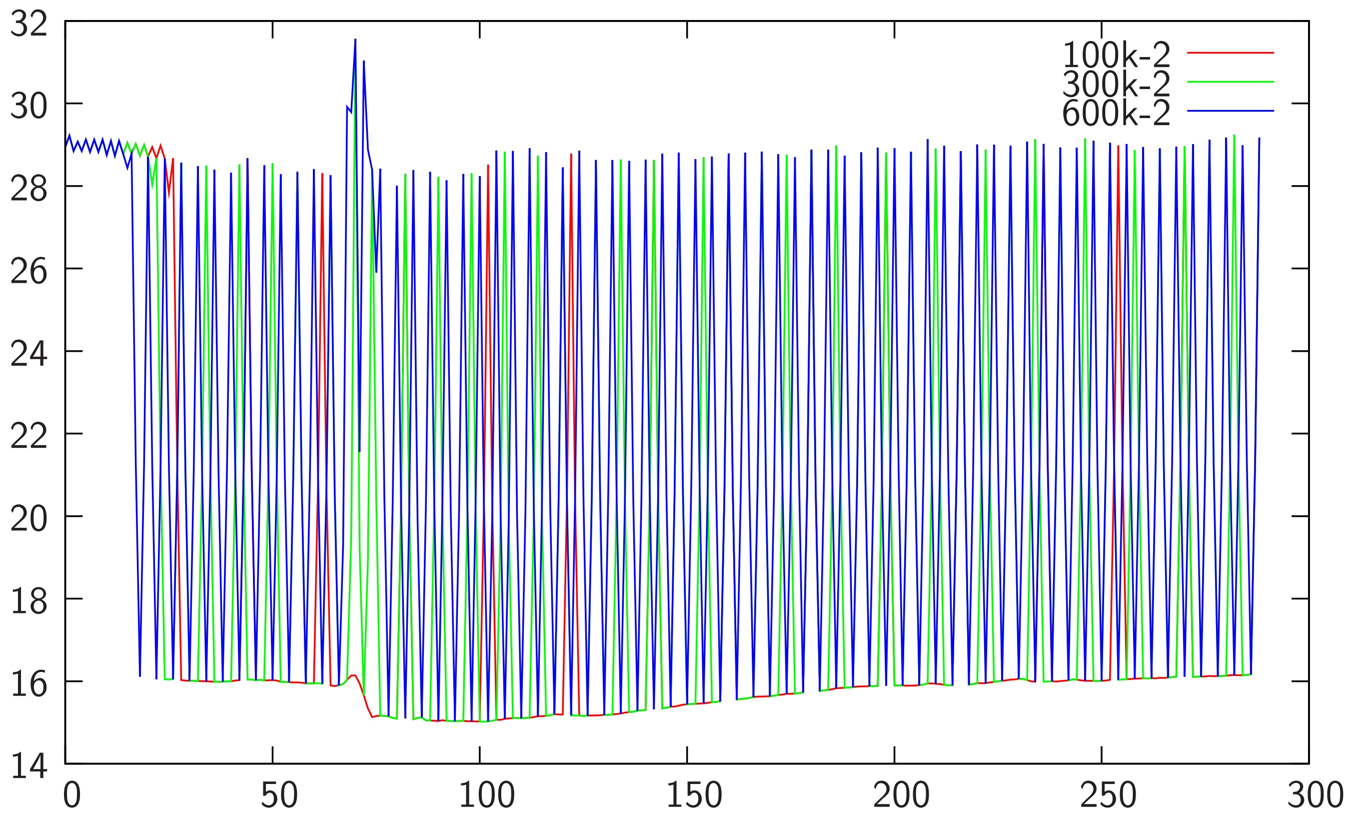

Figura 32. Recepción del vídeo con 2 capas.

```
0035      1860      1860
0036      1868      1868
0037      1843      1843
0038      1902      1902
0039      1929      1929
0040      1866      1866
0041      1875      1875
0042      1875      1875
0043      1861      1861
0044      1844      1844
0045      1842      1842
0046      1774      1774
0047      1875      1875
0048      1840      1840
0049      1919      1919
0050      1834      1834
0051      1881      1881
0052      1897      1897
0053      1940      1940
0054      1925      1925
0055      1903      1903
0056      1910      1910
0057      1925      1925
0058      1893      1893
0059      1902      1902
0060      1879      1879
0061      1908      1908
0062      1845      1845
0063      1968      1968
0064      1918      1918
0065      1895      1895
0066      1768      1768
0067      1593      1593
0068      1275      1275
0069      1029      1029
0070       928       928
0071       904       904
0072      1081      1081
0073      1492      1492
0074      1996      1996
0075      2003      2003
0076      2043      2043
0077      2117      2117
0078      2075      2075
0079      2042      2042
0080      2000      2000
0081      2034      2034
0082      2025      2025
0083      2036      2036
0084      1938      1938
0085      1907      1907
0086      1821      1821
0087      2006      2006
0088      2107      2107
0089      2079      2079
0090      2052      2052
0091      2035      2035
0092      2042      2042
0093      2054      2054
0094      1989      1989
0095      2026      2026
0096      2007      2007
0097      1997      1997
0098      1969      1969
0099      2015      2015
0100      2008      2008
0101      2018      2018
0102      2022      2022
0103      1962      1962
0104      1911      1911
0105      1951      1951
0106      1947      1947
0107      1909      1909
0108      1867      1867
0109      1898      1898
0110      1854      1854
0111      1887      1887
0112      1848      1848
0113      1875      1875
0114      1807      1807
0115      1797      1797
0116      1794      1794
0117      1845      1845
0118      1835      1835
0119      1843      1843
0120      1873      1873
0121      1870      1870
0122      1886      1886
0123      1818      1818
0124      1857      1857
0125      1829      1829
0126      1867      1867
0127      1839      1839
0128      1848      1848
0129      1820      1820
0130      1835      1835
0131      1828      1828
0132      1833      1833
0133      1835      1835
0134      1827      1827
0135      1792      1792
0136      1821      1821
0137      1806      1806
0138      1780      1780
0139      1787      1787
0140      1790      1790
0141      1809      1809
0142      1824      1824
0143      1781      1781
0144      1800      1800
0145      1779      1779
0146      1733      1733
0147      1762      1762
0148      1761      1761
```

```
0149     1739     1739
0150     1739     1739
0151     1785     1785
0152     1787     1787
0153     1797     1797
0154     1779     1779
0155     1832     1832
0156     1774     1774
0157     1796     1796
