Transmisión Progresiva de imágenes JPEG2000

Juan Pablo García Ortiz

Ingeniería Informática

Septiembre 2003

Director de proyecto

Vicente González Ruiz

Septiembre, 2003

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

A mis queridos padres y a mi hermana, que siempre han estado ahí cuando los he necesitado, apoyándome en todo momento.

A mi amada novia Rosa, cuyo cariño ha sido vital para mí.

Al tutor de mi proyecto, Vicente, por su inestimable ayuda y profesionalidad.

A todos, mi sincero agradecimiento.

El estándar JPEG2000 es a día de hoy uno de los estándares de compresión y descompresión de imágenes más funcionales y sofisticados que existen. Sus avanzadas características lo convierten en el estándar ideal a emplear cuando se requiera visualizar imágenes remotas de forma eficiente. Probablemente debido a una mayor complejidad que sus estándares anteriores (JPEG por ejemplo), JPEG2000 no está todavía ampliamente difundido y es raro encontrar su uso fuera de aplicaciones específicas tales como la visualización remota e interactiva de imágenes astronómicas y médicas.

Uno de los principales inconvenientes de las aplicaciones de visualización remota de imágenes basadas en JPEG2000 radica en la necesidad de un servidor específico para la interacción de los clientes de visualización. Por este motivo, el estándar contempla un protocolo de transmisión de datos específico llamado JPIP. Sin embargo, este proyecto trata de demostrar que dicha necesidad es realmente sólo una posibilidad, y que no es necesario disponer de un servidor de imágenes JPEG2000 basado en JPIP para que la manipulación interactiva de imágenes remotas sea posible.

En el presente proyecto se ha desarrollado una aplicación portable (independiente de la plataforma y del sistema operativo) capaz de visualizar progresivamente e interaccionar con imágenes remotas, empleando la infraestructura de comunicación más difundida hoy día: el protocolo HTTP y la red Internet.

El documento se divide en los siguientes capítulos:

- El Capítulo 1, titulado "El estándar JPEG2000", introduce el estándar de compresión y descompresión de imágenes JPEG2000. En él se habla sobre las posibilidades del estándar, su arquitectura, el formato de los archivos, etc.

- A continuación, el Capítulo 2 llamado "El protocolo JPIP", realiza una introducción al protocolo JPIP para la transmisión de imágenes JPEG2000. Se describen sus principales características, el mecanismo de comunicación, etc.

- El Capítulo 3, titulado "El protocolo HTTP/1.1", realiza una breve introducción al protocolo HTTP en su versión 1.1. Se analiza la estructura de los mensajes, así como las características avanzadas del protocolo.

- Seguidamente, en el Capítulo 4 que se llama "Kakadu", se introduce el paquete de software Kakadu, explicando de forma escueta las clases que forman su biblioteca de clases y funciones, así como los diferentes

programas que la acompaña. También se analiza la compatibilidad con el lenguaje de programación Java.

- Finalmente, el Capitulo 5 titulado "Sistema desarrollado", describe el sistema de software desarrollado para el presente proyecto. Se justifica la técnica empleada y el formato de archivo elegido, se analiza en detalle la estructura y funcionamiento de la aplicación, se evalúa su rendimiento y finalmente se proponen diversas mejoras.

Capítulo 1

El estándar JPEG2000

El estándar JPEG2000 es un sistema de compresión y descompresión de imágenes (de tonos de gris y de color) digitales en formato *raster*, propuesto recientemente por el JPEG (*Joint Photographic Experts Group*). Este estándar está publicado como un ISO/IEC estándar [8, 6] y consta de varias partes, empezando por la Parte 1, que define los requerimientos mínimos necesarios para la compresión/descompresión de las imágenes (codificación, sintaxis del *codestream*, ...). La primera parte sería lo mínimo necesario para cualquier implementación. Las partes sucesivas van añadiendo funcionalidad extra.

En el momento de la creación del presente documento, las partes existentes son las qué aparecen en la tabla 1. La Parte 9 del estándar será tratada más adelante en el presente documento.

1.1. Características principales

Algunas de las principales características del estándar JPEG2000 son las siguientes:

- **Compresión de imágenes *continuous-tone* y *bi-levels*:** El estándar es capaz de comprimir imágenes de tono continuo (*continuous-tone*) y binarias (*bi-levels*). Permite además una compresión de imágenes con un rango dinámico de profundidad de bits para cada componente, desde 1 bit, con una calidad comparable al estándar ITU-T G4, hasta los 32 bits, siendo, posiblemente, el único estándar que hasta el momento ofrece esta posibilidad.

- **Transmisión progresiva de las imágenes:** Ofrece una progresión

Parte	Descripción
1	Sistema base
2	Extensión. Añade funcionalidad al sistema base
3	JPEG 2000 para secuencias de vídeo
4	*Conformance*
5	Software de referencia (implementaciones en C y Java)
6	Formato de archivo compuesto
8	JPSEC, para habilitar opciones de seguridad
9	JPIP, protocolo para la transmisión progresiva de imágenes
10	JP3D, para imágenes tridimensionales
11	JPWL, para aplicaciones *wireless*

Tabla 1.1: Partes del estándar JPEG2000.

en calidad, en resolución, en posición espacial o en componente de imagen, lo cual es muy apropiado para la transmisión sobre enlaces de comunicación lentos. Estas progresiones, que se consiguen ordenando adecuadamente el *codestream*, pueden coexistir entremezcladas en una misma imagen. En el caso del presente proyecto, será necesario tener en cuenta esto, pero no será empleado.

- **Posibilidad de compresión con o sin perdida:** Gracias a la inclusión de una transformada wavelet reversible, el estándar JPEG2000 ofrece esta posibilidad.

- **Acceso aleatorio al *codestream* de la imagen así como posibilidad de procesamiento del mismo sin necesidad de descompresión:** Los *codestreams* ofrecen diversos mecanismos para soportar el acceso aleatorio en función de una región de interés. Además, permiten la rotación, la traslación y el escalado, entre otros, de la imagen que definen, sin necesidad de descompresión.

- **Robustez ante errores en los datos:** JPEG2000 define una serie de mecanismos (marcadores de sincronización, etc.) para permitir la detección de errores de los datos. Por ejemplo, las aplicaciones que hacen uso de canales de comunicación *wireless* harían uso de esta característica.

- **Arquitectura abierta:** La arquitectura del estándar es abierta, de forma que se define la base, que sería la Parte 1, a la cual se le va

añadiendo la funcionalidad que se requiera para aplicaciones específicas.

- **Mejor rendimiento a bajos *bit-rates*:** JPEG2000 ofrece un rendimiento superior a bajos *bit-rates* con respecto al resto de los estándares. Las aplicaciones de transmisión de imágenes a través de la red se benefician de esta característica.
- **Posibilidad de definir regiones de interés (*ROI*):** A menudo existen partes de una imagen que son más importantes que otras. El estándar permite definir ciertas ROIs en la imagen con el fin de que sean codificadas y transmitidas con mayor prioridad, mejor calidad y menor distorsión que el resto.

1.2. Arquitectura

La Parte 1 del estándar especifica una arquitectura básica, formada por un conjunto de etapas de procesamiento, que se puede ver en forma de diagrama de bloques en la figura 1.1. Dichas etapas son aplicadas a cada componente de cada *tile* de la imagen a comprimir.

Las etapas de la arquitectura del estándar JPEG2000 son las siguientes:

- ***Offset*:** Las muestras de la imagen, $x[\boldsymbol{n}]$, donde $\boldsymbol{n}$ representa el punto de coordenadas $[n_1, n_2]$ de la imagen, con una profundidad de B bits, deben ser valores con signo en el rango:

  $$-2^{B-1} \leq x[\boldsymbol{n}] \leq 2^{B-1}$$

  Por ello, si las muestras originales de la imagen son sin signo, como suele pasar en la mayoría de los casos, es necesario añadirles un *offset* de -2^{B-1}.

- **Transformada de color:** La transformada de color es opcional, ya que sólo puede ser aplicada cuando se poseen al menos tres componentes de color y las tres primeras son del mismo tamaño y la misma profundidad de bits. Se asume que estas primeras tres componentes son RGB, el rojo (*Red*), el verde (*Green*) y el azul (*Blue*).

  El objetivo de la transformada de color es convertir las componentes RGB de la imagen en otras componentes con un modelo de color diferente, con el fin de reducir la redundancia que existe entre los canales R, G y B, y poder así aumentar las tasas de compresión.

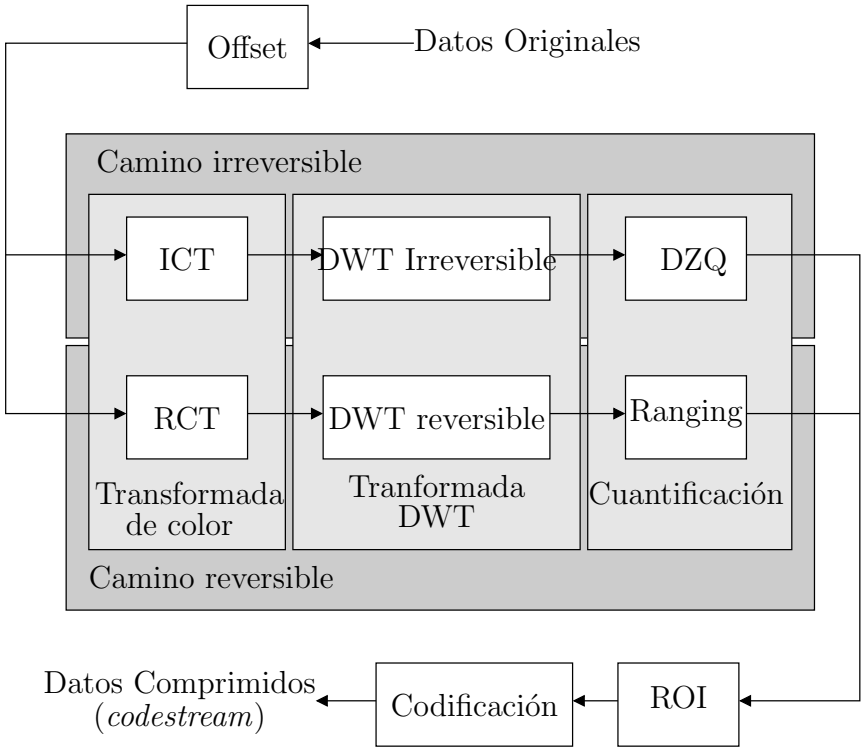

Figura 1.1: Arquitectura del estándar JPEG2000.

Como se puede apreciar en la figura 1.1, existen dos tipos de transformadas de color, una para el camino irreversible, la ICT, y otra para el camino reversible, la RCT.

La transformada ICT convierte de RGB a YCbCr. La transformada RCT convierte de RGB a Y'DbDr.

Como hemos comentado, asumimos que las tres componentes, $x_0[\boldsymbol{n}]$, $x_1[\boldsymbol{n}]$ y $x_2[\boldsymbol{n}]$, son el rojo, $x_R[\boldsymbol{n}]$, el verde, $x_G[\boldsymbol{n}]$, y el azul, $x_B[\boldsymbol{n}]$. La transformada ICT es:

$$x_Y[\boldsymbol{n}] \triangleq \alpha_R x_R[\boldsymbol{n}] + \alpha_G x_G[\boldsymbol{n}] + \alpha_B x_B[\boldsymbol{n}]$$
$$x_{Cb}[\boldsymbol{n}] \triangleq \frac{0{,}5}{1-\alpha_B}(x_B[\boldsymbol{n}] - x_Y[\boldsymbol{n}])$$
$$x_{Cr}[\boldsymbol{n}] \triangleq \frac{0{,}5}{1-\alpha_R}(x_R[\boldsymbol{n}] - x_Y[\boldsymbol{n}])$$

Donde $\alpha_R \triangleq 0{,}2999$, $\alpha_G \triangleq 0{,}587$ y $\alpha_B \triangleq 0{,}114$.

Con las mismas condiciones que para la transformada ICT, se define la transformada RCT como sigue:

$$x_{Y'}[\boldsymbol{n}] \triangleq \left\lfloor \frac{x_R[\boldsymbol{n}] + 2x_G[\boldsymbol{n}] + x_B[\boldsymbol{n}]}{4} \right\rfloor$$

$$x_{Db} \triangleq x_B[\boldsymbol{n}] - x_G[\boldsymbol{n}]$$

$$x_{Dr} \triangleq x_R[\boldsymbol{n}] - x_G[\boldsymbol{n}]$$

En nuestro caso, las imágenes van a cumplir la condición necesaria para aplicar la transformada de color, ya que van a ser imágenes en color con sólo tres componentes, RGB.

- **Transformada wavelet:**

  La segunda etapa de minimización entrópica se basa en la transformada wavelet discreta o DWT (*Discrete Wavelet Transform*). Esta representa los puntos de la imagen a comprimir en un dominio espacio-frecuencial que posee dos ventajas fundamentales: (1) reducir la correlación espacial de cada componente de la imagen, lo que es esencial de cara a maximizar las tasas de compresión, y (2) encontrar una representación multiresolución para cada componente, propiedad que puede ser de ayuda a la hora de procesar imágenes muy grandes.

  A continuación vamos a describir la DWT desde el punto de vista de la Teoría de los Bancos de Filtros [9].

  Según dicha teoría, una secuencia unidimensional de N muestras $x[n],\ n = 0, \cdots, N-1$ puede ser representada mediante dos secuencias $l[n],\ n = 0, \cdots, \frac{N}{2} - 1$ y $h[n],\ n = 0, \cdots, \frac{N}{2} - 1$ de tamaño mitad, donde $l[n]$ es el resultado de filtrar $x[n]$ usando un filtro paso bajo L y $h[n]$ es el resultado de filtrar $x[n]$ mediante un filtro paso alto H. Esta banco de dos filtros se define de forma que desde el punto de vista de la frecuencia, lo que el filtro L deja pasar es lo que H no deja pasar y viceversa. Por este motivo, la información contenida en las bandas $l[n]$ y $h[n]$ es la misma que en la secuencia original $x[n]$ que puede ser regenerada usando los correspondientes filtros de síntesis.

  Este proceso de descomponer una señal en dos señales, una de baja frecuencia y otra de alta puede aplicarse recursivamente a ambas subbandas. Cuando sólo se aplica a la banda de baja frecuencia, se habla

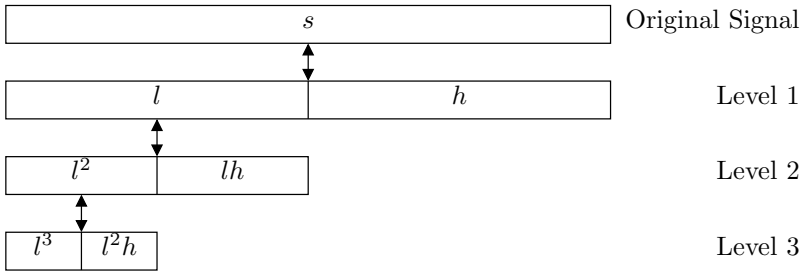

Figura 1.2: Una descomposición DWT diádica de 3 niveles.

de la DWT en su versión diádica (*dyadic*). Si se aplica a ambas, obtenemos la DWT en su versión paquetizada (*packet*). En el caso de la compresión de imágenes, la forma más utilizada en la diádica porque la energía se acumula principalmente en la zona de baja frecuencia.

En la figura 1.2 se muestra gráficamente el proceso de descomposición aplicado en la DWT diádica unidimensional. El número máximo de etapas o niveles es igual a $\log_2(N)$. A los coeficientes de las bandas $l^{\log_2(N)}$ y $l^j h[n],\ j = 0, \cdots, \log_2(N) - 1;\ n = 0, \cdots, \frac{N}{2^{j+1}} - 1$, donde j es el nivel de la descomposición, se les llama coeficientes *wavelet*.

Usando $l^n[0]$ y $l^j h[n]$ es posible obtener $\log_2(N)$ niveles de resolución a la hora de representar $x[n]$. Así, el primer nivel de resolución estaría formado por una señal constante e igual a $l^{\log_2(N)}[0]$, el segundo nivel utilizaría los coeficientes $l^{\log_2(N)}[0]$ y $l^{\log_2(N)-1}h[0]$, y así sucesivamente. Finalmente, nótese que se cumple que $x[n] = l^0[n]$.

La DWT se diferencia además de otras transformadas clásicas como la DFT (Discrete Fourier Transform) o la DCT (Discrete Cosine Transform) en un aspecto fundamental: se trata de una representación espacio-frecuencial, no únicamente frecuencial. Gracias a esto es posible seleccionar un subconjunto de coeficientes *wavelet* para reconstruir sólo una parte de la señal original. Esto es muy útil cuando procesamos imágenes muy grandes y sólo podemos visualizar una parte de la imagen.

Ya que estamos trabajando con imágenes y éstas tienen dos dimensiones, es necesario extender este proceso al caso bidimensional. Por suerte la DWT diádica es separable, lo que significa que podemos calcular la DWT diádica bidimensional aplicando la DWT diádica unidimensional primero a las filas y luego a las columnas de la imagen (o viceversa).

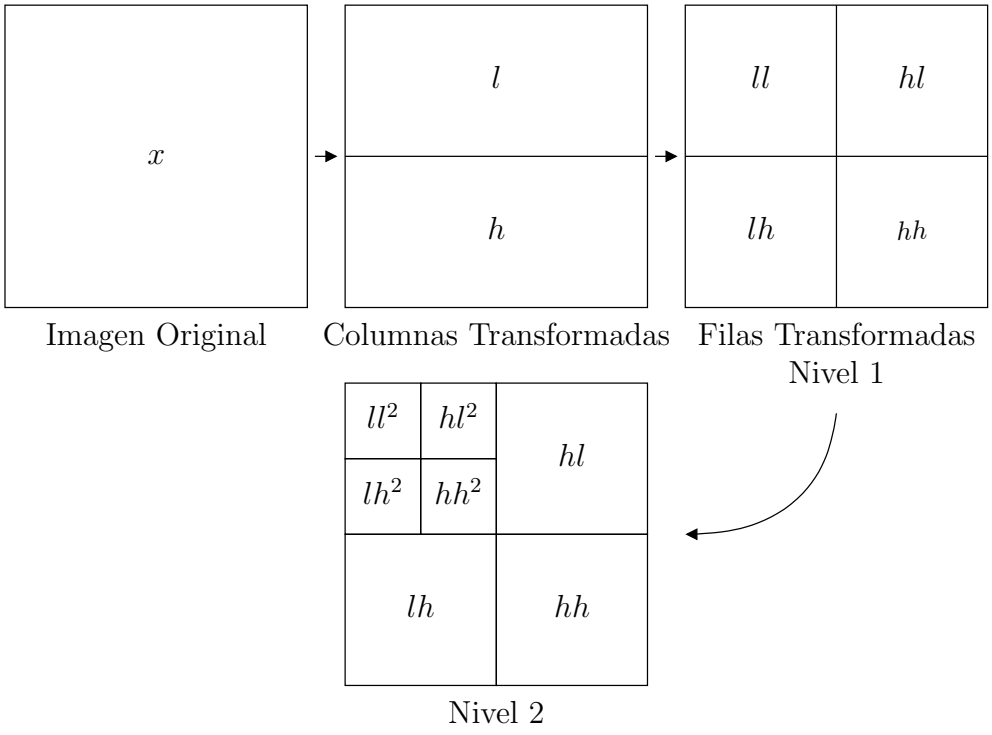

Figura 1.3: Una descomposición DWT diádica bidimensional de 2 niveles.