0158     1778     1778
0159     1784     1784
0160     1847     1847
0161     1809     1809
0162     1808     1808
0163     1828     1828
0164     1785     1785
0165     1787     1787
0166     1796     1796
0167     1840     1840
0168     1823     1823
0169     1795     1795
0170     1830     1830
0171     1709     1709
0172     1798     1798
0173     1806     1806
0174     1781     1781
0175     1791     1791
0176     1812     1812
0177     1794     1794
0178     1744     1744
0179     1765     1765
0180     1763     1763
0181     1728     1728
0182     1745     1745
0183     1774     1774
0184     1794     1794
0185     1804     1804
0186     1775     1775
0187     1793     1793
0188     1813     1813
0189     1883     1883
0190     1812     1812
0191     1827     1827
0192     1719     1719
0193     1783     1783
0194     1795     1795
0195     1756     1756
0196     1773     1773
0197     1793     1793
0198     1778     1778
0199     1737     1737
0200     1763     1763
0201     1723     1723
0202     1774     1774
0203     1774     1774
0204     1759     1759
0205     1762     1762
0206     1804     1804
0207     1739     1739
0208     1684     1684
0209     1731     1731
0210     1744     1744
0211     1804     1804
0212     1810     1810
0213     1742     1742
0214     1755     1755
0215     1734     1734
0216     1778     1778
0217     1773     1773
0218     1765     1765
0219     1718     1718
0220     1764     1764
0221     1788     1788
0222     1782     1782
0223     1730     1730
0224     1738     1738
0225     1734     1734
0226     1740     1740
0227     1696     1696
0228     1688     1688
0229     1676     1676
0230     1655     1655
0231     1646     1646
0232     1686     1686
0233     1760     1760
0234     1735     1735
0235     1718     1718
0236     1690     1690
0237     1631     1631
0238     1650     1650
0239     1718     1718
0240     1644     1644
0241     1625     1625
0242     1666     1666
0243     1648     1648
0244     1672     1672
0245     1648     1648
0246     1613     1613
0247     1673     1673
0248     1661     1661
0249     1662     1662
0250     1682     1682
0251     1722     1722
0252     1726     1726
0253     1709     1709
0254     1659     1659
0255     1670     1670
0256     1694     1694
0257     1724     1724
0258     1688     1688
0259     1689     1689
0260     1715     1715
0261     1711     1711
0262     1692     1692
0263     1681     1681
0264     1661     1661
0265     1716     1716
0266     1642     1642
0267     1694     1694
0268     1680     1680
0269     1674     1674
0270     1650     1650
0271     1675     1675
0272     1669     1669
0273     1684     1684
0274     1667     1667
0275     1726     1726
0276     1685     1685
0277     1630     1630
0278     1652     1652
0279     1617     1617
0280     1661     1661
0281     1585     1585
0282     1607     1607
0283     1631     1631
0284     1684     1684
0285     1611     1611
0286     1674     1674
0287     1630     1630
0288     1558     1558
0289        0        0
---- -------- --------
Average  1776     1776

Total average: 1776.65190311 Kbps (notice that the first GOP is not↩
         included)
```

APÉNDICE H
COMPARACIÓN DE RESULTADOS CON UN ANCHO DE BANDA DE 100KBS

APÉNDICE I
COMPARACIÓN DE RESULTADOS CON UN ANCHO DE BANDA DE 600KBS

APÉNDICE J
TEMPORIZACIÓN DEL TRABAJO REALIZADO

Si tuvieramos que dividir en proyecto de forma temporal, podríamos hacerlo en 5 grandes bloques, atendiendo al tiempo que se le ha dedicado a cada uno de ellos:

1. Búsqueda de antecedentes y planteamiento del proyecto a construir. **Tiempo dedicado: 20 días.**
2. Creación del código de la aplicación Cliente. **Tiempo dedicado: 45 días.**
3. Correción de errores para conseguir los objetivos planteados inicialmente. **Tiempo dedicado: 95 días.**
4. Fase de pruebas. **Tiempo dedicado: 40 días.**
5. Elaboración de la memoria del proyecto. **Tiempo dedicado: 35 días.**

AGRADECIMIENTOS

Le quiero agradecer a Vicente González Ruiz la oportunidad que me ha brindado con este proyecto y por su paciencia, y también a mi familia por haber estado apoyándome durante todo este tiempo.

BIBLIOGRAFÍA

[1] A. Skodras, C. Christopoulos, and T. Ebrahimi, “”the jpeg 2000 still image compression standard”,” *IEEE Signal Processing Magazine*, vol. 18, no. 5, 2001.

[2] T. J. P. E. G. (JPEG), *Recommendation T.81: Digital Compression and Coding of Continuous-tone Still Images*, International Telecommunication Union (ITU), September 1992.

[3] A. Marcos, *Compresión de imágenes. Norma JPEG*. Editorial Ciencia 3, 1999.

[4] “Motion jpeg,” http://en.wikipedia.org/wiki/Motion_JPEG.

[5] G. K. Wallace, “The JPEG Still Picture Compression Standard,” *Communications of the ACM*, vol. 34, no. 4, pp. 30 – 44, April 1991, se puede conseguir en ftp://ftp.uu.net/graphics/jpeg/wallace.ps.Z.