Este proceso se describe gráficamente en la figura 1.3. De esta forma, tras cada fase de descomposición se obtienen cuatro bandas en lugar de dos.

- **Cuantificación:** Cada coeficiente *wavelet* c se cuantifica mediante la siguiente expresión:

$$q_b[n] = sign\,(c)\left\lfloor \frac{|c|}{\Delta_b} \right\rfloor$$

El estándar JPEG2000 permite definir un *step-size* de cuantificación, Δ_b, especifico de cada subbanda. Para el camino reversible, el *step-size* de cuantificación debe ser necesariamente igual a uno. Por el camino irreversible la cuantificación recibe el nombre de *DZQ*, y por el reversible, el de *Ranging*.

- **Ajuste de la ROI (*Region Of Interest*):** Como se comentó anteriormente, JPEG2000 ofrece mecanismos mediante los cuales el com-

Figura 1.4: Transformada DWT a la imagen "lena".

presor puede asignar una prioridad más alta a ciertas regiones de la imagen. En esta etapa es donde se realiza la codificación de la ROI a tener en cuenta en la codificación. El método empleado en el estándar para la codificación de la ROI es el llamado método *MAXSHIFT*, que consiste en desplazar a la izquierda los coeficientes *wavelet* que conforman la ROI.

- **Codificación:** El codificador empleado por el estándar JPEG2000 es el codificador *MQ*, similar al codificador *QM* empleado en el estándar JPEG. El codificador *MQ* es empleado también en el estándar JBIG-2. Se trata de un codificador aritmético binario que permite codificar sin redundancia estadística los planos de bits de los coeficientes *wavelets*.

1.3. Particiones de los datos

El estándar JPEG2000 define una gran variedad de particiones dentro de los datos asociados a una imagen, con el fin de permitir la manipulación eficiente de la imagen, o de un trozo de ella, a diferentes resoluciones.

El objetivo de este capitulo es describir estas particiones, ya que a ellas se hará referencia en muchas partes de este documento.

1.3.1. El concepto de *canvas*

Para entender el concepto de cada una de las particiones que se definen en el estándar JPEG2000, es necesario tener claro el concepto de *canvas*. El *canvas* es una zona bidimensional de dibujo donde se mapearán todas las particiones que forman la imagen. Las coordenadas a las que se refiera de aquí en adelante son respecto al *canvas*, cuya dimensión, ancho y alto,

corresponderá a la dimensión total de la imagen a visualizar. Esto no conlleva necesariamente que la imagen comprimida ocupe la totalidad del *canvas*. Cada partición se sitúa y mapea sobre el *canvas* de una determinada forma.

1.3.2. Componentes

Una imagen está compuesta por una o más componentes. En nuestro caso, por ejemplo, las imágenes con las que vamos a trabajar van a ser imágenes con tres componentes, la componente rojo, la componente verde y la componente azul (RGB).

En el estándar JPEG2000, las componentes tienen asociados además unos factores de espaciado, o *sub-sampling factors*. Sea una componente cualquiera c, definida por un conjunto bidimensional de muestras, $x_c[n_1, n_2]$, se definen dos factores de espaciado, S_1^c y S_2^c, uno para las filas y otro para las columnas respectivamente. Cada muestra de la componente, $x_c[n_1, n_2]$ tiene la posición $[n_1 S_1^c, n_2 S_2^c]$ dentro del *canvas*.

Para una imagen simple RGB, las tres componentes de color tendrán una dimensión igual a la del *canvas*, y unos factores de espacio igual a uno.

1.3.3. *Tiles*

El estándar JPEG2000 permite dividir una imagen en regiones rectangulares más pequeñas llamadas *tiles* (que en español sería algo así como baldosas). Cada *tile* es comprimido de forma independiente en relación al resto, por lo que los parámetros de compresión pueden ser distintos para cada *tile* si se desea.

Una de las posibles aplicaciones de estas particiones son las imágenes que contienen elementos variados y bien diferenciados, como texto, gráficos o material fotográfico. El uso de los *tiles* también reduce la memoria necesaria para la descompresión de una imagen, al tener que aplicar el proceso de descompresión *tile* a *tile* en vez de a la imagen completa.

La forma de crear los *tiles* está definida por cuatro parámetros enteros positivos, que, como se verá más adelante, aparecerán en el marcador *SIZ*: los dos primeros, T_1, T_2, definen el alto y el ancho de los *tiles*, y los dos restantes Ω_1^T y Ω_2^T, definen el punto de anclaje del conjunto de *tiles*. Esto puede verse gráficamente en la figura 1.5.

Los *tiles* le dan al estándar JPEG2000 una gran versatilidad, pero en la mayoría de los casos, su uso no merecerá la pena e incluso, la calidad de la imagen puede verse afectada, como se puede apreciar en la figura 1.6.

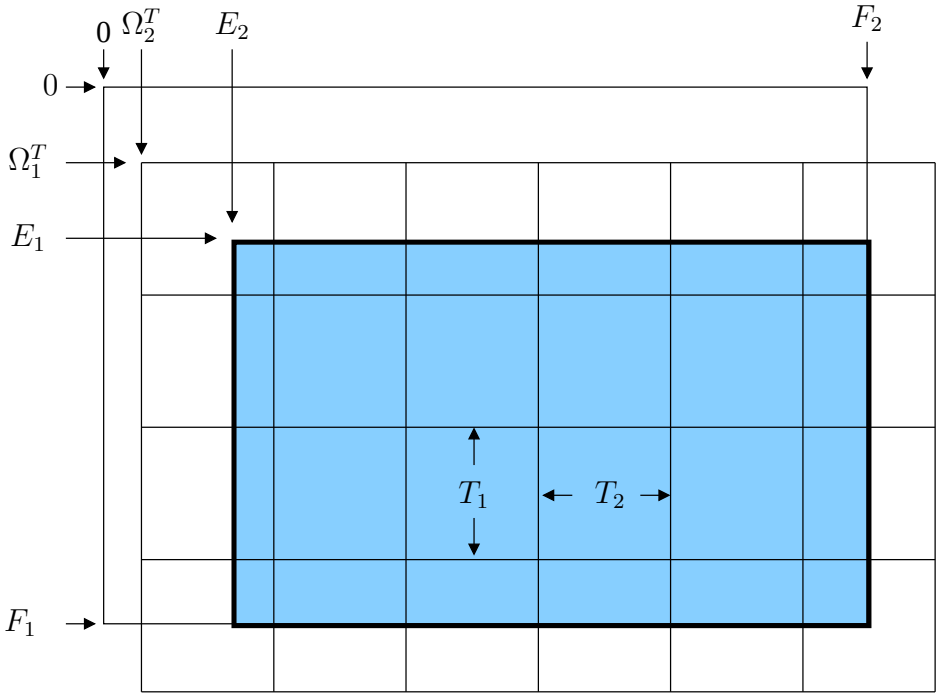

Figura 1.5: La partición en *tiles* sobre el *canvas*. La región en celeste identifica a la imagen, la región en la que está incluida al *canvas*, y los rectángulos grises los diferentes *tiles*.

Figura 1.6: La imagen "ski" de 720×576: (a) imagen original, (b)-(d) imágenes reconstruidas después de una compresión JPEG2000 a 0.25 bpp: (b) sin *tiling*, (c) con *tiles* de 128×128, y (d) con *tiles* de 64×64.

1.3.4. *Tile*-componente

La transformada DWT y todos los pasos siguientes de cuantificación y codificación son aplicados independientemente a cada *tile*-componente. Un *tile*-componente, del *tile* t y la componente c, está formando por la zona bidimensional que define t sobre la cual se mapea la componente c en función de sus factores de espaciado. Esto quiere decir que si tenemos una imagen con un único *tile* que ocupe el total de la imagen, con tres componentes de

color, tendremos tres *tile*-componentes, a los que a cada uno se le aplica el proceso de compresión de forma independiente.

1.3.5. Resoluciones

Para cada *tile*-componente, identificado por el *tile* t y la componente c, hay un total de $D_{t,c}+1$ resoluciones, donde $D_{t,c}$ es el número de etapas DWT aplicadas. El r-ésimo nivel de resolución de la imagen comprimida se obtiene tras aplicar r veces la transformada DWT directa o $D_{t,c}-r$ etapas de la transformada inversa, y se corresponde con la banda de frecuencia ll^r. El valor de r toma una valor en el rango $0 \leq r \leq D_{t,c}$. La menor resolución, $r=0$, corresponde a la última subbanda obtenida tras aplicar todos los pasos de la transformada DWT al *tile*-componente. La mayor resolución se corresponde al *tile*-componente original.

En la figura 1.4 se pueden apreciar las distintas resoluciones. En ese caso, $D_{t,c}=2$, con lo que existe un total de 3 resoluciones. La resolución mayor sería la primera imagen. Las restantes resoluciones se corresponden con la parte más nítidas de las restantes imágenes, de forma que, la resolución 0 correspondería con la pequeña zona cuadrada situada en la esquina superior izquierda de la tercera imagen.

Siendo $[E_1^t, F_1^t) \times [E_2^t, F_2^t)$ la región que ocupa el *tile* t en el *canvas*, la dimensión del *tile*-componente definido por el *tile* t y la componente c, en la resolución r es:

$$E_i^{t,c,r} = \left\lceil \frac{E_i^t}{2^{(D_{t,c}-r)} S_i^c} \right\rceil,\ F_i^{t,c,r} = \left\lceil \frac{F_i^t}{2^{(D_{t,c}-r)} S_i^c} \right\rceil, \text{ para } i \in 1,2$$

1.3.6. *Code-blocks* y precintos

Cada *tile*-componente es dividido en *code-blocks*, que son codificados independientemente. Esta partición en *code-blocks* es definida mediante el punto de anclaje, $[\Omega_1^C, \Omega_2^C]$, y el alto y el ancho máximo del *code-block*, $J_1^{t,c}$ y $J_2^{t,c}$. En la Parte 1 del estándar, el punto de anclaje debe ser obligatoriamente $[0,0]$ pero, en cambio, en la Parte 2, cada $\Omega_i^{t,c}$ puede tomar el valor de 0 o de 1.

La división en *code-blocks* es fundamental para el paradigma EBCOT [11], empleado en el estándar JPEG2000.

En cada resolución r de cada *tile*-componente t, c, los *code-blocks* son agrupados en precintos (*precincts*). Esta agrupación viene definida por el alto, en número de *code-blocks*, $P_1^{t,c,r}$, y el ancho, también en número de *code-blocks*, $P_2^{t,c,r}$ de cada precinto.

En la figura 1.7 se puede observar la división de una resolución de un *tile*-componente en *code-blocks* y precintos.

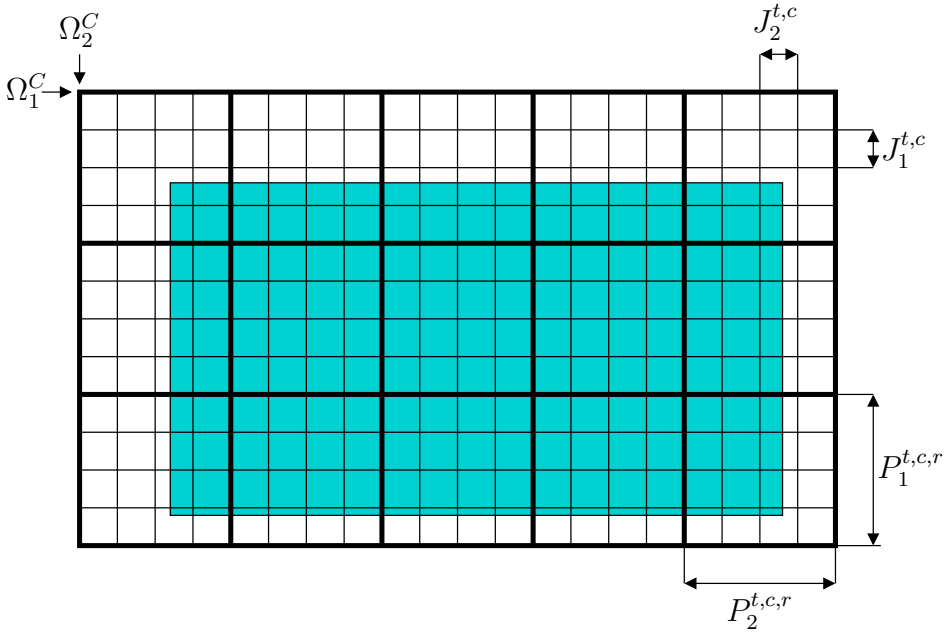

Figura 1.7: Partición en *code-blocks* y precintos de un *tile*-componente a una determinada resolución. La región en celeste identifica a la resolución del *tile*-componente.

Cada precinto P es identificado por un par de índices $[p_1, p_2]$, los cuales están comprendidos en los siguientes rangos:

$$0 \leq p_1 < N_1^{P,t,c,r} \text{ y } 0 \leq p_2 < N_2^{P,t,c,r}$$

$N_1^{P,t,c,r}$ y $N_2^{P,t,c,r}$ identifican el número de precintos vertical y horizontalmente, respectivamente. Estos valores son obtenidos mediante la siguiente ecuación:

$$N_i^{P,t,c,r} = \begin{cases} \left\lceil \frac{F_i^{t,c,r} - \Omega_i^C}{P_i^{t,c,r}} \right\rceil - \left\lfloor \frac{E_i^{t,c,r} - \Omega_i^C}{P_i^{t,c,r}} \right\rfloor & \text{si} \quad F_i^{t,c,r} > E_i^{t,c,r} \\ 0 & \text{si} \quad F_i^{t,c,r} = E_i^{t,c,r} \end{cases}$$

Los precintos juegan un papel importante en la organización de los datos comprimidos dentro del *codestream*.

1.3.7. Capas y paquetes

El paquete es la unidad fundamental en la organización del *codestream*. Cada precinto contribuye al *codestream* con tantos paquetes como capas de calidad (*quality layer*) haya. Como hemos mencionado anteriormente, cada *code-block* es codificado independientemente. Los datos codificados de cada *code-block* son divididos en diferentes secciones llamadas capas de calidad. Todos los *code-blocks* de los precintos de un mismo *tile* son divididos en el mismo número de capas, aunque el tamaño de las capas entre *code-blocks* puede ser diferente (el tamaño puede ser incluso cero). Para una capa l determinada, el conjunto de las capas l de todos los *code-blocks* que forman un precinto forman un paquete. La figura 1.8 muestra un ejemplo gráfico.

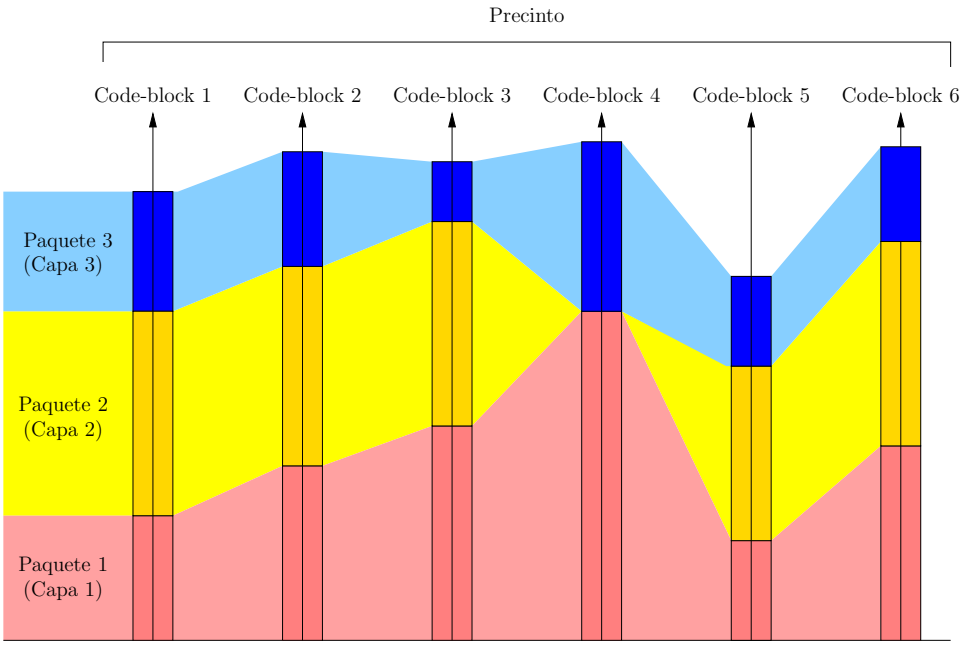

Figura 1.8: Ejemplo de capas de calidad y paquetes.

El número de capas de calidad para un *tile* t es identificado como Λ_t. Aunque el número de capas de calidad puede variar de *tile* a *tile*, los compresores harían bien en usar el mismo número de capas para todos los *tiles*, para evitar ambigüedades cuando se pretende descartar un determinado número de capas de calidad en una imagen.

El número total de paquetes en un *codestream* sería:

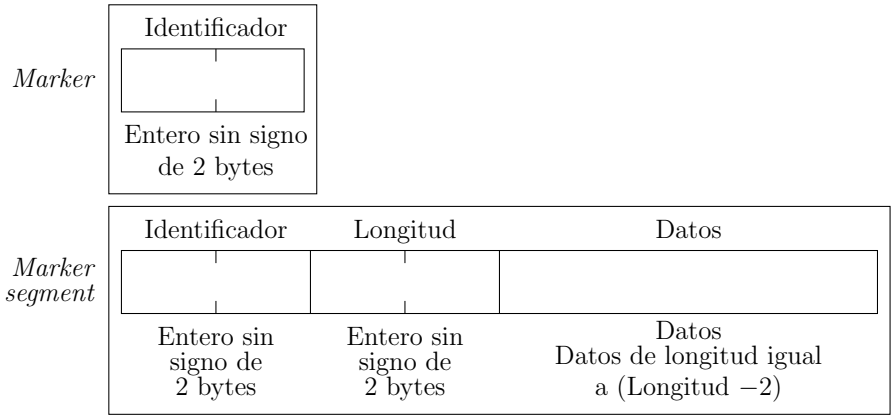

Figura 1.9: Tipos de marcadores.

$$\text{número de paquetes} = \sum_{t_1=0}^{N_1^T-1} \sum_{t_2=0}^{N_2^T-1} \sum_{c=0}^{C-1} \sum_{r=0}^{D_{t,c}} \Lambda_t N_1^{P,t,c,r} N_2^{P,t,c,r}$$

Un paquete $\zeta_{t,c,r,p,l}$ es identificado por el *tile* t, la componente c, la resolución r, el precinto p ($p \equiv [p_1, p_2]$) y la capa de calidad l.

1.4. Formato del *codestream*

A continuación se explicará el formato del *codestream* según la Parte 1 del estándar JPEG2000.

1.4.1. Marcadores

El *codestream* está basado en marcadores (*markers*), los cuales poseen un identificador único, en forma de entero sin signo de 16 bits. Estos marcadores pueden presentarse solos, es decir, únicamente el identificador, o acompañados de información adicional, recibiendo entonces el nombre de segmento marcador (*marker segment*), como aparece en la figura 1.9.

El segmento marcador posee, a continuación del identificador, otro entero sin signo de 16 bits que posee la longitud de los datos incluidos en el marcador, incluyendo los dos bytes de dicho entero, pero sin contar los dos bytes del identificador.

En la tabla 1.2 pueden verse todos los marcadores disponibles en la Parte 1 del estándar. No entraremos a explicar todos los marcadores, sino que nos limitaremos a los marcadores que nos interesan. Para una mayor información, consultar [14].

1.4.2. Organización del *codestream*

El *codestream* comienza siempre por el marcador SOC (*Start Of Codestream*), el cual no lleva información ninguna. A partir de este marcador comienza lo que se denomina la cabecera principal. Los marcadores que pueden aparecer en esta cabecera aparecen en la tabla 1.2.

Al marcador SOC le sigue siempre el marcador SIZ, que posee información global necesaria para la descompresión de los datos, como la dimensión de la imagen, la dimensión de los *tiles*, el punto de anclaje de la partición de los *tiles*, el número de componentes, sus factores de espaciado, etc.

Como se puede puede ver en la tabla 1.2, otros dos marcadores necesarios en la cabecera principal son: COD, que contiene información relativa a la codificación de la imagen, como el número de capas, el número etapas de la transformada DWT, el tamaño de los *code-blocks*, la progresión, etc.; QCD, que contiene los parámetros de cuantificación. Estos dos marcadores son necesarios, pero pueden aparecer en cualquier parte de la cabecera principal.

El resto del *codestream*, hasta el marcador EOC (*End Of Codestream*), situado justo al final del mismo, se organiza tal y como aparece en la figura 1.10. Para cada *tile* de la imagen, tenemos un conjunto de datos. Este conjunto de datos se divide en uno o más *tile-part*. Cada *tile-part* se compone por una cabecera y un conjunto de paquetes. La cabecera del primer *tile-part* es la cabecera principal del *tile*. La cabecera de cada *tile-part* comienza con el marcador SOT (*Start Of Tile*) y finaliza con el marcador SOD (*Start Of Data*), a partir del cual comienza la secuencia de paquetes, según el último marcador COD o POC. Los marcadores admitidos por cada cabecera aparecen en la tabla 1.2. La cabecera principal finaliza al encontrar el primer SOT.

Para permitir el acceso aleatorio a los datos de un *codestream*, JPEG2000 ofrece la posibilidad de incluir los marcadores TLM, PLM y/o PLT. Los marcadores TLM y PLM se incluyen en la cabecera principal, mientras que el marcador PLT va en la cabecera de un *tile* o *tile-part*. La misión del marcador TLM es almacenar la longitud de cada *tile-part* que aparece en el *codestream*, aunque esta longitud incluye tanto la cabecera como la secuencia de paquetes, por lo que para saber donde comienza la secuencia de paquetes habría que procesar la cabecera primero. El marcador PLM almacena la

Nombre	Identificador	Descripción	Posición	¿Datos?	¿Obligatorio?
SOC	0xFF4F	Comienzo del *codestream*	M	No	Si
SOT	0xFF90	Comienzo de *tile* (o *tile-part*)	T,P	Si	Si
SOD	0xFF93	Comienzo de los datos de un *tile* (o *tile-part*)	T,P	No	Si
EOC	0xFFD9	Fin del *codestream*	E	No	Si
SIZ	0xFF51	Tamaño de la imagen y de los *tiles*	M	Si	Si
COD	0xFF52	Parámetros de codificación	M,T	Si	Si
COC	0xFF53	Parámetros de codificación para un componente	M,T	Si	No
QCD	0xFF5C	Parámetros de cuantificación	M,T	Si	Si
QCC	0xFF5D	Parámetros de cuantificación para un componente	M,T	Si	No
RGN	0xFF5E	Región de interés (ROI)	M,T	Si	No
POC	0xFF5F	Cambio de tipo de progresión	M,T,P	Si	No
TLM	0xFF55	Longitudes de todos los *tile-parts* del *codestream*	M	Si	No
PLM	0xFF57	Longitudes de todos los paquetes del *codestream*	M	Si	No
PLT	0xFF58	Longitudes de todos los paquetes de un *tile-part*	T,P	Si	No
PPM	0xFF60	Cabeceras de todos los paquetes del *codestream*	M	Si	No
PPT	0xFF61	Cabeceras de todos los paquetes de un *tile-part*	T,P	Si	No
SOP	0xFF91	Comienzo de paquete	S	Si	No
EPH	0xFF92	Fin de la cabecera del paquete	S	Si	No
CRG	0xFF63	Registro de un componente	M	Si	No
COM	0xFF64	Comentario	M,T,P	Si	No

Tabla 1.2: Marcadores de un *codestream* JPEG2000. Las posiciones donde los marcadores pueden aparecer son identificadas por: M, para la cabecera principal; T, para la cabecera de un *tile*; P, para la cabecera de un *tile-part*; S, cuando aparece dentro de los datos comprimidos de un *tile-part*; E: cuando aparece al final del *codestream*.

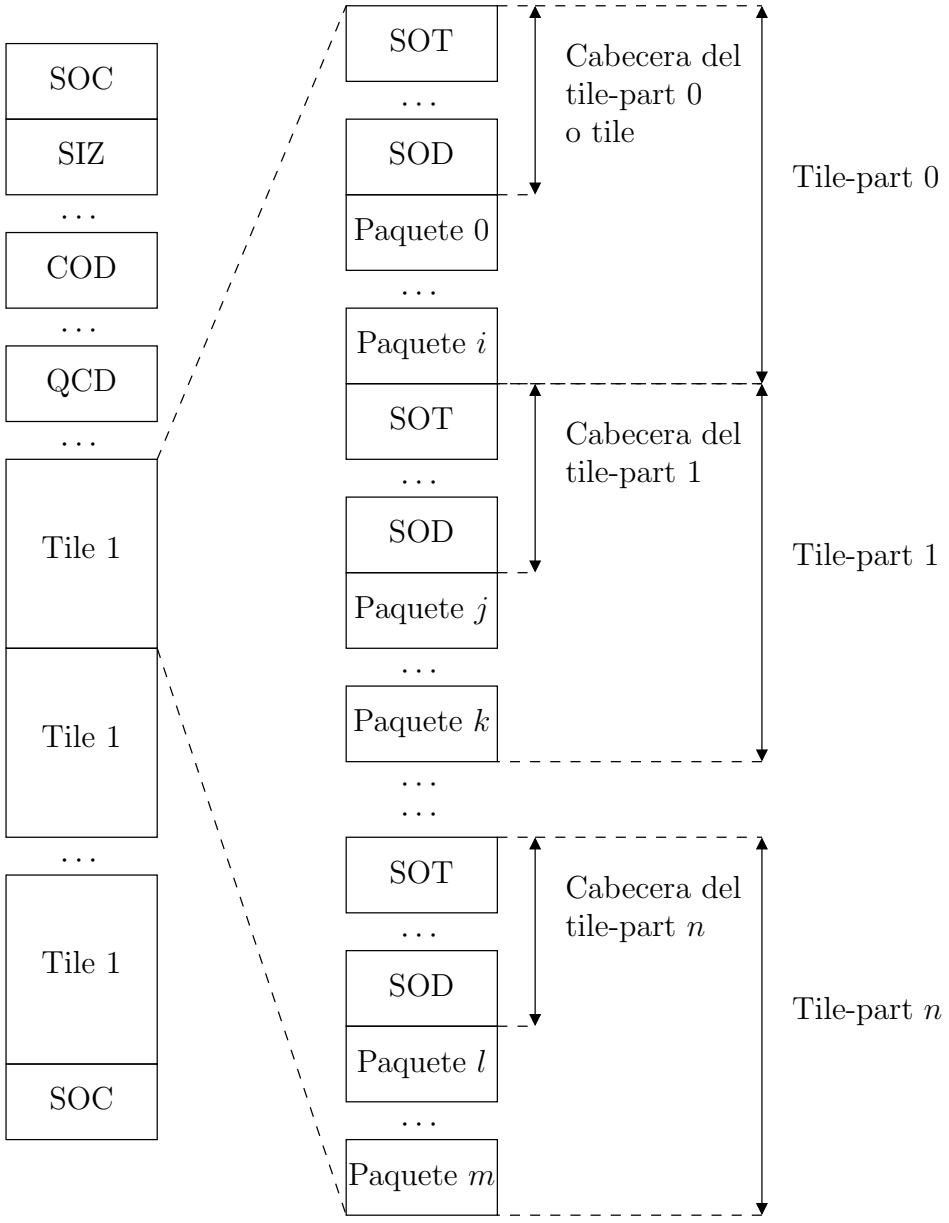

Figura 1.10: Organización del *codestream*.

longitud de cada paquete de cada *tile-part* del *codestream*. Cada paquete del *codestream* posee una longitud determinada, que no puede calcularse de antemano, por lo que la inclusión de este marcador facilita el acceso aleatorio a los paquetes. El marcador PLT cumple la misma función que el marcador PLM, pero a nivel de *tile-part*, es decir, almacena la longitud de todos los paquetes del *tile-part* al cual pertenece.

Los marcadores PLM y PLT producen una dilatación del tamaño del *codestream*, aunque la forma de almacenar las longitudes de los paquetes ayuda a evitar que resulte excesiva: una longitud L de un paquete determinado, que puede representarse con B_L bits, se almacena, tanto en un marcador PLM como en uno PLT, con $\left\lceil \frac{B_L}{7} \right\rceil$. Esto quiere decir que para una longitud de 8 bits, necesitamos dos bytes. A primera vista no parece una gran ayuda, pero hay que considerar que almacenamos las longitudes ocupando el menor número de bytes posibles, porque, por ejemplo, una longitud representable con 17 bits, no ocuparía 4 bytes como sería lo normal (lo normal es almacenar los enteros en 1 byte, en 2 bytes o en 4 bytes), sino 3 bytes.

Los marcadores SOP y EPH pueden ir intercalados entre los paquetes. Si se utiliza el marcador SOP, éste aparece al comienzo de cada paquete, y puede servir para detectar posibles errores en la secuencia de los paquetes: por ejemplo, en los marcadores PLM y PLT, la longitud de los paquetes incluye al marcador SOP, por lo que, si empleáramos estos marcadores para acceder a paquetes aleatoriamente, podríamos comprobar que se ha accedido correctamente al paquete si los primeros bytes del mismo se corresponden con el marcador SOP apropiado. El marcador SOP almacena el índice del paquete al cual se asocia, en un entero sin signo de dos bytes. Si se utiliza el marcador EPH, éste aparece al final de cada paquete y no contiene información alguna. Su función es únicamente delimitar el paquete. En el marcador COD se especifica de forma independiente si aparece o no cada uno de estos marcadores, SOP y EPH.

1.4.3. Progresiones

Los paquetes de cada *tile-part* aparecen según la progresión indicada en el último marcador COD o POC leído antes del marcador SOD, a partir del cual comienza la secuencia de paquetes.

Según la Parte 1 del estándar JPEG2000, existen 5 posibles progresiones:

- **Progresión LRCP (*Layer-Resolution-Component-Position*):**

  para cada capa l

Figura 1.11: Ejemplo de progresión en calidad.

Figura 1.12: Ejemplo de progresión en resolución.

para cada resolución r
    para cada componente c
        para cada precinto p
            incluir el paquete como $\zeta_{t,c,r,p,l}$

Como la progresión puede cambiar de *tile* a *tile*, la variable t identifica el *tile* al cual pertenece la secuencia de paquetes. Esta progresión es una progresión en calidad, de manera que hasta que no son leídos todos los paquetes del *tile*, para una cierta capa de calidad, no se pasa a la siguiente capa. Un ejemplo de progresión en calidad puede verse en la figura 1.11.

- **Progresión RLCP (*Resolution-Layer-Component-Position*):**

para cada resolución r
    para cada capa l
        para cada componente c
            para cada precinto p
                incluir el paquete como $\zeta_{t,c,r,p,l}$

Es una progresión en resolución, ya que se van procesando todos los paquetes de una resolución a otra, desde la resolución menor hasta la mayor, incluyendo todas las capas de calidad. Un ejemplo de progresión en resolución puede verse en la figura 1.12.

- **Progresión RPCL (*Resolution-Position-Component-Layer*):**

para cada resolución r
    para cada precinto p
        para cada componente c
            para cada capa l
                incluir el paquete como $\zeta_{t,c,r,p,l}$

Figura 1.13: Ejemplo de progresión en posición.

Al igual que la anterior, ésta es una progresión en resolución también. La diferencia con la anterior es que, en cada resolución, aquí la lectura es de precinto a precinto, mientras que en la anterior es por capas de calidad.

- **Progresión PCRL (*Position-Component-Resolution-Layer*):**

para cada precinto p
    para cada componente c
        para cada resolución r
            para cada capa l
                incluir el paquete como $\zeta_{t,c,r,p,l}$

Esta es una progresión por posición, de forma que se va procesando precinto a precinto, empezando por el situado en la esquina superior izquierda hasta el situado en la esquina inferior derecha. Un ejemplo de progresión por posición lo tenemos en la figura 1.13.

- **Progresión CPRL (*Component-Position-Resolution-Layer*):**

para cada componente c
    para cada precinto p
        para cada resolución r
            para cada capa l
                incluir el paquete como $\zeta_{t,c,r,p,l}$

La última progresión que admite el estándar es una progresión por componente, de manera que hasta no haber leído y procesado todos los paquetes de una componente, no se pasaría al siguiente. Un ejemplo de progresión por componente lo podemos ver en la figura 1.14.

Figura 1.14: Ejemplo de progresión por componente.

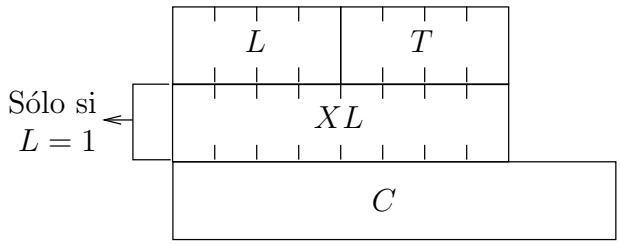

Figura 1.15: Estructura de una caja JP2.

1.5. Formato de los archivos JP2

El *codestream* de una imagen podría ser perfectamente almacenado directamente en un archivo, sin añadir nada más, y funcionar como archivo de imagen, ya que el *codestream* posee toda la información necesaria para la reconstrucción de la imagen. De hecho, los archivos .J2C son así, contienen un único *codestream*, sin más. No obstante, numerosas aplicaciones necesitan añadir otro tipo de información útil, como por ejemplo paletas de color, *copyright*, etc. La Parte 1 del estándar define un tipo de archivo, con extensión .JP2, que permite incluir en un mismo fichero varios *codestream* e información extra.

Los archivos JP2 están organizados en cajas (*boxes*). La estructura de una caja puede verse en la figura 1.15.

Los primeros cuatro bytes de la caja contienen la longitud de la misma, L, en forma de entero sin signo. Los siguientes cuatro bytes forman una cadena de cuatro caracteres ASCII que contienen el tipo de caja, T. Si $L = 1$, la longitud real de la caja viene dada por los 8 bytes siguientes a T, que forman un entero sin signo, XL. Si $T \neq 1$ entonces el parámetro XL no aparece. A continuación están los datos de la caja C. Si $L \neq 1$ la longitud de C es $L - 8$. Si $L = 8$ la longitud de C es $XL - 16$.

Las cajas pueden contener a su vez otras cajas. Sabremos que una caja contiene o no subcajas en función del tipo, T, de caja que sea. Si una caja contiene subcajas, sólo puede contener subcajas, es decir, no puede combinar cajas con otro tipo de datos.

Al igual que ocurría con los marcadores del *codestream*, existen algunas

cajas cuya presencia es obligatoria en el archivo JP2. Estas cajas son: la caja *JPEG2000 Signature*, que debe ser única y cuyo cometido es únicamente marcar el comienzo del archivo y detectar posibles errores de la transmisión; la caja *File type*, que posee la lista de formatos a los que pertenece el archivo, ya que existen otros además del JP2; la caja *JP2 Header*, que contiene a su vez varias cajas con diversa información relativa a la imagen, algunas de las cuales, resultan un poco redundantes ya que la información que poseen se haya también en el *codestream*; la caja *Contiguous Code-stream box*, cuyo contenido es el *codestream* de la imagen. La organización de las cajas en un archivo JP2, así como las posibles cajas existentes se reflejan en la figura 1.16.

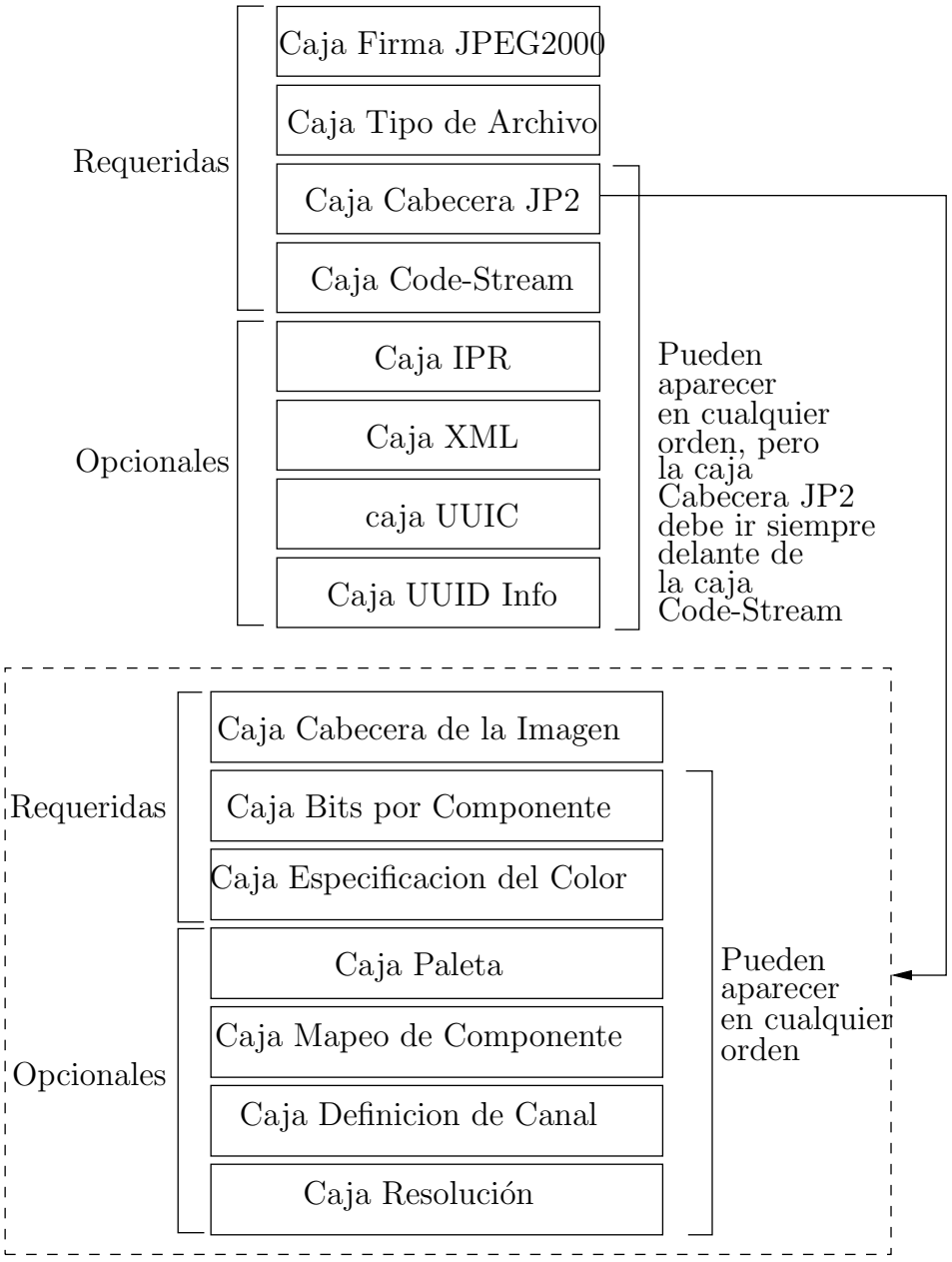

Figura 1.16: Cajas de un archivo JP2.