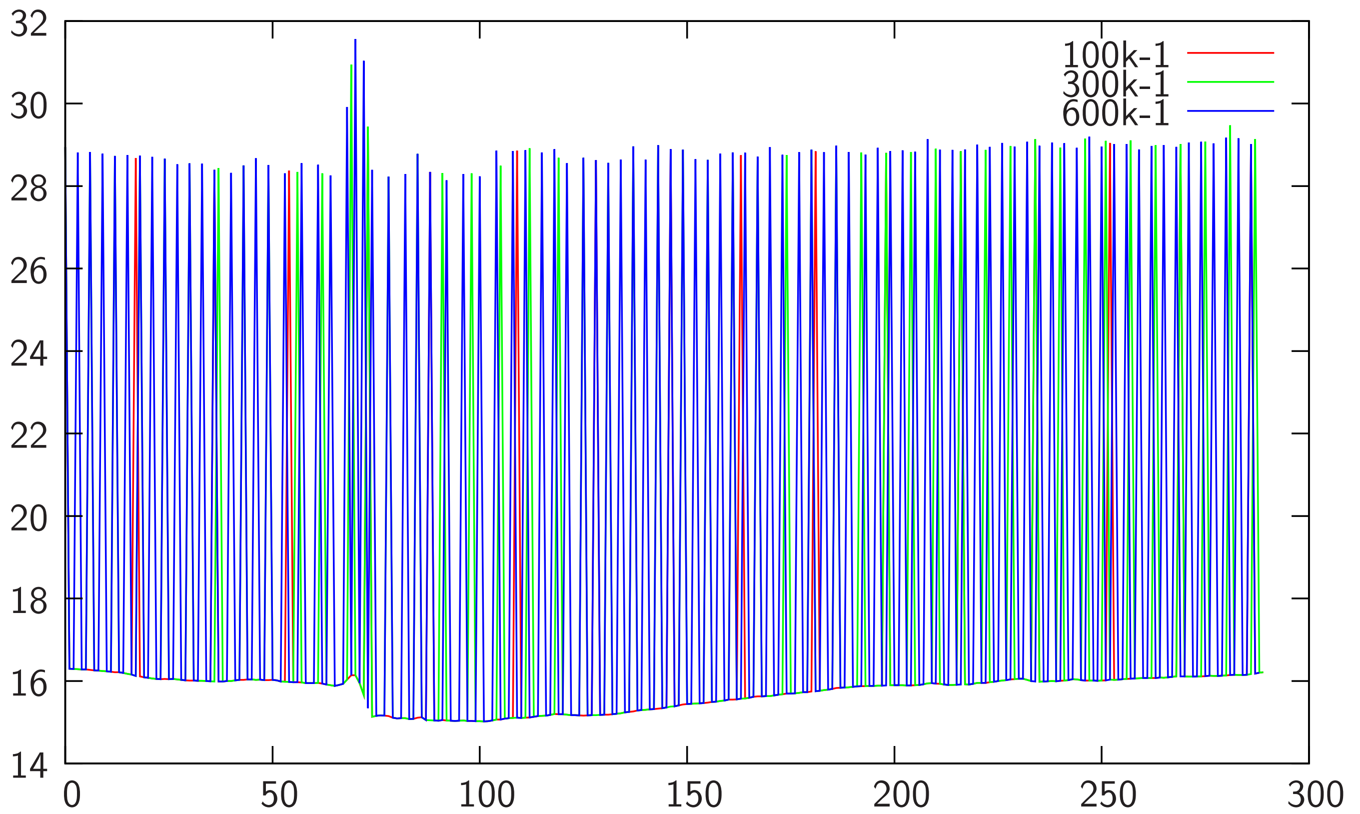

Figura 33. Recepción del vídeo con 1 capas.

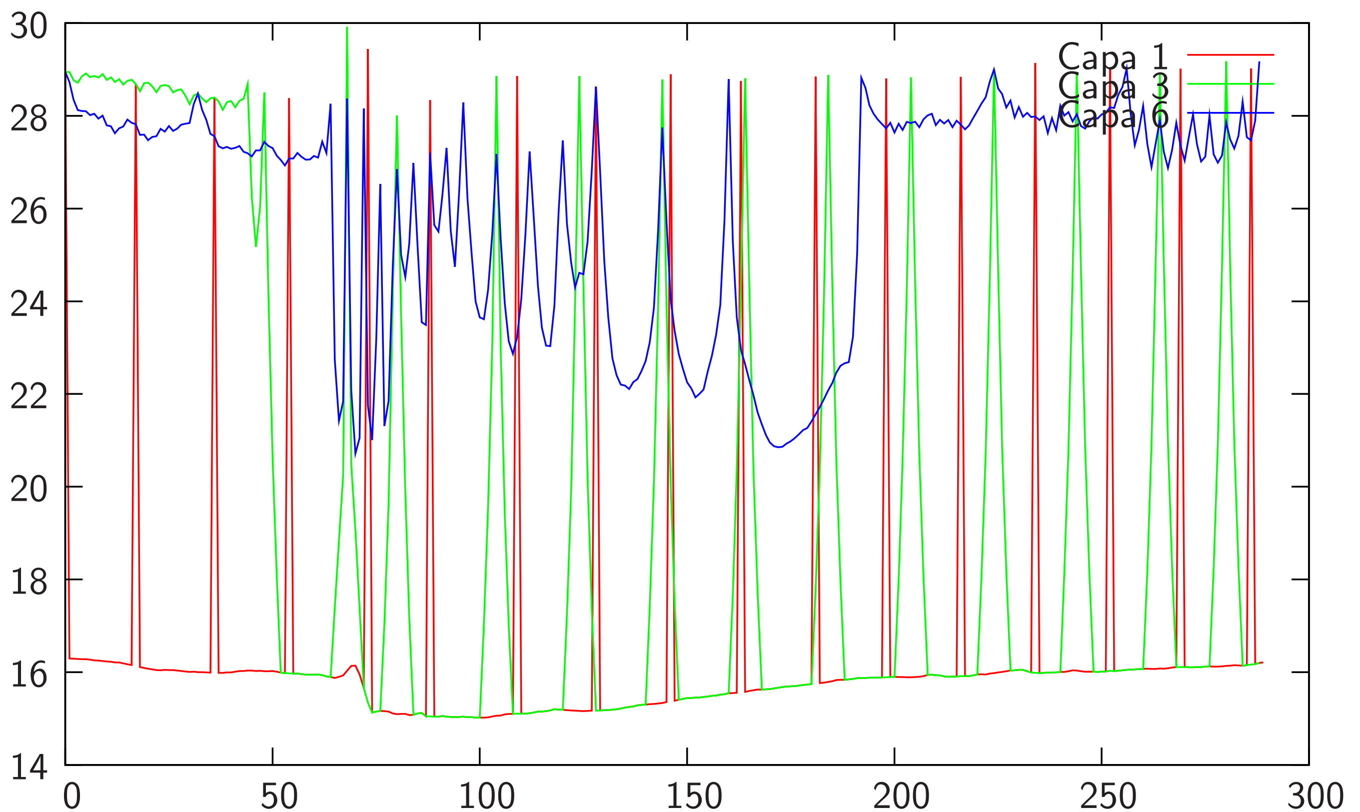

Figura 34. Resultados con 1 capa, 3 capas y 6 capas con un ancho de banda de 100kBs.