Capítulo 2

El protocolo JPIP

2.1. Introducción

El protocolo JPIP [7, 13, 15] es un protocolo para la interacción con imágenes JPEG2000 remotas, desarrollado por el ISO/IEC *Joint Technical Commitee of Photographic Experts* (JPEG), que forma parte de la Parte 9 del estándar JPEG2000. El protocolo JPIP es una evolución del protocolo JPIK, desarrollado por David Taubman [10].

2.2. Arquitectura

El protocolo JPIP se basa en una arquitectura típica cliente/servidor. La parte del servidor es en la que se encuentra el servidor JPIP y la colección de archivos de imágenes JPEG2000 (*raw* .J2C, .JP2, .JPX, etc.) a la cual tendrá acceso el cliente. En la parte del cliente tendremos la aplicación, que generalmente será un visualizador interactivo de imágenes remotas, que solicitará información al servidor. En la figura 2.1 podemos ver la arquitectura básica para el protocolo JPIP.

Cuando el cliente desea visualizar una región determinada, a una resolución determinada, de una imagen determinada, no precisa conocer en qué lugar del archivo remoto de la imagen se encuentra la información necesaria para la reconstrucción de dicha región, sino que simplemente realiza una petición al servidor indicándole el nombre de la imagen, la resolución y la región de interés. El servidor es el encargado entonces de leer la información necesaria del archivo de la imagen para reconstruir la región solicitada. Además de la región de interés y la resolución, el cliente puede añadir más requerimientos en sus peticiones al servidor, como podría ser el número de

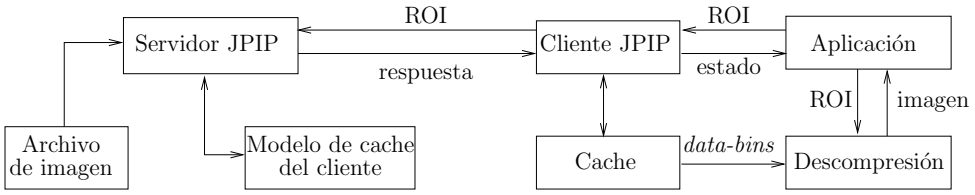

Figura 2.1: Arquitectura del protocolo JPIP.

capas de calidad, el número de componentes, etc.

El cliente contendrá una *cache* en donde va almacenando los datos que el servidor le envía, organizados en *data-bins*, cuyo concepto es analizado más adelante. Esto permite el poder comenzar la visualización de una determinada región si de ella se conservan datos de anteriores peticiones al servidor. El servidor puede de forma opcional mantener el modelo de *cache* del cliente para evitar transmitirle datos que ya le ha transmitido anteriormente.

En el cliente, el proceso de descompresión/visualización (la parte *Aplicación* y *Descompresión* de la figura 2.1) está separado del proceso de comunicación (la parte *Cliente JPIP* de la figura 2.1). Ambos procesos se ejecutarán paralelamente, de forma que mientras uno, el de comunicación, esté realizando la petición al servidor de la región de interés y recibiendo los *data-bins* apropiados y almacenándolos en la *cache*, el otro, el de visualización, puede ir recogiendo los *data-bins* de la *cache* relacionados con la región a visualizar y reconstruyendo la imagen. El proceso de visualización comenzaría cuando el usuario solicita una nueva región de la imagen y finalizaría cuando el proceso de comunicación termina de leer todos los *data-bins* asociados. El proceso de visualización estaría continuamente reconstruyendo la imagen una y otra vez y no debe importarle si no están todos los *data-bins* necesarios para ello (si alguna porción no es posible reconstruirla, se podría rellenar de algún color, por ejemplo).

El protocolo JPIP ha sido diseñado para ser independiente del protocolo de transporte finalmente utilizado. De esta forma, aunque inicialmente fue concebido para usar HTTP/1.1, también es capaz de correr sobre UDP y TCP.

Actualmente el protocolo JPIP sólo ha sido implementado para emplear el protocolo HTTP solamente, o el protocolo HTTP en asociación con el protocolo TCP. En este último caso, el funcionamiento es similar al primero, pero los *data-bins* son enviados por el servidor al cliente a través de una conexión en TCP simple separada de la conexión HTTP.

2.2.1. Concepto de *data-bin*

La información almacenada en un archivo de imagen, tanto un archivo *raw* .J2C, como otro más complejo como .JP2 o .JPX, es dividida bajo el protocolo JPIP en bloques llamados *data-bins*. Cada *data-bin* es identificado por dos valores: el tipo de *data-bin* y el identificador, que identifica de manera única al *data-bin* dentro del grupo al cual pertenece en función del tipo.

Los diferentes *data-bins* con los que trabaja el protocolo JPIP son los siguientes:

- **Precinto (*Precinct data-bins*):** Como su propio nombre indica, este *data-bin* contiene la información de un precinto, a una resolución r, de una componente c, de un *tile* t. Esta está formada por todos los paquetes de un precinto, ordenados por capa de calidad, desde la menor hasta la mayor. El identificador I es obtenido mediante la siguiente formula:

$$I = t + ((c + (s \times numComponentes)) \times numTiles)$$

  Tanto el índice de *tile*, t, como el índice de componente c, y el valor s, comienzan en cero. A cada precinto de cada *tile*-componente, incluyendo todas las resoluciones, le es asignado un único número de secuencia s. A todos los precintos de la resolución más baja le son asignados en primer lugar un número de secuencia que empezaría con el precinto situado en la esquina superior izquierda e iría avanzando por columna y por fila. En las resoluciones siguientes, a los precintos se les va asignando un número de secuencia correlativo de la misma forma.

- **Cabecera (*Header data-bins*):** Un *data-bin* de cabecera puede ser tanto la cabecera principal, como la cabecera de un *tile*. Si el identificador es igual a 0, el *data-bin* contiene la cabecera principal del *code-stream*, desde el marcador SOC (incluido) hasta el primer marcador SOT (no incluido). Si el identificador es igual a n, donde $n \geq 1$, el *data-bin* contiene la cabecera del *tile* $n - 1$, pero sin incluir ni el marcador SOT ni el marcador SOD. Bajo el protocolo JPIP no existen *tile-parts*, por lo que si en un archivo de imagen los *tiles* está divididos en *tile-parts*, el servidor JPIP, cuando vaya a enviar el *data-bin* de la cabecera de un *tile* determinado, deberá unir todas las cabeceras de todos los *tile-parts* que forman dicho *tile* en una sola.

- **Meta información (*Meta data-bins*):** Estos *data-bins* son opcionales y su existencia dependerá de si el archivo de imagen es un archivo de la familia .JP2 (.JPX, .JPM, etc.) o no. En el caso de así sea, cada *data-bin* contendrá un conjunto de cajas *boxes* (Sección 1.5). El servidor elegirá el número de cajas a incluir en un *data-bin*, teniendo presente que todos los *data-bins* de una imagen deben tener el mismo número de cajas. El valor del identificador no es relevante salvo en el caso de ser igual a cero, en cuyo caso el *data-bin* contiene el conjunto de cajas esenciales de la imagen, que en el caso de ser un archivo .JP2 éstas serían la caja JPEG2000 Firma (*JPEG2000 signature box*), la caja Tipo de Archivo (*File Type box*) y la caja Cabecera JP2 (*JP2 Header box*), con todas sus subcajas asociadas.

- **Catálogo (*Catalog data-bins*):** Un cliente que recibe uno o más *data-bins* con meta información, puede analizar su contenido para averiguar qué cajas contiene. Sin embargo, en algunos caso, la interpretación de esas cajas puede depender de su posición con respecto a otras cajas que podrían aparecer almacenados en otros *data-bins*, algo que no se puede conocer de antemano. Con el fin de solucionar este problema se crean los *data-bins* de catálogo, que contienen como información la estructura de cajas de un archivo determinado. No contiene ningún dato de ninguna caja. Cada caja representada en dicha estructura posee un índice mediante el cual encontrar la información asociada a la misma (en que *data-bin* de meta información se encuentra qué tamaño ocupa y en que posición).

2.2.2. Sesiones y canales

Una petición del cliente al servidor puede ser con estado (*stateful*) o sin estado (*stateless*). Las peticiones con estado son realizadas en el contexto de una sesión de comunicación, cuyo estado es mantenido por el servidor. Las peticiones sin estado no requieren ninguna sesión. El uso de sesiones mejora el rendimiento del servidor, ya que, por ejemplo, al establecer una sesión con un archivo de imagen determinado, el servidor abre dicho archivo y lo prepara para ser transmitido en *data-bins*, con lo que todas las peticiones bajo esa misma sesión no provocan que el servidor realice nuevamente este proceso, como ocurriría con peticiones sin estado (o peticiones asociadas a otras sesiones). Las peticiones sin estado pueden considerarse bajo una sesión única que termina cuando finaliza la respuesta a la petición.

Bajo una sesión, con peticiones con estado, el cliente puede abrir múlti-

ples canales, pudiendo realizar múltiples peticiones con estado asociadas a la misma sesión concurrentemente. Esto es especialmente útil para aplicaciones que visualizan simultáneamente diversas regiones de interés de una misma imagen. Los canales se pueden cerrar y abrir otros de forma independiente y sin afectar a la sesión. El cerrar una sesión supondría cerrar todos los canales abiertos asociados a ella.

2.2.3. *Caching*

Como vimos anteriormente, la arquitectura del protocolo JPIP aconseja una *cache* para el cliente (aunque no es estrictamente necesaria). En ella el cliente va almacenando los distintos *data-bins* recibidos del servidor, y es utilizada para la descompresión de la imagen.

Vimos también que el servidor puede opcionalmente mantener un modelo de *cache* del cliente. Los modelos se crean por cada sesión que se abra. Este modelo consistiría en almacenar el rango de bytes de cada *data-bin* enviado al cliente. El modelo se desecha una vez cerrada la sesión asociada.

Los clientes pueden rehusar su *cache* entre diferentes sesiones. Incluso los clientes que realicen sólo peticiones sin estado, al ser consideradas éstas como peticiones bajo una sesión única que se cierra al terminar la respuesta del servidor, podrían mantener la misma *cache* para todas sus peticiones. Esto provoca que el servidor no tenga un modelo de *cache* que se corresponda con el contenido de la *cache* del cliente. El protocolo JPIP permite al cliente en sus peticiones al servidor especificar el contenido de su *cache*, parcial o totalmente, con el fin de que el servidor actualice su modelo. De esta forma, si el cliente mantiene su *cache* entre sesiones, ya sean con peticiones con o sin estado, podría enviarle su contenido al servidor con el fin de actualizar el modelo. Esto no es obligatorio, pero sí recomendable para un rendimiento óptimo. Cuando sí puede resultar obligatorio es cuando, bajo una sesión, en la cual el servidor mantiene el modelo de *cache* del cliente, el cliente, por el motivo que sea (por ejemplo, falta de memoria), descarta *data-bins* de la *cache*. Si posteriormente estos *data-bins* son requeridos, bajo la misma sesión, mediante una petición al servidor, éste no se los enviará al cliente, ya que para él ya se los ha enviado. En este caso sería imprescindible que el cliente le notificará al servidor los cambios producidos en su *cache*.

2.2.4. Peticiones

Las peticiones JPIP consisten en una secuencia de pares “parámetro = valor”. Esta secuencia es indicada como una cadena ASCII. Esto permite

que una petición JPIP pueda ser encapsulada dentro de una consulta GET del protocolo HTTP, a continuación del carácter '?', concatenando todos los pares con el símbolo '&'.

Algunos de los parámetros disponibles para una petición JPIP son los siguientes:

- **"fsiz=R_x,R_y":** Especifica la resolución asociada a la región de interés solicitada. El servidor escogerá la más pequeña resolución de la imagen tal que la dimensión de la misma, R'_x y R'_y cumpla que $R'_x \geq R_x$ y $R'_y \geq R_y$.

- **"roff=P_x,P_y":** Especifica la posición de la esquina superior izquierda de la región de interés requerida dentro de la resolución solicitada. Si esta posición no es indicada, el servidor asumirá que es $0, 0$.

- **"rsiz=S_x,S_y":** Especifica el tamaño de la región de interés requerida. El servidor cortará este tamaño para ajustarlo a la imagen real según la resolución especificada.

- **"comps=C_0, C_1, ..., C_N":** Indica las componentes que el servidor debe enviar de la región de interés. Las demás componentes serán ignoradas. Si este parámetro es ignorado, el servidor envía todas las componentes existentes.

- **"len=número de bytes":** El cliente le indica con este parámetro al servidor el número máximo de bytes máximo que debe enviar en la respuesta. El servidor tendrá en cuenta este límite, no sólo en la respuesta a la petición actual, sino en las restantes respuestas al resto de las peticiones que el cliente haga bajo la misma sesión.

- **"layers=L":** Con este parámetro el cliente puede limitar el número de capas de calidad a enviar por el servidor en la respuesta.

- **"target=imagen":** Este parámetro identifica al archivo de imagen del cual extraer la región de interés. Cuando se emplea el servidor actúa como servidor HTTP, este parámetro no es necesario ya que el nombre de la imagen se obtiene de la propia URL indicada en el comando GET.

- **"cnew=transporte":** Cuando el cliente desee abrir un nuevo canal bajo la misma sesión, empleará este parámetro, en el cual se incluye el protocolo que se debe utilizar para el nuevo canal. Los tipos de transporte permitidos hasta el momento son "httpƨ "http-tcp".

- **"cid=identificador de canal":** Al crear el cliente un nuevo canal, el servidor le envía el identificador del canal, que debe ser incluido en todas las peticiones asociadas a ese canal.
- **"model=..."**: Como se ha comentado, el cliente puede necesitar comunicarle al servidor el contenido de su *cache* para que éste actualice el contenido del modelo que mantiene. Esto lo puede hacer con este parámetro. Un ejemplo sería el siguiente:

  ```
  model=Hm,H*,M0,P0:20,-P1001
  ```

  con lo que le estaríamos indicando al servidor que incluyese en el modelo de *cache* que posee la cabecera principal del *codestream*, las cabeceras de todos los *tiles*, el *data-bin* de meta información número cero (el principal), que incluya los primeros 20 bytes del precinto 0 y que elimine el precinto número 1001.
- **"quality=calidad":** El cliente le indica la servidor cual es el máximo de calidad de la imagen a transmitir (desde 0 a 100). El servidor puede ignorar este parámetro.

Un ejemplo de una petición JPIP usando el protocolo HTTP/1.1 sería la siguiente:

```
GET imagen.jp2?fsiz=512,640&roff=0,0&fsiz=512,478&
              comps=0,1,2&len=2000 HTTP/1.1
Host: www.servidor.com
```

En ella estamos requiriendo las tres primeras componentes de una región de interés de tamaño (512x478) cuya esquina superior izquierda está situada en la posición (0,0) de la mínima resolución de la imagen *imagen.jp2* en la que quepa una región de tamaño (512x640). También se limita la máxima longitud de la respuesta del servidor en 2000 bytes.

2.2.5. Respuestas

Las respuestas del servidor bajo el protocolo JPIP son similares a las respuestas de los servidor HTTP: una primera parte ASCII que contiene un conjunto de cabeceras, y a continuación los datos de la respuesta en forma de *data-bins*.

Algunas de las cabeceras que puede enviar el servidor al cliente son las siguientes:

- **"JPIP-fsiz: R'_x, R'_y":** El servidor le envía el tamaño de la resolución al cliente, que puede ser diferente del requerido por éste.
- **"JPIP-rsiz: S'_x, S'_y":** El servidor le envía el tamaño de la región de interés al cliente, que puede ser diferente del requerido por éste.
- **"JPIP-roff: P'_x, P'_y":** El servidor le envía la posición de la región de interés al cliente, que puede ser diferente de la requerida por éste.
- **"JPIP-comps: C'_0, C'_1, ..., C'_N":** El servidor le envía al cliente las componentes que forman parte de la región de interés enviada, que pueden no ser las mismas que fueron requeridas por el cliente.
- **"JPIP-layers: L'":** El servidor le envía al cliente el número de capas de calidad incluidas en la región de interés enviada, que puede no ser el mismo número requerido por el cliente.
- **"JPIP-cnew: cid=...":** Cuando el cliente solicita la apertura de un canal nuevo, el servidor le envía el identificador de canal *cid* y una serie de información relativa al canal, como el puerto y el *host* a los que tiene que dirigirse el cliente para las peticiones a través de este canal, el protocolo empleado para el canal, etc.

Como se puede ver, el servidor tiene total libertad para alterar los parámetros enviados por el cliente. Las razones por las que el servidor cambie los parámetros especificados por el cliente en una petición pueden ser varias: puede ocurrir que el responder fielmente a lo requerido por el cliente requiriera un uso excesivo de recursos por parte del servidor, en cuyo caso el servidor puede limitar la respuesta, alterándola con el fin de optimizar el uso de los recursos; también la petición del cliente puede que no sea válida, con lo que el servidor podría modificarla para poder tratarla. El cliente debe de estar preparado para aceptar respuestas del servidor que no concuerden con sus peticiones.

A continuación podemos ver un ejemplo completo de petición y respuesta, con el protocolo JPIP empleando al protocolo HTTP como base:

```
Servidor ⟷ Cliente

⟵ GET imagen.jp2&fsiz=12000,8000&rsiz=600,400&roff=5000,6000 HTTP/1.1
⟵ Host: jpip.servidor
⟵
```

```
⟶ HTTP/1.1 200 OK
⟶ JPIP-fsiz: 6000,8000
⟶ JPIP-rsiz: 300,200
⟶ JPIP-roff: 2500,3000
⟶ Content-type: image/jpp-stream
⟶ (Resto de cabeceras...)
⟶
⟶ (Data-bins de la respuesta...)
```

En este ejemplo el cliente solicita una región de interés de tamaño 600x400 y cuya esquina superior izquierda esté situada en la posición (5000,6000) de la resolución mínima de la imagen *imagen.jp2* encuadrada por un área de 12000x8000. El servidor acepta la petición, pero la respuesta es ligeramente diferente a la esperada por el cliente: la región devuelta tendrá un tamaño de 300x200 y su esquina superior izquierda se situará en la posición (2500x3000) de una resolución de 6000x8000.

2.3. Indexado de archivos JP2

Si bien el protocolo JPIP es el núcleo de la Parte 9 del estándar JPEG2000, el grupo ISO/IEC JPEG no podía dar una solución tan cerrada a la interacción de imágenes remotas; si fuese así, cualquier aplicación que se quisiera explotar las estupendas cualidades del nuevo formato de compresión para la visualización remota de imágenes, necesitaría obligatoriamente un servidor, cuya implementación no es poco compleja y su implantación no siempre es posible. Por ello comprendió que debía ofrecer una solución parcial [7], a parte del protocolo, para que quien quisiera pudiese simular la funcionalidad del protocolo JPIP, sin necesidad del servidor, utilizando solamente la posibilidad que ofrece el tan difundido protocolo HTPP/1.1 para acceder a partes de un archivo llamada *byte-ranging*, que se verá más adelante.