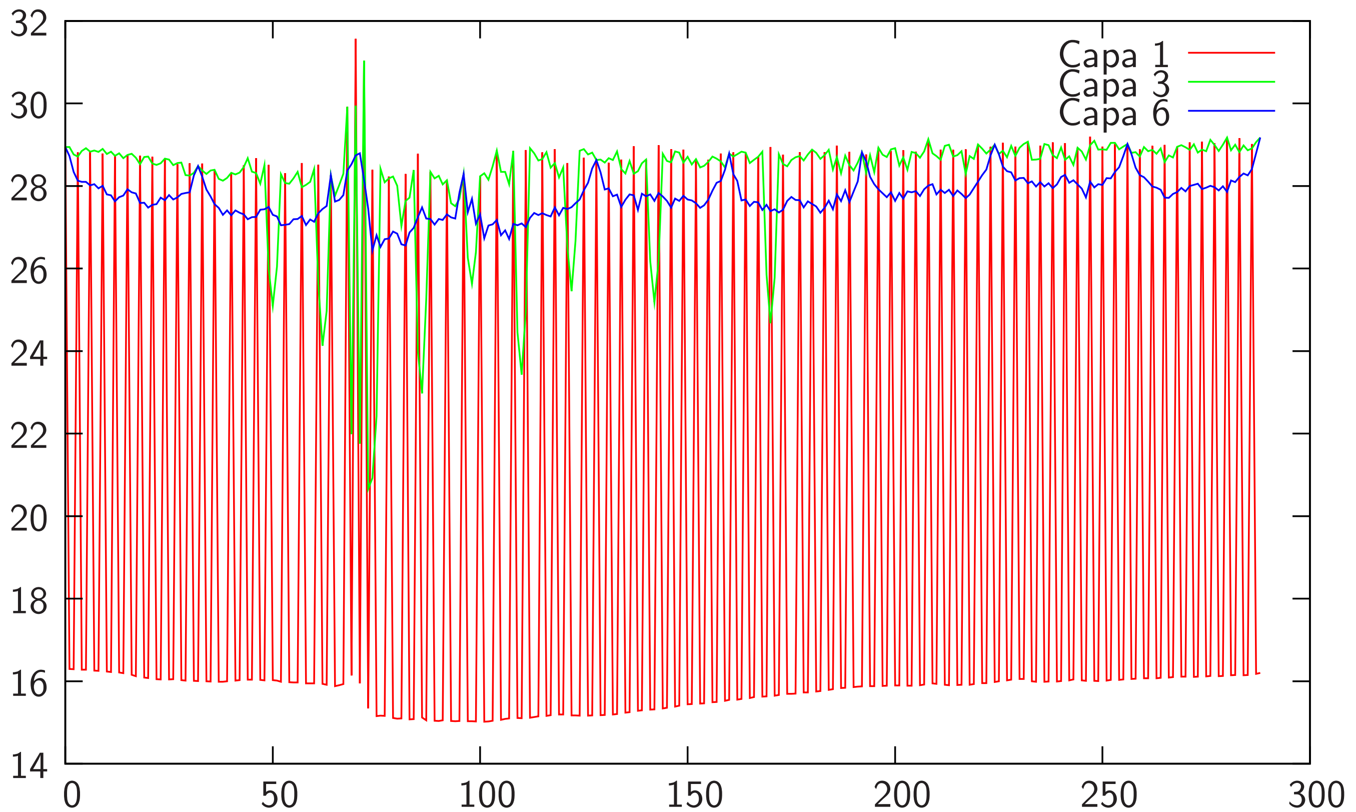

Figura 35. Resultados con 1 capa, 3 capas y 6 capas con un ancho de banda de 600 kBs.

[6] I. T. 11172-5:1998, "Coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s," *International Organization for Standardization (ISO)*, 1998.

[7] M. N. and T. D, "Spatial scalability and compression efficiency within a flexible motion compensated 3d-dwt," *IEEE International Conference on Image Processing (ICIP2004)*, 2004.

[8] T. D., "Successive refinement of video: fundamental issues, past efforts and new directions," *International Symposium on Visual Communications and Image Processing (VCIP2003), SPIE*, 2003.

[9] L. G. Coloma, *Redes ATM. Principios de interconexión y su aplicación*. Ra-Ma, 2000.

[10] F. Pereira and T. Erahimi, "The mpeg-4 book," *Pearson Education*, 2002.

[11] I. JTC1/SC29/WG11, "Coding of moving pictures and audio," *Technical report, International Organisation for Standardisation*, 2002.

[12] F. Pereira and P. Nunes, "Levels for mpeg-4 visual profiles," *MPEG-4 Industry Forum*, 2002.