Esta solución consiste en la definición de un conjunto de cajas para el formato de archivo definido por el estándar JPEG2000 (.JP2, .JPX, etc.) con las que poder indexar el archivo de imagen, con el fin de que el cliente, leyendo estas cajas, sepa en qué posición está cada *data-bin* y qué tamaño tiene, de forma que puede leerlos él directamente, sin necesidad de servidor alguno.

Para cada *codestream* que exista dentro de un archivo de imagen, por ejemplo, un archivo .JP2, se incluiría una caja *Codestream Index*, cuyo tipo

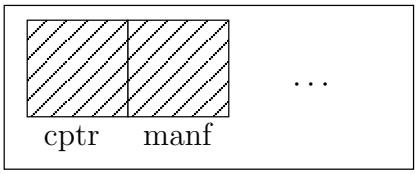

Figura 2.2: Caja *Codestream Index*.

es 'cidx'. Esta caja contiene a su vez una caja *Codestream Finder*, una caja *Manifest*, y una secuencia de cajas de índice.

La caja *Codestream Finder*, cuyo tipo es 'cptr', contiene la información necesaria para acceder directamente al *codestream* asociado a la supercaja *Codestream Index*. Esta información ocupa un total de 20 bytes y está divida en los siguientes valores:

- DR**:** Un entero sin signo de 2 bytes que especifica la posición del *codestream* o de la caja *Fragment Table* asociada. Si $DR = 0$, el *codestream* o la caja *Fragment Table* existe en el mismo archivo. Si $DR > 0$, el valor de DR es la entrada de la tabla contenida en la caja *Data Reference* que posee la referencia del archivo que contiene el *codestream* o la caja *Fragmente Table*.

- $CONT$**:** Es un entero sin signo de 2 bytes. Si $CONT = 0$, quiere decir que el *codestream* no está fragmentado, dentro del archivo referenciado por DR, por lo que los valores de $COFF$ y $CLEN$ se refieren al *codestream* entero. Si $CONT = 1$ el *codestream* está fragmentado, y requiere por tanto una caja *Fragment Table*. Entonces, los valores de $COFF$ y $CLEN$ se refieren a esta caja. Nótese que aún a pesar de tener tan sólo dos valores posibles, 0 o 1, el valor de $CONT$ se almacena inexplicablemente con dos bytes.

- $COFF$**:** Entero sin signo de 8 bytes que contiene el *offset* del *codestream* o la caja *Fragment Table* con respecto al comienzo del archivo referenciado por DR.

- $CLEN$**:** Entero sin signo de 8 bytes que contiene la longitud en bytes del *codestream* o la caja *Fragment Table*.

En nuestro caso, asumiremos que $DR = 0$ y $CONT = 0$ en todos los casos, por lo que obviaremos la descripción de las cajas *Data Reference* y *Fragment Table*.

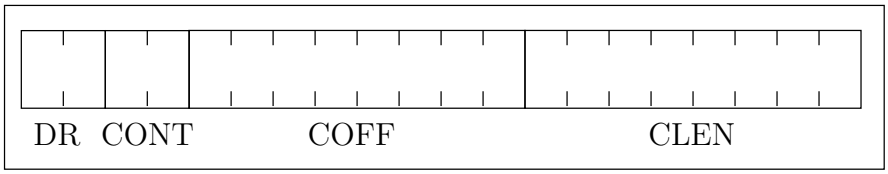

Figura 2.3: Caja *Codestream Finder*.

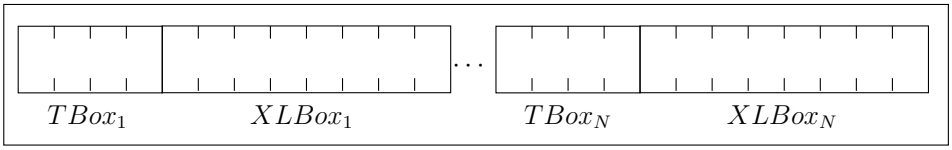

Figura 2.4: Caja *Manifest*.

La caja *Manifest*, cuyo tipo es 'manf', es empleada para indexar las cajas que se encuentren a continuación, en el mismo nivel. Por ejemplo, en el caso de la caja *Codestream Index*, la caja *Manifest* incluida en ella contendrá un índice de todas las cajas también incluidas en la caja *Codestream Index* situadas a continuación de la caja *Manifest*. El contenido de la caja está formado por una secuencia de N pares $(TBox_i, XLBox_i)$. El valor de N corresponderá al número de cajas que se presenten a continuación de la caja *Manifest*. $TBox_i$ es una cadena ASCII de cuatro bytes que contiene el tipo de la caja i. $XLBox_i$ es un entero sin signo de 8 bytes con la longitud en bytes de la caja i.

A continuación de la caja *Manifest*, incluida en la caja *Codestream Index*, nos encontramos con una caja de índice por cada *data-bin* del *codestream* asociado, esto es: cabecera principal, caja *Main Header Index Table*, cabecera de *tile*, caja *Tile Header Index Table* y precinto, caja *Precinct Packet Index Table*. Además es posible incluir otro tipo de cajas como caja de *tile-part*, caja *Tile-part Index Table*, y la caja para la cabecera de los paquetes, caja *Packet Header Index Table*.

La caja *Main Header Index Table*, de tipo 'mhix', contiene el valor $TLEN$ y una secuencia de N pares (OFF_i, LEN_i), donde N es el número de partes en las que está dividida la cabecera principal del *codestream*. $TLEN$ es un entero sin signo de 4 bytes que indica la longitud total de la cabecera en bytes. OFF_i es un entero sin signo de 4 bytes que contiene el *offset* de la parte i de la cabecera. LEN_i es un entero sin signo de 4 bytes que almacena

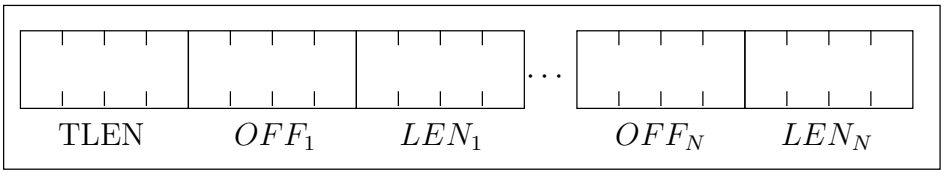

Figura 2.5: Caja *Main Header Index Table*.

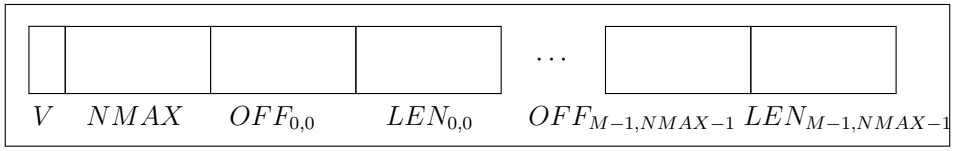

Figura 2.6: Caja *Fragment Array Index*.

la longitud en bytes de la parte i de la cabecera.

Excepto la caja *Main Header Index Table*, todas las demás cajas de índice utilizan la caja *Fragment Array Index*, de tipo 'faix', que es una caja que alberga una tabla bidimensional de fragmentos. En su contenido encontramos el valor V, el valor $NMAX$, y una secuencia de $M \times NMAX$ pares $(OFF_{i,j}, LEN_{i,j})$, donde $0 \leq i < M$ y $0 \leq j < NMAX$. V es un valor de 1 byte que indica el tamaño del resto de valores: si $V = 0$ el resto de valores ocupan 4 bytes, pero si $V = 1$ el resto de valores ocuparán 8 bytes. $NMAX$ es un entero sin signo que contiene el número de elementos en cada fila de la tabla. $OFF_{i,j}$ es un entero sin signo que almacena el *offset* del fragmento (i, j) de la tabla. $LEN_{i,j}$ es un entero sin signo con la longitud en bytes del fragmento (i, j) de la tabla. El valor de M se obtendría mediante la longitud total de la caja (contenida en la cabecera de la misma).

La caja *Tile-part Index Table*, de tipo 'tpix', contiene en su interior una única caja, una caja *Fragment Array Index*, en la cual habrá tantas filas como *tiles* haya, y tantas columnas como *tile-parts* haya por *tile*, de forma que el par $(OFF_{i,j}, LEN_{i,j})$ sería el *offset* y la longitud del *tile-part* j del *tile* i. Con esta caja indexamos los *tile-parts* enteros, desde el marcador SOT hasta el último paquete del mismo, ambos incluidos.

La caja *Tile Header Index Table*, de tipo 'thix', contiene también en su interior una única caja *Fragment Array Index*, la cual también contendrá tantas filas como *tiles* haya y tantas columnas como *tile-parts* haya por *tile*. Pero en esta ocasión esta caja no indexa el contenido total de cada *tile-part*

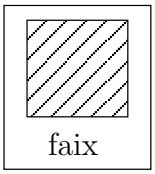

Figura 2.7: Caja *Tile-part Index Table*.

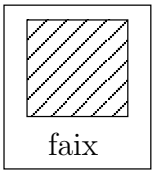

Figura 2.8: Caja *Tile Header Index Table*.

sino únicamente su cabecera, por lo que el par $(OFF_{i,j}, LEN_{i,j})$ sería el *offset* y la longitud de la cabecera del *tile-part* j del *tile* i. Las cabeceras de cada *tile-part* no son incluidas enteras, sino que sólo se incluye entera la cabecera del primer *tile-part* y del resto, sólo se incluye el marcador PPT, si existe. Esto es necesario tenerlo en cuenta porque, aunque el resto de los marcadores se ignores, ignorar el marcador POC si existiese puede repercutir en una descodificación errónea.

La caja *Precinct Packet Index Table*, de tipo 'ppix', contiene en su interior una caja *Manifest*, que indexa las cajas que vienen a continuación, y que forman un conjunto de N cajas, $faix_i$, de tipo *Fragment Array Index*. Cada caja $faix_i$ contiene información de todos los paquetes de la componente i de la imagen, de forma que, el par $(OFF_{j,k}, LEN_{j,k})$ de la caja $faix_i$, contiene el *offset* y la longitud del paquete k, del *tile* j, de la componente i. Los paquetes aparecerán de un mismo precinto aparecerán contiguos, y el orden será el mismo empleado para construir el número de secuencia s para calcular el identificador del *data-bin* del precinto.

La caja *Precinct Header Index Table*, de tipo 'phix', sigue la misma estructura que la anterior: contiene una caja *Manifest* que indexa las cajas que vienen a continuación, y que forman un conjunto de N cajas, $faix_i$, de tipo *Fragment Array Index*. Cada caja $faix_i$ contiene la información de todas las cabeceras de todos los paquetes de la componente i, de forma que, el par $(OFF_{j,k}, LEN_{j,k})$ de la caja $faix_i$ contiene el *offset* y la longitud de

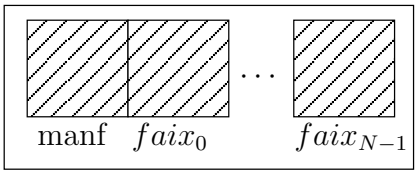

Figura 2.9: Caja *Precinct Packet Index Table*.

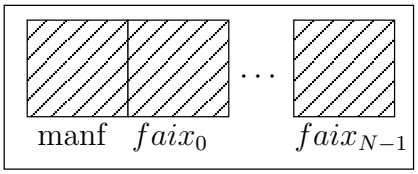

Figura 2.10: Caja *Precinct Header Index Table*.

la cabecera del paquete k, del *tile* j, de la componente i. La ordenación de las cabeceras de los paquetes seguirá la misma regla que la empleada en la caja *Precinct Packet Index Table* para los paquetes.

Capítulo 3

El protocolo HTTP/1.1

3.1. Introducción

El Protocolo de Transferencia de HiperTexto (*HyperText Transfer Protocol*), HTTP, es un sencillo protocolo cliente-servidor que articula los intercambios de información entre los clientes Web y los servidores HTTP. Fue propuesto por Tim Berners-Lee, atendiendo a las necesidades de un sistema global de distribución de información como el *World Wide Web*. La especificación completa del protocolo HTTP en su última versión, la 1.1, está recogida en el RFC 2068 [3].

El protocolo HTTP trabaja sobre el protocolo TCP y define una arquitectura típica cliente-servidor: un servidor se mantiene escuchando a través de un puerto de comunicaciones, por defecto el 80, esperando solicitudes de conexión de los clientes Web. Una vez que se establece la conexión, el protocolo TCP se encarga de mantener la comunicación y garantizar un intercambio de datos libre de errores.

HTTP se basa en sencillas operaciones de petición/respuesta. Un cliente establece una conexión con un servidor y envía un mensaje con los datos de la petición. El servidor responde con un mensaje similar, que contiene el estado de la operación y su posible resultado. Todas las operaciones se realizan sobre un recurso u objeto (documento HTML, aplicación CGI, archivo de imagen, etc.), el cual es conocido por su URL.

Las principales características del protocolo HTTP son:

- Toda la comunicación entre los clientes y servidor se realiza a partir de caracteres de 8 bits. De esta forma, se puede transmitir cualquier tipo de documento, ya sea texto o datos binarios, respetando su formato original.

- Permite la transferencia de objetos multimedia. El contenido de cada objeto intercambiado está identificado por su clasificación MIME.
- Existen tres tipos de peticiones básicas (existen más pero no son muy utilizados) que un cliente puede realizar a un servidor: solicitar un objeto (GET), enviar un objeto (POST), o solicitar la información de un objeto (HEAD).
- No mantiene el estado. Cada petición de un cliente a un servidor no es influida por las peticiones anteriores. El servidor trata cada petición como una operación totalmente independiente del resto. Si se quisiera mantener el estado, sería necesario el empleo de *Magic Cookies*.

3.2. Mensajes

El diálogo entre el cliente y el servidor se establece a través de mensajes.

3.2.1. Estructura

La estructura de un mensaje la siguiente:

```
<línea inicial>
[<cabeceras>]

[<cuerpo del mensaje>]
```

La primera parte del mensaje, la línea inicial, es una línea de texto en ASCII que finaliza con una secuencia CRLF (un carácter de 8 bits con el valor 13 seguido de otro con el valor 10). La segunda parte, opcional, es el conjunto de cabeceras, cada una de las cuales está formada por una o más líneas de texto en ASCII, todas ellas delimitadas por la secuencia CRLF. El cuerpo del mensaje, si existe, no tiene por qué ser texto. El tipo de los datos incluidos en el cuerpo del mensaje vendrá especificado en alguna de las cabeceras.

Los mensajes se clasifican en dos tipos: mensajes de petición, enviados por el cliente al servidor, y mensajes de respuesta, enviados por el servidor al cliente.

Para los mensajes de petición, la línea inicial tiene la siguiente forma:

```
<Comando> <URL> <Protocolo>
```

Como se puede ver, la línea de petición es dividida en 3 partes, separadas entre sí por un solo espacio en blanco. El comando especifica la acción a llevar a cabo sobre el objeto identificado por la URL incluida. El protocolo especifica la versión el protocolo que requiere el cliente. En el caso de la versión 1.1, la cadena del protocolo sería `HTTP/1.1`.

Un ejemplo de línea de petición, en la cual solicitamos una página Web, sería la siguiente:

```
GET /mipagina.html HTTP/1.1
```

En el caso de los mensajes de respuesta, a la línea inicial se le llama línea de estado y tiene la siguiente estructura:

```
<protocolo> <código de estado> <frase aclaratoria>
```

Al igual que ocurre con la línea inicial del mensaje de petición, la línea de estado también se divide en tres partes, separadas entre sí por un único espacio en blanco. El protocolo es la versión del protocolo que acepta el servidor y sigue la misma estructura que para los mensajes de petición. El código de estado es un número decimal de tres dígitos con el código del resultado del intento del servidor por llevar a cabo la acción requerida. La última parte es una pequeña frase aclaratoria, de una sola línea, del código de estado.

Un ejemplo de línea de estado devuelta por un servidor que no ha encontrado el objeto especificado en la URL de la petición, podría ser la siguiente:

```
HTTP/1.1 404 Not Found
```

3.2.2. Comandos

Algunos de los comandos aceptados en la versión 1.1 del protocolo HTTP son los siguientes:

- *GET*: Se utiliza para solicitar el recurso u objeto del servidor identificado por la URL.
- *HEAD*: Solicita información sobre un objeto, como puede ser el tamaño, el tipo, la fecha de modificación, etc. Es utilizado por los gestores de *caches* de páginas o los servidores *proxy*, para conocer cuando es necesario actualizar la copia que se mantiene de un fichero.

- *POST*: Sirve para enviar información al servidor, que puede ser por ejemplo los datos contenidos en un formulario de una página Web.
- *PUT*: Permite actualizar información de un objeto determinado del servidor. Es similar a *POST* pero en este caso la información enviada al servidor debe ser almacenada en la URL que acompaña al comando.

Otros de los comandos permitidos en HTTP/1.1 son *OPTIONS*, *TRACE* y *DELETE*.

3.2.3. Códigos de estado

Existen cinco categorías de códigos de estado, organizadas por el primer dígito del código:

- 1xx: Para mensajes informativos.
- 2xx: Asociados con operaciones realizadas correctamente.
- 3xx: Para informar de operaciones complementarias que se deben realizar para finalizar la operación requerida por el cliente.
- 4xx: Indican error por parte del cliente, como puede ser error de sintaxis en el mensaje de petición, o petición invalida.
- 5xx: Indican error por parte del servidor, al no poder completar una petición que realizó una petición aparentemente correcta.

La lista de todos los códigos definidos para el protocolo HTTP/1.1 se muestra en la tabla 3.1. Esta lista es la lista básica que deben de soportar los clientes, pero se pueden definir otros tipos códigos para aplicaciones específicas.

Los códigos de estado más comunes son los siguientes:

- 200 ("OK"): Operación realizada satisfactoriamente.
- 201 ("Created"): La operación ha sido realizada correctamente y como resultado se ha creado un nuevo objeto, cuya URL de acceso se proporciona en el cuerpo de la respuesta. Este nuevo objeto ya está disponible.
- 202 ("Accepted"): La operación ha sido realizada correctamente y como resultado se ha creado un nuevo objeto, cuya URL de acceso se proporciona en el cuerpo de la respuesta. El nuevo objeto no está disponible por el momento.

Código	Frase aclaratoria
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Moved Temporarily
303	See Other
304	Not Modified
305	Use proxy
400	Bad Request
401	Unauthorized
402	Payment required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Time-out
409	Conflict
410	Gone
411	Length Required
412	Precondition failed
413	Request Entity Too Large
414	Request-URI Too Large
415	Unsupported Media Type
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Time-out
505	HTTP Version not supported

Tabla 3.1: Códigos de estado del protocolo HTTP/1.1.

- 204 ("No Content"): La operación ha sido aceptada, pero no ha producido ningún resultado de interés. El cliente no deberá modificar el documento que está mostrando en este momento.
- 301 ("Moved Permanently"): El objeto al que se accede ha sido movido a otro lugar de forma permanente. El servidor proporciona, además, la nueva URL en la variable *Location* de la respuesta.
- 302 ("Moved Temporarily"): El objeto al cual se accede ha sido movido a otro lugar de forma temporal. El servidor proporciona además la nueva URL en la variable *Location* de la respuesta.
- 304 ("Not Modified"): Cuando se hace un *GET* condicional, y el documento no ha sido modificado, se devuelve este código de estado.
- 400 ("Bad Request"): La petición tiene un error de sintaxis y no es entendida por el servidor.
- 401 ("Unauthorized"): La petición requiere una autorización especial, que normalmente consiste en un nombre y clave que el servidor verificará. El campo *WWW-Authenticate* informa de los protocolos de autentificación aceptados para este recurso.
- 403 ("Forbidden"): Está prohibido el acceso a este recurso. No es posible utilizar una clave para modificar la protección.
- 404 ("Not Found"): La URL solicitada no existe.
- 500 ("Internal Server Error"): El servidor ha tenido un error interno, y no puede continuar con el procesamiento.
- 501 ("Not Implemented"): El servidor no tiene capacidad, por su diseño interno, para llevar a cabo el requerimiento del cliente.
- 503 ("Service Unavaliable"): El servidor está actualmente deshabilitado, y no es capaz de atender el requerimiento.

3.2.4. Cabeceras

Las cabeceras son un conjunto de variables que se incluyen en los mensajes HTTP con el fin de modificar su comportamiento o incluir información información de interés. La estructura de una cabecera es la siguiente:

```
<Nombre de la variable>: <valor>
```

Los nombres de las variables se pueden escribir con cualquier combinación de mayúsculas y minúsculas. Se debe incluir un espacio en blanco entre el carácter `':'` y el valor de la variable. No debe de existir ningún espacio en blanco antes del nombre de la variable. El valor de la variable puede ocupar varias líneas, todas ellas separadas por la secuencia CRLF, debiendo cumplir que al inicio de cada una de ellas debe haber al menos un espacio en blanco o un tabulador.

Existen cuatro tipos de cabeceras: cabeceras generales, cabeceras de petición, cabeceras de respuesta y cabeceras de entidad.

Las cabeceras generales se pueden incluir tanto en los mensajes de petición como en los mensajes de respuesta. Algunas de estas cabeceras son las siguientes:

- *Date*: Especifica la fecha local de la operación, la cual debe incluir la zona horaria correspondiente.
- *Pragma*: Permite incluir información variada relacionada con el protocolo HTTP en la petición respuesta que se está realizando. Por ejemplo, un cliente puede incluir la cabecera `Pragma: no-cache` para informar de que desea una copia nueva del recurso especificado, ignorando la *cache* que pueda existir.
- *Connection*: Su uso se verá más tarde. Puede ser empleada tanto como por el cliente como por el servidor para indicarle el uno al otro que la conexión va a ser cerrada.
- *Transfer-Encondig*: Indica qué tipo de transformación ha sido aplicada al cuerpo del mensaje con el fin de transmitirlo de forma óptima.

Las cabeceras de petición sólo pueden ser incluidas en los mensajes de petición. Algunas de estas cabeceras son las siguientes:

- *Accept*: Es un campo opcional que contiene una lista de tipos MIME aceptados por el cliente.
- *Authorization*: Define la clave de acceso que envía un cliente para acceder a un recurso de uso protegido o limitado. La información incluye el formato de autorización empleado, seguido de la clave de acceso propiamente dicha.
- *From*: Campo opcional que contiene la dirección de correo electrónico del usuario del cliente.

- *If-Modified-Since*: Permite realizar operaciones *GET* condicionales, en función de si la fecha de modificación del objeto requerido es anterior o posterior a la fecha proporcionada.
- *User-agent*: Indica el tipo y la versión del cliente que realiza la petición.
- *Range*: Como se verá más adelante, el cliente puede especificar mediante esta cabecera el rango de bytes que requiere del recurso solicitado al servidor.
- *Host*: Especifica la dirección, y opcionalmente, el puerto, del servidor al cual se le envía la petición. Todas las peticiones HTTP/1.1 que pasen a través de Internet deben incluir esta cabecera.

Las cabeceras de respuesta sólo pueden ser incluidas en los mensajes de respuesta. Algunas de estas cabeceras son las siguientes:

- *Location*: Informa sobre la dirección exacta del recurso al que se ha accedido. Cuando el servidor proporciona un código de respuesta de la serie 3xx, este parámetro contiene la URL necesaria para accesos posteriores a este recurso.
- *Server*: Especifica la cadena que identifica el tipo y la versión del servidor.
- *WWW-Authenticate*: Cuando se accede a un recurso protegido o de acceso restringido, el servidor devuelve un código de estado 401 y utiliza este campo para informar de los modelos de autentificación válidos para acceder a este recurso.
- *Accept-Ranges*: Esta cabecera es enviada por el servidor para notificarle al cliente si acepta o no la posibilidad de *byte-ranging*.

Tanto los mensajes de petición como los de respuesta pueden incluir datos, es decir, el cuerpo del mensaje. Las cabeceras que contienen información relacionada con los datos almacenados en el cuerpo del mensaje son denominadas cabeceras de entidad. Algunas de las cabeceras de entidad son las siguientes:

- *Content-Enconding*: Si los datos del cuerpo del mensaje han sido codificados de alguna forma, por ejemplo, que hayan sido comprimidos o encriptados, esta cabecera especifica el tipo de codificación aplicada.

- *Content-Length*: Define la longitud total del cuerpo del mensaje.
- *Content-Range*: Cuando se está leyendo rangos de bytes, *byte-ranging*, el mensaje de trocea en submensajes. Cada submensaje posee esta cabecera que especifica de qué rango se trata.
- *Content-Type*: Especifica el tipo MIME de los datos incluidos en el cuerpo del mensaje. Cuando el servidor envía datos a un cliente, el cliente utiliza el valor de esta cabecera para saber cual es el tratamiento adecuado a aplicar a dichos datos.

3.3. Características avanzadas

3.3.1. Mensajes *chunkeados*

El servidor puede trocear en bloques el mensaje de respuesta, con el fin de poder transmitir los datos de forma óptima. Esto puede ocurrir cuando la longitud del mensaje a transmitir sea excesiva o cuando el servidor no conozca a priori el tamaño total de los datos. En esta caso, al mensaje se le denomina mensaje *chunkeado* (*chunked message*).

Imaginemos que realizamos una petición a un cliente HTTP/1.1 como la siguiente:

```
GET /muyGrande.dat HTTP/1.1
Host: www.servidor.com
```

El servidor decide trocear el mensaje, con lo cual, la respuesta sería similar a como sigue:

```
HTTP/1.1 200 OK
Transfer-Enconding: chunked

3fe
Estos son los datos del bloque 1
y que tienen una longitud de
0x3FE bytes ...
a2
Y estos son los datos del bloque 2,
que tienen una longitud de
0xA2 bytes ...
```

```
0
```

Como se puede apreciar, cuando el servidor decide trocear el mensaje, incluye la cabecera *Transfer-Encoding* con el valor *chunked*. En el ejemplo, el cuerpo del mensaje, que comienza a partir de la línea en blanco situada justo después de las cabeceras del mensaje, está dividido en dos trozos. Cada trozo comienza con la longitud de los datos que contiene, expresada en hexadecimal. Dicha longitud finaliza con la secuencia CRLF, a partir de la cual comienzan los datos. Después de los datos existe una secuencia CRLF. Después del último trozo se incluye un cero, en ASCII, indicando que no existen más trozos. A continuación del cero se incluye una secuencia CRLF, opcionalmente un conjunto de cabeceras de entidad, y finalmente, otra secuencia CRLF. En el ejemplo no se ha incluido ninguna cabecera de entidad, con lo cual el mensaje de respuesta finaliza con una línea vacía. Las cabeceras de entidad contendrían información acerca de los datos de cada trozo.

Todo cliente de HTTP/1.1 debe ser capaz de recibir y decodificar mensajes *chukeados*.

3.3.2. Conexiones persistentes

En las versiones anteriores del protocolo HTTP, cada vez que un cliente realizaba una petición a un servidor, éste establecía una conexión, realizaba la petición, y una vez recibida la respuesta, la conexión era cerrada. De esta forma, debía abrir y cerrar una conexión por cada petición, lo que provocaba una sobrecarga de los servidores y una congestión en Internet.

En la versión 1.1 del protocolo HTTP, las conexiones se vuelven persistentes, es decir, el cliente puede realizar multitud de peticiones con la misma conexión. Esto, que es algo disponible en la mayoría de los protocolos cliente/servidor, no era posible hasta el momento.

En HTTP/1.1 todas las conexiones son persistentes. El servidor asume que el cliente desea tener abierta la conexión, a menos que reciba la cabecera *Connection* con el valor *close*. Por otra parte, el cliente también asumirá que el servidor mantendrá abierta la conexión, a menos que reciba la cabecera *Connection* con el valor *close*. Tanto el cliente como el servidor, una vez reciben la cabecera *Connection* con el valor *close*, deben cerrar su conexión.

A pesar de ser conexiones persistentes, muchos servidores incluirán la cabecera *Connection* con el valor *close* tras la primera petición, obligando al cliente a cerrar su conexión. El cliente puede no obstante retrasar esto,

incluyendo en sus peticiones la cabecera *Connection* con el valor *Keep-Alive*. El servidor mantendrá entonces abierta la conexión el tiempo que tenga configurado.

El cliente que use conexiones persistentes debe limitar el número de conexiones simultaneas que mantiene con un servidor determinado como mucho a 2.

3.3.3. *Pipelining*

Con la inclusión de la posibilidad de crear conexiones HTTP/1.1 persistentes, se añade también la de realizar *pipelining* con los mensajes. Esto quiere decir que el cliente puede enviar múltiples peticiones seguidamente, sin tener que esperar, como ocurría con versiones anteriores, a recibir la respuesta de una petición para poder realizar otra. El servidor tiene la obligación de enviar las respuestas en el mismo orden en el que recibió las peticiones.

Lo único que es preciso tener en cuenta es que, como se comentó anteriormente, el servidor podría incluir en alguna de sus respuestas la cabecera *Connection* con el valor *close*, y posteriormente cerrar la conexión. Esto, si sólo disponemos de un sólo hilo para realizar todas las peticiones a la vez, y luego esperar las respuestas, no existe mayor problema, ya que terminaríamos de leer respuestas en cuanto se recibiera dicha cabecera. Pero si decidiéramos optimizar aún más la comunicación, y tuviésemos dos hilos, uno de los cuales fuese el encargado de enviar todas las peticiones y el otro el encargado de ir leyéndolas, en cuanto el que las lee recibiera la cabecera que indica un cierre de la conexión, debería comunicárselo al hilo encargado de las peticiones, si aún está en ejecución, para que finalice.

3.3.4. *Byte-ranging*

El protocolo HTTP/1.1 permite solicitar porciones (*byte-ranges*) de un determinado recurso u objeto del servidor. Esto es especialmente útil para aplicaciones que no requieren el contenido entero de un determinado archivo, sino aquella parte que solicite de forma interactiva el usuario.

Un ejemplo de esta situación lo podemos encontrar en el *plug-in* de *Acrobat* que es utilizado para visualizar archivos PDF en los navegadores. De esta forma, el usuario no tiene que esperar a recibir el documento completo para comenzar a inspeccionar una parte del mismo.

Un servidor HTTP/1.1 no tiene obligación de ofrecer la posibilidad de *byte-ranging*. Si un servidor lo soporta, existe la posibilidad, aunque tampoco

es obligatorio, de que envíe la cabecera *Accept-Ranges* con el valor bytes (el único valor permitido hasta el momento, y que indica que los rangos son en unidades de bytes). Si un servidor no lo soporta existe la posibilidad, aunque tampoco es obligatorio, de que envíe la cabecera *Accept-Ranges* con el valor *none*. El problema es que es posible que el servidor soporte *byte-ranging* pero no lo comunique. Si quisiéramos realizar *byte-ranging* sobre un servidor, del cual no hemos recibido la confirmación de que lo soporta, pero tampoco la negación, y éste no lo soportase, recibiríamos un código de error 501 ("Not Implemented").

El rango de bytes que se quiera se especifica en la cabecera *Range*. En una misma petición se puede especificar uno o más de un rango. El valor de esta cabecera se forma de la siguiente forma:

```
bytes=<rango1>,<rango2>,...,<rangoN>
```

Los rangos se separan por comas. No debe existir espacio alguno entre el carácter `'='` y la palabra `'bytes'`, así como entre el carácter `'='` y el primer rango especificado. Cada rango de bytes se forma de la siguiente forma:

```
[<inicio>]-[<fin>]
```

Tanto *inicio* como *fin* son números decimales. Si ambos están presentes, *inicio* representa el *offset* del primer byte a transmitir, incluido, y *fin* representa el *offset* del último byte a transmitir, incluido también. Los *offset* son respecto al inicio del archivo. Si *inicio* está ausente, *fin* representa el número de bytes del final del archivo a transmitir. Si *fin* está ausente, se asume que es el final del archivo.

Algunos ejemplos serían:

- Los últimos 100 bytes:

  ```
  bytes=-100
  ```

- Los primeros 100 bytes:

  ```
  bytes=0-99
  ```

- Desde el byte 80 hasta el final del archivo:

  ```
  bytes=80-
  ```

- El rango desde el byte 10 hasta el byte 15 y desde el 20 al 25:

```
bytes=10-15,20-25
```

Cuando el servidor recibe una petición de uno o más *byte-ranges* de un archivo determinado, éste, en el caso de responder satisfactoriamente, devolverá el código 206 ("Partial Content"). La estructura de la respuesta del servidor es distinta de si requerimos un sólo *byte-range* o más de uno.

Imaginemos que solicitamos los primeros 100 bytes de un archivo, de la siguiente forma:

```
GET /imagen.jpg HTTP/1.1 Host: www.servidor.com Range: bytes=0-99
```

El servidor, en el caso de responder satisfactoriamente, enviaría una respuesta similar a la siguiente:

```
HTTP/1.1 206 Partial Content
Server: Apache/1.3.12 (Win32)
ETag: "0-fffe-278427e3"
Accept-Ranges: bytes
Content-Length: 100
Content-Range: bytes=0-99/1024
Content-Type: image/jpeg

(Los 100 bytes iniciales de la imagen...)
```

En la respuesta vemos que la cabecera *Content-Length* contiene el tamaño total del rango solicitado, 100 bytes. La cabecera *Content-Range* contiene una copia del rango especificado en la petición, `'0-99'`, seguida del carácter `'/'` y un número decimal que representa el tamaño total del archivo. Las demás cabeceras son las típicas de cualquier respuesta. Nótese que en este ejemplo, el servidor sí nos notifica que acepta *byte-ranging*, incluyendo la cabecera *Accept-Ranges* con el valor bytes.

En el ejemplo anterior vemos que cuando la petición se refiere a un único rango, la respuesta que se devuelve es una respuesta "típica", a excepción del código 206, que nos indica que es un *byte-range*, y de la inclusión de la cabecera *Content-Range*. Para el caso de las peticiones de varios *byte-ranges*, la respuesta del servidor es algo más compleja. Veamos el siguiente ejemplo, en el cual solicitamos los 100 primeros bytes, el rango que va desde el byte 120 al 150 y el rango que va desde el byte 200 al 230:

```
GET /imagen.jpg HTTP/1.1
Host: www.servidor.com
Range: bytes=0-99,120-150,200-230
```

El servidor, en el caso de responder satisfactoriamente, enviaría una respuesta similar a la siguiente:

```
HTTP/1.1 206 Partial Content
Server: Apache/1.3.12 (Win32)
ETag: "0-55fe-278427e3"
Accept-Ranges: bytes
Content-Length: 100
Content-Type: multipart/byteranges; boundary=cadenadelimitadora


--cadenadelimitadora
Content-Type: image/jpeg
Content-Range: 0-99/1024

(Datos del primer rango...)
--cadenadlimitadora
Content-Type: image/jpeg
Content-Range: 120-150/1024

(Datos del segundo rango...)
--cadenadelimitadora
Content-Type: image/jpeg
Content-Range: 200-230/1024

(Datos del tercer y último rango...)
--cadenadelimitadora--
```

En la respuesta del servidor vemos como ahora la cabecera *Content-Length* no especifica el tamaño de ningún rango, sino del cuerpo del mensaje de respuesta, que incluye tanto la suma de los rangos solicitados como las subcabeceras de cada rango. El tipo MIME especificado en la cabecera *Content-Type* ya no es *image/jpeg*, sino *multipart/byteranges*. Esta cabecera incluye además una cadena delimitadora, a continuación de la cadena `'boundary='`. Esta cadena delimitadora será la empleada para separar los

diferentes rangos en el mensaje. Hay que tener en cuenta que, a diferencia de lo que ocurre al solicitar un sólo rango, al solicitar varios rangos no es que el mensaje se transmita en partes, una por rango, sino que el mensaje es un tipo de mensaje especial (*multipart/byteranges*), en cuyo cuerpo están incluidos los rangos requeridos. Esto quiere decir que el servidor podría enviar el mensaje de respuesta a una petición multi-rango de forma *chunkeada* si lo considerara conveniente.

El cuerpo del mensaje comienza, a diferencia del resto de mensajes, tras dos líneas vacías, y está dividido en tantas secciones como rangos se haya solicitado. Al inicio de cada sección se incluye una línea con la cadena `'--'` seguida de la cadena delimitadora. A continuación se encuentra un conjunto de cabeceras relacionadas con el rango asociado a dicha sección. Las cabeceras suelen ser la cabecera *Content-Range* y la cabecera *Content-Type*. La primera cabecera, al igual que se comentó para el caso de peticiones de un único rango, contiene el rango correspondiente y el tamaño total del archivo. Los datos del rango de una sección comienzan tras la primera línea vacía. Al final del mensaje se incluye una línea con la cadena `'--'` seguida de la cadena delimitadora y `'--'`.

Capítulo 4

Kakadu

4.1. Introducción

Kakadu [12] es una implementación propietaria de la Parte 1 del estándar JPEG2000, realizada por David Taubman. Existen otras implementaciones [1], pero ésta es quizás la mas eficiente. En el presente capítulo se dará una breve descripción de la versión 4.0 de Kakadu.

4.2. Características

El núcleo del sistema está escrito en C++ y ha sido implementado con la intención de ser totalmente independiente de plataforma. La plataforma destino debe al menos soportar enteros de 32 bits. El código está especialmente optimizado para procesadores Pentium.

Debido a la complejidad del estándar JPEG2000, los puntos en los que se ha centrado la implementación de Kakadu son: el uso de la memoria, la velocidad de ejecución, y el ofrecer una arquitectura altamente flexible, con el fin de cubrir las necesidades de una amplia variedad de aplicaciones, desde meros compresores/descompresores hasta complejas aplicaciones interactivas o sistemas cliente/servidor.

4.3. Organización

El núcleo de la implementación de Kakadu es lo que denominamos el sistema base, o *core system*. Es un conjunto de clases que proporcionan la funcionalidad necesaria para interaccionar con los *codestreams* del estándar JPEG2000, según la especificación de la Parte 1. Este núcleo está diseñado

para ser totalmente portable a cualquier plataforma. Basándose en este núcleo, se crea otro conjunto de clases para proporcionar funcionalidad adicional, como es el soporte del protocolo JPIP definido en la Parte 9 del estándar o el soporte para diferentes archivos de imágenes (.JP2, .MJ2, etc.). Se añade además un conjunto de aplicaciones con el fin de demostrar el funcionamiento de dichas clases. Tan sólo el *core system* es totalmente portable. En el resto de la implementación de Kakadu, habrá partes portables y otras, en cambio, necesitarán de las correspondientes modificaciones para ser portadas. Todo el código de Kakadu está escrito empleando el lenguaje de programación C++.