[13] W. T. S. G. B. G. L. A., "Overview of the h.264/avc video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, 2003.

[14] ——, "Overview of the h.264/avc video coding standard," *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY*, 2003.

[15] J. R. Ohm, "Three-dimensional subband coding with motion compensation," *IEEE Transactions on Image Processing*, 1994.

[16] H. S. D. Marpe and T. Wiegand, "Vision general de la extension de vídeo escalable de codificacion de la norma h.264/avc," *IEEE Transactions on Circuitos y Sistemas de Tecnología de Vídeo*, 2007.

[17] T. J. P. E. Group, *ISO/IEC 15444-1 (JPEG2000, Part 1)*.

[18] D. Taubman, "High performance scalable image compression with ebcot," *IEEE Transactions on Image Processing*, vol. 9, no. 7, pp. 1158–1170, 2000.

[19] D. S. Taubman and M. W. Marcellin, "Jpeg2000. image compression fundamentals, standards and practice," *Kluwer Academic Publishers*, 2002.

[20] F. S. F. G. and J. Mohr, "Motion jpeg2000 for high quality video systems," *Consumer Electronics, IEEE Transactions on*, 2003.

[21] R. Leung and D. Taubman, "Transform and embedded coding techniques for maximum efficiency and random accessibility in 3d scalable compression," *IEEE Transactions on Image Processing*, 2004.

[22] L. R. and T. D., "Context modeling and accessibility for 3d scalable compression," *IEEE International Conference on Image Processing (ICIP2003)*, 2003.

[23] T. J. and T. D., "Optimal erasure protection assignment for scalable compressed data with small packets and short channel codewords," *EURASIP Journal on Applied Signal Processing; Special issue on Multimedia over IP and Wireless Networks*, 2004.

[24] T. D. and T. J., "Optimal erasure protection for scalably compressed video streams with limited retransmission," *IEEE Transactions on Image Processing*, 2004.

[25] S. A. and T. D., "Lifting-based invertible motion adaptive transform (limat) framework for highly scalable video compression," *IEEE Transactions on Image Processing*, 2003.

[26] ——, "Highly scalable video compression using a lifting-based 3d wavelet transform with deformable mesh motion compensation," *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, 2002.

[27] ——, "Motion-compensated highly scalable video compression using an adaptive 3d wavelet transform based on lifting," *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, 2001.

[28] G. D. S. M.;, "Multi-frame motion estimation: application to motion compensated prediction," *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium on*, 1999.

[29] M. N. and T. D., "A flexible structure for fully scalable motion compensated 3d-dwt with emphasis on the impact of spatial scalability," *IEEE Transactions on Image Processing*, 2004.

[30] N. Mehrseresht and D. Taubman, "An efficient content-adaptive mc 3d-dwt with enhanced spatial and temporal scalability," *IEEE Transactions on Image Processing*, 2004.

[31] D. Taubman and A. Secker, "Highly scalable video compression with scalable motion coding," *IEEE International Conference on Image Processing (ICIP2003)*, 2003.

[32] H. S. D. Marpe; and T. Wiegand, "Overview of the scalable video coding extension of the h.264/avc standard," *IEEE Transactions on Circuits and Systems for Video Technology*, 2007.

[33] A. Haar, "Zur theorie der orthogolanen funktionen-systeme," *Mathematische Annalen*, 1910.

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

En la actualidad, para la reproducción de un vídeo a través de Internet es necesario tener que parar el vídeo para darle tiempo para poder reproducir parte del vídeo sin tener que se pause.
Con este proyecto fin de carrera lo que se quiere conseguir es diseñar un servicio de streaming de vídeo escalable donde el cliente va a ver una reproducción de dicho vídeo acorde al ancho de banda que este disponga.
El lenguaje implementado para conseguir esto ha sido C.

Today, for playing a video over the Internet you must have to stop the video to give you time to play part of the video without having to pause.
With this final project what we want to achieve is to design a video streaming service scalable where the customer will see a reproduction of the video according to the bandwidth that this provides.
The language implemented to achieve this has been C.