4.3.1. Sistema base (*core system*)

El conjunto de clases que forman el sistema base está dividido en subsistemas atendiendo a su funcionalidad. La figura 4.1 muestra los principales subsistemas del sistema base de Kakadu (no refleja el subsistema para el procesamiento de la ROI). Las líneas de puntos vinculan clases del sistema base con representaciones internas de la arquitectura de un *codestream*. Las instancias de estas clases actúan a modo de interfaz, es decir, ofrecen una serie de servicios con los cuales operar con los elementos asociados, pero no almacenan nada.

Los principales subsistemas del sistema base son los siguientes:

- **Subsistema de parámetros (*Coding Parameter Sub-System*)**: Los parámetros de codificación incluidos en un *codestream*, y que son necesarios para de descodificación adecuada de la imagen, está repartidos por un conjunto de marcadores, como son SIZ, COD, COC, etc. Estos están situados en diferentes partes, como la cabecera principal, la del *tile* o la del *tile-part*, y se sobreescriben unos a otros en función de su posición. El subsistema de parámetros es un conjunto de clases, todas ellas derivadas de la clase *kdu_params*, cuyas instancias se organizan internamente mediante listas multidimensionales para reflejar la jerarquía real de los parámetros. Estas clases ofrecen una interfaz cómoda y eficaz. La mayor parte de la interacción con el subsistema de parámetros se realiza a través del objeto raíz de toda la jerarquía de objetos, el cual siempre será una instancia de la clase *siz_params*. Se tiene acceso a este objeto a través de la clase *kdu_codestream*.

  Las principales clases incluidas en este subsistema son las siguientes:

  - *crg_params*: Para albergar la información de los marcadores CRG.

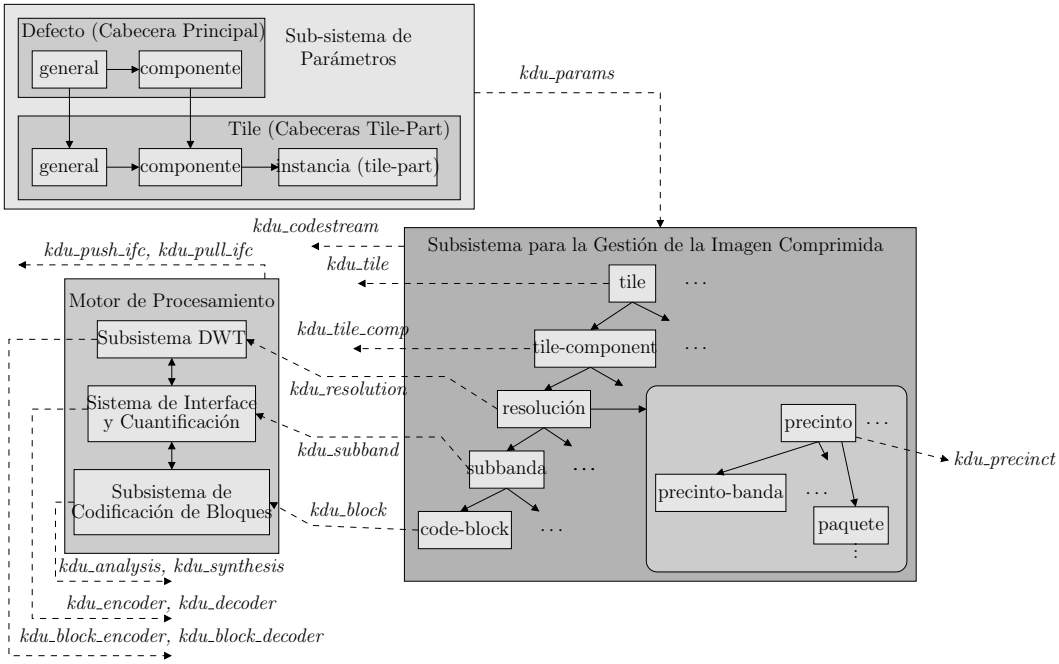

Figura 4.1: Sistema base de Kakadu.

- *poc_params*: Para albergar la información de los marcadores POC.
- *rgn_params*: Para albergar la información de los marcadores RGN.
- *qcd_params*: Para albergar la información de los marcadores QCD.
- *siz_params*: Para albergar la información del marcador SIZ. Además es la clase del objeto raíz de toda la jerarquía interna de objetos de parámetros.
- *org_params*: No contiene información de ningún marcador en especial sino que contiene información acerca de la organización del *codestream*, como por ejemplo, la estructuración de los *tile-parts*.
- *cod_params*: Para albergar la información de los marcadores COD.

▪ **Subsistema para la gestión de la imagen comprimida (*Compressed Image Management Sub-System*)**: Es quizás el subsistema más importante de todo el sistema base. Está formado por una

jerarquía de clases, cada una de las cuales ofrece una interfaz para operar con un tipo determinado de partición de la imagen almacenada en el *codestream* (*tile*, *tile-component*, resolución, precinto, etc.). La raíz es la clase *kdu_codestream*. A partir de esta clase se accede al resto de clases que representan un tipo de partición, como *kdu_tile* para el *tile*, *kdu_resolution* para la resolución, *kdu_subband* para la subbanda, etc. La partición más pequeña de la cual se ofrece una clase interfaz es el *code-block*, con la clase *kdu_block*.

Las principales clases incluidas en este subsistema son las siguientes:

- *kdu_codestream*: Es la clase principal, mediante la cual se tiene acceso al resto de clases.
- *kdu_tile*: Representa a un tile determinado.
- *kdu_tile_comp*: Representa a un tile-component determinado. Se tiene acceso a través de la clase anterior.
- *kdu_resolution*: Representa a una resolución determinada. Se tiene acceso a través de la clase anterior.
- *kdu_subband*: Representa a una subbanda determinada. Se tiene acceso a través de la clase anterior.
- *kdu_precinct*: Representa a un precinto determinado. Se tiene acceso a través de la clase *kdu_resolution*.
- *kdu_block*: Representa a un *code-block* determinado. Se tiene acceso a través de la clase anterior.

- **Subsistemas de procesamiento de los datos (*Data Processing Sub-systems*)**: Este es el conjunto de los subsistemas que identifican a cada una de las etapas del procesamiento de la imagen, como es la transformada DWT, la cuantificación, la codificación o la transformada de color, si existiese. Cada uno de ellos contiene un conjunto de clases con el código necesario para llevar a cabo dicha etapa. Cabe decir que todas estas clases interaccionan con el *codestream* mediante la interfaz de las clases *kdu_resolution*, *kdu_subband* y *kdu_block*, mencionadas anteriormente.

  Todas las clases de estos subsistemas son derivadas, para la compresión, de la clase *kdu_push_ifc*, y para la descompresión, de la clase *kdu_pop_ifc*.

  Las principales clases del subsistema encargado de la transformada DWT son:

- *kdu_analysis*: Para la transformada DWT.
- *kdu_synthesis*: Para la inversa de la transformada DWT.

Las principales clases del subsistema encargado de la codificación/descodificación son:

- *kdu_encoder*: Para la codificación.
- *kdu_decoder*: Para la descodificación.

Como se ha mencionado, la clase *kdu_codestream* es la clase central del subsistema encargado de ofrecer acceso a las diversas particiones de una imagen y, además, de servir de soporte al resto de los subsistemas. Se podría decir que la clase más importante del sistema base de Kakadu es la clase *kdu_codestream*.

Un objeto instancia de la clase *kdu_codestream* se inicializa mediante un objeto de la clase *kdu_compressed_source*, que identifica una fuente de datos comprimidos, un *codestream*. El tipo de fuente de datos puede ser la combinación de uno o más de los siguientes tipos:

- Secuencial (*Sequential*): Obliga a que los datos sean leídos secuencialmente, desde el principio al fin.
- De acceso aleatorio (*Seekable*): Permite acceder aleatoriamente a los datos.
- Con *cache* (*Cached*): La fuente de los datos tiene la estructura de una *cache*, organizada en *data-bins*, según especifica el protocolo JPIP.

La clase *kdu_compressed_source* es una clase abstracta. Kakadu incluye un conjunto de clases, derivadas de esta clase, con el fin de dar soporte a diferentes fuentes de datos (archivos J2C *raw*, JP2, MJ2, etc.). Estas clases ya no forman parte del sistema base.

Con el fin de minimizar el uso de memoria, todos los objetos del sistema son destruidos una vez se dejen de utilizar. Cuando se "accede" a un determinado elemento, se crea la interfaz apropiada para trabajar con él. Cuando se "cierra" dicho elemento, se destruye. Por ejemplo, ejecutando el método *open_tile* de la clase *kdu_codestream*, se tiene acceso a un *tile* determinado, creándose un objeto de la clase *kdu_tile*. Al cerrar el *tile* llamando al método *close* de la clase *kdu_tile*, la información de dicho *tile* es eliminada, por lo que no se podrá volver a acceder a él. Este es el comportamiento por defecto, aunque puede ser deshabilitado, haciendo que todos los recursos sean persistentes, con el método *set_persistent* de la clase *kdu_codestream*.

4.3.2. Clases para el acceso a archivos

Kakadu, en su versión 4.0 ofrece soporte completo para el tratamiento de archivos con formato JP2 y MJ2. Se espera que en versiones posteriores se de soporte para los formatos JPX y JPM. Los archivos *raw* J2C, que contienen un *codestream* únicamente, también son soportados.

Las principales clases para el acceso a los archivos son las siguientes:

- *jp2_source*: Permite el acceso a los archivos JP2.
- *mj2_video_source*: Permite el acceso a los archivos MJ2.
- *kdu_simple_file_source*: Permite el acceso a cualquier tipo de archivo sin estructura definida, como los J2C.

Todas estas clases derivan de la clase *kdu_compressed_source*. Son prácticamente portables a cualquier plataforma.

4.3.3. Clases para el soporte del protocolo JPIP

Kakadu ofrece el soporte necesario para desarrollar aplicaciones cliente/servidor empleando el protocolo JPIP, definido en la Parte 9 del estándar JPEG2000. En realidad, Kakadu realiza su propia implementación, variando ligeramente la especificación del protocolo.

Las principales clases que dan soporte al protocolo JPIP son las siguientes:

- *kdu_cache*: Es una clase derivada de la clase *kdu_compressed_source*, y su misión es proveer una fuente de datos con estructura de *cache*. Esta clase es especialmente empleada en aplicaciones cliente/servidor, como las que se pueden desarrollar basándose en el protocolo JPIP. En la *cache*, los datos son organizados en *data-bins*. La clase ofrecerá la posibilidad de eliminar de la *cache* los *data-bins* que hace más tiempo que fueron usados.
- *kdu_client*: Implementa un completo cliente JPIP. Deriva de la clase *kdu_cache*. Su implementación no es independiente de plataforma, ya que está diseñada para correr bajo sistemas Windows. Si se quisiera pasar a otro sistema operativo, habría que cambiar la parte de utilización de sockets y creación de hilos.
- *kdu_serve*: Proporciona el componente independiente de plataforma necesario para la creación de un servidor JPIP. La aplicación *kdu_server* hace uso de esta clase.

4.3.4. Clases para la reconstrucción de la imagen

Para la reconstrucción de la imagen, a partir de un *codestream*, identificado por un objeto de la clase *kdu_codestream*, Kakadu sólo pone a disposición la clase *kdu_region_decompressor*.

Esta clase es especialmente útil para aplicaciones que requieren una descompresión interactiva, como visualizadores o editores de imágenes.

Para reconstruir una región de una imagen tan sólo le especificamos al objeto de la clase *kdu_region_decompressor* la referencia al objeto de la clase *kdu_codestream* que identifica a la imagen y qué región es, abstrayéndonos del procedimiento necesario a seguir para conseguir tal fin.

La reconstrucción de una región de la imagen es dividida en etapas, cuyo número depende del tamaño del *buffer* que disponga el objeto para llevar a cabo la tarea.

La implementación de esta clase es totalmente independiente de arquitectura.

4.3.5. Aplicaciones de ejemplo

Todas las aplicaciones de ejemplo incluidas en la versión 4.0 de Kakadu pueden correr en los sistemas Windows, Linux y Solaris, a excepción de *kdu_client* y *kdu_server*, diseñadas únicamente para correr en sistemas Windows.

Algunas de las aplicaciones de ejemplo son las siguientes:

- *kdu_compress*: Permite la compresión de imágenes, almacenadas en otros formatos como RAW, BMP, PPM o PGM, mediante el estándar JPEG2000. El formato de salida puede ser o bien J2C o JP2. Como parámetros se le pasa el tipo de compresión/codificación a realizar, el tamaño de los precintos, de los *code-blocks*, el número de *tiles*, el tipo de progresión, etc.

- *kdu_v_compress*: Como archivo de entrada acepta un archivo con formato VIX, que consiste en un secuencia de imágenes de vídeo en formato *raw* con una cabecera de texto, y produce como salida un archivo MJ2 con la compresión del vídeo según el estándar JPEG2000.

- *kdu_maketlm*: Permite añadir a un archivo de imagen, ya sea en formato J2C o en formato JP2, los marcadores TLM (generalmente, sólo habrá uno). Se le puede indicar al compresor, *kdu_compress*, que incluya también los marcadores PLT, pero la inclusión de los marcadores TLM se hace de forma independiente con esta aplicación.

- *kdu_expand*: Permite descomprimir una imagen JPEG2000, almacenada en un archivo J2C o en un archivo JP2. El formato de salida puede ser RAW, BMP, PGM o PPM. La aplicación ofrece la posibilidad de descomprimir sólo un área de la imagen.

- *kdu_v_expand*: Descomprime la secuencia de vídeo almacenada en un archivo MJ2 y produce un archivo con formato VIX.

- *kdu_transcode*: A partir de un archivo de imagen JPEG2000, esta aplicación permite cambiar la compresión/codificación de la misma. Tiene el mismo efecto que si descomprimiéramos la imagen y la volviésemos a comprimir con unos parámetros de compresión diferentes.

- *kdu_show*: Es la aplicación estrella de Kakadu. Se trata de un visualizador interactivo de imágenes. Permite visualizar imágenes locales, en formato JP2, JPX o J2C, pero también permite visualizar imágenes remotas actuando a modo de cliente JPIP.

- *kdu_hyperdoc*: Es una herramienta para generar la documentación en formato HTML de las clases a partir de los comentarios incluidos en su código fuente. También es empleada en Kakadu para crear las interfaces Java de las clases.

- *kdu_server*: Es un servidor JPIP.

4.4. Compatibilidad con Java

Empleando la API JNI [4], Kakadu ofrece la posibilidad de construir clases interfaces en Java para las clases escritas en C++. La razón de la inclusión en Kakadu de esta posibilidad es para intentar dar una solución a las dependencias de la arquitectura de algunas aplicaciones. Aunque el sistema base y muchas de las aplicaciones incluidas son fácilmente compilables para cualquier tipo de arquitectura, existen otras aplicaciones, como el *kdu_show*, que son fuertemente dependientes de la plataforma. Las partes fuertemente dependientes de arquitectura de una aplicación son la interfaz de usuario, la comunicación por red y la gestión de hilos concurrentes. Estas partes pueden ser realizadas en Kakadu con Java, de forma el núcleo de nuestra aplicación estará escrita en Java, con la portabilidad que eso conlleva, y seguiremos disponiendo de la potencia de la librería Kakadu.

Las clases de la biblioteca Kakadu poseen, en sus comentarios, instrucciones para construir la interfaz Java apropiada. Estas instrucciones son in-

terpretadas por la aplicación *kdu_hyperdoc*, la cual es la encargada de crear las clases Java.

Todas las macros que existan en el código fuente de Kakadu, son exportadas a Java como variables "public final static" de la clase *Kdu_globals*.

Algunas funciones no pueden ser exportadas a Java, como por ejemplo, la sobrecarga de operadores. Por esta razón la mayoría de las clases disponen de alternativas para que puedan ser exportadas.

Cuando se emplean objetos Java que representan objetos nativos, es preciso tener en cuenta que la memoria de los objetos nativos no es liberada automáticamente por el recolector de basura. Por ello, Kakadu incluye en cada clase interfaz Java el método *Destroy*, cuya misión es destruir el objeto nativo al cual representa.

Existen varias formas para exportar los elementos nativos a Java. Los principales son los siguientes:

- *Reference*: Es aplicado a las clases, y es el método más comúnmente usado para exportar las clases a Java.

- *Interface*: Es aplicado a las clases y es usado en circunstancias especiales para evitar el uso ineficiente de la memoria, cuando un objeto C++ es internamente una interfaz de otro objeto C++ interno. Las clases exportadas de esta forma no contienen métodos virtuales, no contienen variables y no son derivadas.

- *Copy*: Es aplicado a las clases cuyos objetos resultantes son de tamaño conocido (no están derivadas ni poseen métodos virtuales) y pueden ser fácilmente pasadas por valor a cualquier función.

- *Callback*: Por defecto, y a menos que se indique lo contrario, todos los métodos de una clase exportada con *Reference* son exportados a Java. Estos métodos sin embargo poseen una debilidad, y es que si la clase a los que pertenecen es derivada en Java, y alguno de los métodos es sobrecargado con un método nuevo escrito en Java, si algún otro método nativo hacía uso de dicho método, no llamará a la nueva versión, como ocurre normalmente con la sobrecarga de métodos (en Java, para todos los métodos, y en C++, sólo para los métodos virtuales), sino que seguirá llamando a la versión nativa original. Esto se puede solucionar exportando el método que queramos como *Callback*. De esta forma, el código nativo podrá llamar a los métodos sobrecargados en Java. De momento, Kakadu establece una serie de restricciones con este tipo de exportación a métodos que acepten únicamente tipos de datos básicos

y devuelvan un tipo de dato básico, sin admitirse *arrays*. El uso de métodos *Callback* produce una sobrecarga de código y complejidad en la interfaz Java.

Las clases Java de Kakadu son agrupadas en el paquete *kdu_jni*. Cualquier aplicación Java que haga uso de estas clases necesitará para su correcto funcionamiento las librerías de enlace dinámico *jni_source* y la correspondiente al sistema base. En el caso de *Windows*, y la versión 4.0 de Kakadu, estas librerías son *jni_source.dll* y, o bien *kdu_v40R.dll* para la versión *Release*, o bien *kdu_v40D.dll* para la versión *Debug*.

Capítulo 5

Sistema desarrollado

5.1. Descripción

Se ha desarrollado para el presente proyecto un visualizador interactivo de imágenes JPEG2000 remotas. El visualizador ha sido desarrollado en Java, apoyándose mediante el API JNI en la biblioteca Kakadu para el procesamiento de las imágenes. Permite visualizar de forma progresiva e interactiva imágenes alojadas en cualquier servidor HTTP/1.1.

5.2. Justificación

Existe una gran demanda de aplicaciones que permitan visualizar imágenes de gran tamaño a través de Internet. JPEG2000, debido a sus características, es el estándar de compresión de imágenes ideal para este cometido.

En la Parte 9 del estándar se define el protocolo JPIP. Este protocolo define el esquema de comunicación en un sistema cliente/servidor para la transmisión de imágenes JPEG2000 de forma eficiente. Este protocolo, no obstante, presenta una serie de desventajas debido a las cuales no se ha empleado para realizar la aplicación del presente proyecto:

- Al ser un protocolo nuevo, requiere de un servidor específico. Por el momento, la única implementación disponible de un servidor JPIP está escrita para el sistema Windows, con lo cual, de necesitar instalar el servidor en otro sistema operativo diferente, habría que realizar las correspondientes modificaciones a dicha implementación. Además, no siempre se tiene la posibilidad de disponer de un servidor específico

con acceso a Internet, sino que únicamente se dispone de un servidor HTTP para alojar el contenido Web de nuestro sitio y no se tiene control total sobre el mismo.

- El protocolo ofrece la posibilidad de mantener *caches*, tanto en el servidor como en el cliente, con el fin de mejorar el rendimiento. Este sistema de *cache* ya se dispone en el protocolo HTTP, en el cual, tanto los *proxys* como los servidores mantienen una *cache*. Con el protocolo JPIP, la *cache* de los *proxys* resulta inservible y descartada, ya que generalmente el proxy no entenderá el contenido de los mensajes y no realizará un *caching* eficiente.

Por estos motivos se ha descartado la opción de usar el protocolo JPIP para transmitir las imágenes entre el cliente y el servidor. Se ha optado por el protocolo HTTP, en su versión 1.1, el principal protocolo empleado hoy día en Internet. Es por ello que no se requerirá trabajar con un servidor específico, sino que simplemente haremos uso de algún servidor HTTP ya implementado, y posiblemente implantado. Además, podremos alojar nuestras imágenes en cualquier servidor que disponga de acceso HTTP, aunque no tengamos el control total del mismo (porque sencillamente no sea nuestro, sino un servidor contratado a algún proveedor de servicios de Internet).

El protocolo HTTP/1.1 posee las siguientes características que lo hacen perfecto para nuestro cometido:

- Es el protocolo que se usa en Internet para el WWW. Cualquier página Web es accedida a través de un servidor HTTP. Es por ello que el protocolo es algo ya testeado y comprobado, y no es algo todavía novedoso y poco utilizado, como ocurre con el protocolo JPIP.

- Al ser un protocolo tan difundido, no es difícil encontrar una implementación de un servidor HTTP para cualquier tipo de plataforma. Esto no ocurre con el protocolo JPIP, para el que la única implementación conocida está diseñada para correr en sistemas Windows, aún teniendo en cuenta que la mayoría de los servidores Web corren en sistemas Unix.

- Ofrece un sistema de *cache* eficiente y ampliamente testado. Este sistema resulta invisible para el cliente, aunque se tiene cierto acceso al mismo. El sistema de *cache* que ofrece el protocolo JPIP es similar al ofrecido por el protocolo HTTP, con la diferencia de que no explota la *cache* de los *proxys* que puedan existir, ya que éstos, a menos

que se realice una implementación específica, no entenderán el nuevo protocolo.

Este no es el primer desarrollo que intenta explotar la combinación del estándar de compresión de imágenes JPEG2000 y el protocolo HTTP. Saching Deshpande y Wenjun Zeng [2] proponen una solución: para cada imagen, a la cual se requiere acceso a través del visualizador, es preciso construir un índice, que se almacena en un archivo independiente. Es posible construir ese índice con varios niveles de complejidad, de forma que cuanto más complejo sea, más espacio ocupa, pero más eficiente resulta la comunicación; y cuanto menos complejo sea, menos espacio ocupa, pero menos eficiente es la comunicación. El cliente leería el índice asociado a una imagen, antes de leer ni un sólo byte de ella, con el fin de localizar aquellas partes de la misma que necesita. Esta solución presenta los siguientes inconvenientes:

- Necesita la implementación de una pequeña aplicación adicional para la construcción de los índices.
- El archivo índice ocupará un espacio en disco, en función de la complejidad elegida para su construcción, con información bastante redundante, ya que mucha de la información almacenada en dicho archivo, se encuentra ya en el archivo de imagen.
- El archivo de índice debe estar ligado inexorablemente al archivo de la imagen. Esto hace que, si modificamos la imagen por cualquier motivo, debamos actualizar el archivo de índice correspondiente.
- EL cliente debe leer todo el archivo índice, antes de poder presentar ninguna visualización de la imagen. Esto, para imágenes grandes, hace que el tiempo de espera inicial sea bastante elevado, independientemente de la complejidad del archivo índice, y aunque se quiera visualizar únicamente una porción de la imagen a una resolución reducida.

La Parte 9 del estándar, no sólo define el protocolo JPIP, sino que define además una forma de incluir un índice en el archivo de imagen, pensando en aplicaciones como la de Saching Deshpande y Wenjun Zeng, o la desarrollada para el presente proyecto. Esto haría que el índice estuviese encapsulado en el archivo de imagen y no en un archivo independiente como proponen Saching Deshpande y Wenjun Zeng. Esto soluciona únicamente los dos

últimos inconvenientes mencionados anteriormente, pero los otros dos continúan existiendo. No sólo eso, sino que la solución que se propone provoca un aumento excesivo del tamaño que ocupa el archivo de imagen.

Como se pudo ver en la sección 2.3, el indexado se facilita para los archivos de la familia JP2. Estos archivos están organizados en cajas, cada una de las cuales puede contener a su vez, subcajas. Para la reconstrucción de una área determinada de una imagen un visualizador necesitaría conocer dónde se encuentran en el archivo de imagen los paquetes de los precintos asociados, y qué tamaño ocupan, para poder leerlos. Para la construcción del índice, según propone la Parte 9, la información de la posición y longitud de todos los paquetes de todos los precintos se almacena en la caja *Precinct Packet Index Table*. En ella, cada paquete ocupa como mínimo 8 bytes, cuatro bytes para el offset y cuatro para la longitud. Teniendo en cuenta que una imagen JPEG2000 grande construida con la idea de ser visualizada remotamente (con un tamaño de precinto pequeño, un número elevado de capas de calidad, etc.), puede contener una cantidad enorme de paquetes, esta sobrecarga de espacio es considerable. Eso sin tener en cuenta el espacio del resto de las cajas necesarias para el índice (y del espacio que la propia estructura de la caja necesita, aunque esté vacía). La versión simple del archivo de índice, en la propuesta de Saching Deshpande y Wenjun Zeng, se construye almacenando los marcadores del *codestream* de la imagen que facilita el acceso aleatorio al mismo, como por ejemplo TLM, PLT, PPM, etc. En el caso de la longitud de los paquetes, ésta es almacenada en los paquetes PLT o PLM. El offset no es necesario almacenarlo realmente, ahorrándonos esos 4 bytes, porque todos los paquetes de un *tile-part* son consecutivos, y la posición de comienzo de cada *tile-part* se sabe mediante el marcador TLM. En los marcadores PLT y PLM, la longitud L de cada paquete, como se vio en la sección 1.4, ocupa $\left\lceil \frac{B_L}{7} \right\rceil$ bytes, donde B_L es el mínimo de bits requeridos para representar L. Esto quiere decir que L, en el peor de los casos, ocupará 5 bytes, pero sólo en el peor de los casos. Normalmente, la longitud de un paquete de una imagen con un tamaño de precinto pequeño y con un nivel elevado de capas de calidad, ocupará como media 2 bytes, ahorrándonos otros 2 bytes por paquete.

La solución adoptada en el presente proyecto no hará uso del protocolo JPIP, sino del protocolo HTTP/1.1 para la transmisión de la imagen. No empleará el indexado propuesto por la Parte 9 del estándar, ni tampoco la solución propuesta por Saching Deshpande y Wenjun Zeng, basándonos en lo anteriormente expuesto. La aplicación desarrollada permitirá visualizar progresivamente un imagen JPEG2000 remota, almacenada en un archivo

J2C, mediante el protocolo HTTP/1.1, sin necesidad de índice alguno.

Además de los protocolos JPIP y HTTP, Jin Li y Hong-Hui Sun [5] propusieron un protocolo especial llamado *Vmedia*.

5.3. Formato del archivo de imagen

Aunque existen varios tipos de formato para los archivos de imagen JPEG2000, la aplicación sólo trabajará con el formato J2C. En realidad, un archivo J2C no posee ningún formato interno, sino que simplemente almacena el *codestream* de una imagen JPEG2000. Las principales razones de utilizar este formato son las siguientes:

- La lectura de la imagen es mucho más sencilla que para el caso del resto de formatos. Por ello, la lectura será también mucho más rápida. Para el caso del formato JP2, por ejemplo, el *codestream* está almacenado en una caja, a la cual se accede una vez leídas las cajas que le preceden. No obstante, se comentará cómo poder ampliar la aplicación desarrollada para que soporte este tipo de archivos.

- Al no incluir otras estructuras, el archivo de imagen ocupa lo que ocupe el *codestream*. Se minimiza por ello la cantidad de bytes a transmitir, limitándose únicamente a lo estrictamente necesario.

Por contra, la adopción del formato J2C, supone la desventaja de no poder incluir otro tipo de información que sí es posible incluir en los otros formatos, como por ejemplo, la paleta de colores, si resultara necesaria. Esto nos limita a que las imágenes que empleemos no requieran información adicional, sino que el *codestream* sea autosuficiente. No podríamos, entonces, emplear imágenes con paleta, ya que la paleta de colores únicamente es posible incluir dentro de una caja en el formato JP2 o similar. No obstante, tampoco es una limitación preocupante, ya que generalmente las imágenes grandes, a las cuales va orientada nuestra aplicación, no son imágenes con paleta.

Nuestra aplicación será capaz de visualizar cualquier tipo de imagen J2C, pero sólo lo hará correctamente si la compresión de la imagen ha sido realizada cumpliendo una serie de requisitos:

- Debe poseer un único *tile*.

- Debe tener una progresión RLCP.

- Debe poseer un *tile-part* por resolución.
- Debe poseer uno o más marcadores TLM.
- Debe poseer uno o más marcadores PLT en la cabecera de cada *tile-part*.

El hecho de tener un solo *tile* es algo común en las imágenes, ya que el uso de varios *tiles* empeora la calidad de la imagen. El *tiling* sólo es recomendado para cierto tipo de aplicaciones, que no es el caso de la desarrollada.

El requisito de tener la progresión RLCP va asociado al de poseer los marcadores PLT y al de tener un *tile-part* por resolución. Los marcadores PLT son los que almacenan la longitud de los paquetes de un *tile-part*. Si hubiese un solo *tile-part* general, habría un marcador PLT (podría haber más, contiguos, pero generalmente, habrá sólo uno por *tile-part*) general con la información de todos los paquetes. Esto supondría que el visualizador, antes de poder recuperar cualquier conjunto aleatorio de paquetes de la imagen, necesitaría leer el marcador PLT completo, el cual puede tener un tamaño considerable, aumentando el tiempo necesario. Esto no sería eficiente, ya que generalmente el conjunto de paquetes que requiere el visualizador corresponden a un área de la imagen a una resolución determinada. Por ello, el *tile* se divide en tantos *tile-parts* como resoluciones tenga la imagen. En la cabecera de cada *tile-part* se incluye el correspondiente marcador PLT. De esta forma, cuando el visualizador requiera un área de la imagen a una resolución determinada, se tendrá que leer los marcadores PLT del *tile-part* asociado a esa resolución y el de todos los *tile-parts* de las resoluciones inferiores, si no lo ha hecho ya.

El dividir el *tile* en *tile-parts* por resolución obliga a asumir o bien una progresión RLCP o bien una RPCL. En nuestro caso se ha optado por escoger la progresión RLCP, debido a que la aplicación desarrollada leerá los paquetes tal y como específica esta progresión, por capa, por componente y por precinto. Además, en el proceso de lectura por HTTP mediante *byte-ranging*, habrá paquetes que aparezcan contiguos, para una región de la imagen determinada, con lo cual, en vez de solicitar al servidor HTTP esos paquetes uno a uno, se podrán juntar en la misma petición, con el consiguiente aumento en la velocidad de transmisión.

Estos son los requerimientos necesarios para que el visualizador funcione correctamente. Si la imagen no cumple estos requisitos, se deberá leer la imagen entera para poder visualizarla.

Después existen una serie de recomendaciones para mejorar el rendimiento del visualizador:

- Usar precintos de pequeño tamaño, de 128x128 o 64x64. El motivo es que el visualizador, para poder visualizar una región determinada de la imagen, leerá todos los paquetes correspondientes a los precintos necesarios para reconstruir dicha región. Por lo tanto, cuanto más pequeños son los precintos en la imágenes, más fina es la granularidad, haciendo que el tamaño de los paquetes a leer por el visualizador sea menor.

- Utilizar un número elevado de capas de calidad. El visualizador va repitiendo el proceso de reconstrucción y visualización del área de la imagen requerido por el usuario desde que empieza a solicitar el primer paquete necesario, hasta que llega la información del último. Por ello, y debido a que la transmisión de los paquetes se realiza por capas, cuantas más capas existan, más se reduce el número de bytes necesarios a leer por el visualizador para que el usuario pueda ver de forma aceptable la imagen.

5.4. Estructura interna de la aplicación

La estructura de la aplicación de divide en un primer nivel en dos grandes sistemas, el sistema de visualización y el sistema de gestión de imagen. Empleamos el concepto de sistema para designar a una parte del programa con una funcionalidad definida, y que puede estar formada por una o más clases Java. No se describirán todas las clases que forman la aplicación, obviando las que sirven solamente de apoyo a la estructura principal.

La figura 5.1 muestra estos dos sistemas principales. En ella se puede apreciar que el sistema de gestión de imagen posee a su vez dos subsistemas, el sistema de comunicación y el sistema de descompresión.

En las figuras representadas en esta sección, tanto la clase como el sistema se representan mediante un rectángulo en el que va incluido el nombre. Los hilos de ejecución que pueda lanzar una determinada clase son también representado mediante rectángulos. Las entidades externas a la aplicación, como el usuario o el servidor, son representadas por una elipse. Las flechas muestran la relación existente entre dos clases o sistemas. Las flechas pueden ir acompañas de un pequeño texto aclaratorio del tipo de relación que es, o de la información que se comparte en dicha relación.

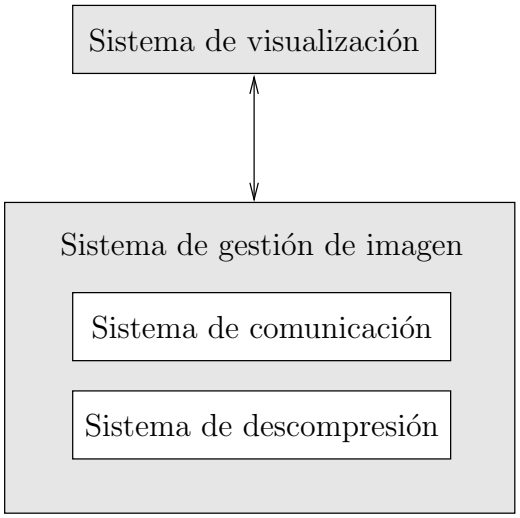

Figura 5.1: Estructura de la aplicación.

5.4.1. Sistema de visualización

El sistema de visualización está formado por todas las clases Java encargadas de construir la interfaz de usuario.

La clase encargada de construir la casi totalidad de la interfaz de usuario es la clase *ImageWindow*, que deriva de la clase *JFrame*. Se encarga de crear la ventana, los botones, las barras de desplazamiento, etc.

La zona donde se visualiza cualquier imagen, o un área de la misma, es administrada por la clase *ImagePanel*. Debido a que tenemos la posibilidad de incluir una vista en miniatura como ventana interna, un *JInternalFrame*, la clase *ImagePanel* debe derivar de la clase *JDesktopPane*. La clase *ImagePanel* es la encargada de permitir la interacción del usuario con la imagen, permitiéndole al usuario hacer *zoom* o moverse por el contenido de la imagen. Las operaciones que realiza el usuario con la imagen, la clase *ImagePanel* las traduce al par (ROI, resolución), es decir, que resolución se requiere, ya que puede cambiar al hacer el usuario *zoom*, y la región de interés (ROI, *Region Of Interest*) de la imagen a visualizar en la resolución dada. Nótese que la posibilidad ofrecida al usuario de hacer *zoom* está directamente vinculada a los niveles de resolución de la imagen, suponiendo que el hacer *zoom* consiste en moverse por dichos niveles. El par (ROI, resolución) es transmitido al sistema de gestión de imagen. El sistema de gestión de imagen, conforme vaya teniendo nuevas porciones de la ROI requerida, se lo irá notificando al

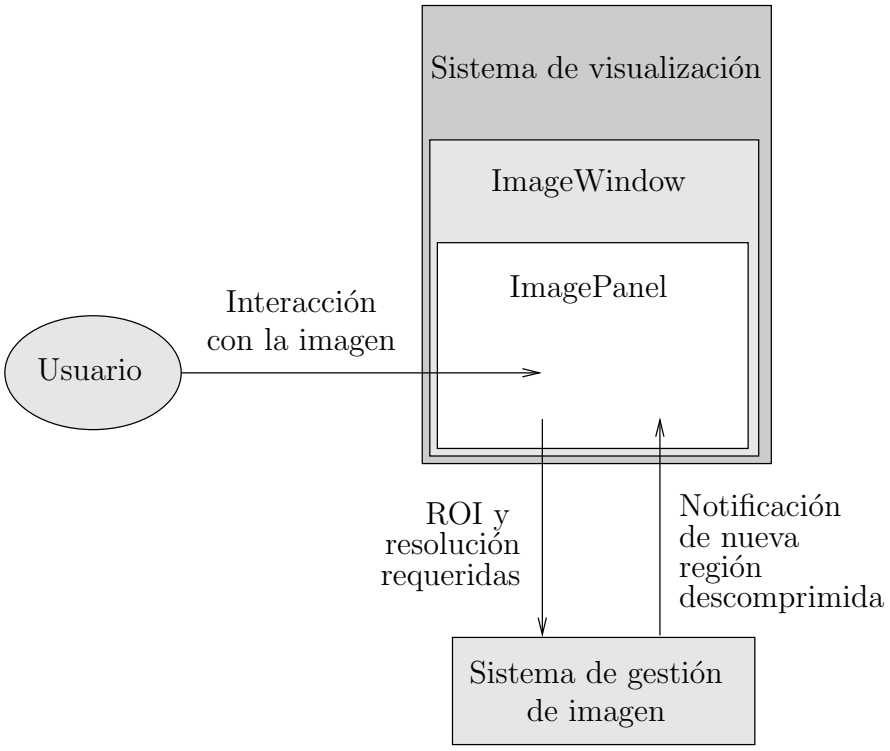

Figura 5.2: Estructura del sistema de visualización.

sistema de visualización, en especial a la clase *ImagePanel*, para que actualice la visualización en pantalla. La clase *ImagePanel* implementa la interfaz *ImageObserver*, a través de la cual el sistema de gestión de imagen puede realizarle las correspondientes notificaciones. Para ello, deberá haber llamado la clase *ImagePanel* al método *addImageObserver()* de la clase principal del sistema de gestión de imagen, la clase *JP2Image*.

La figura 5.2 representa de forma general la estructura del sistema de visualización, y su relación con el resto de sistemas o entidades.

5.4.2. Sistema de gestión de imagen

El sistema de gestión de imagen es el encargado de la gestión y tratamiento de la imagen a petición del sistema de visualización. El sistema de visualización le comunica a este sistema qué ROI y resolución requiere el usuario, y éste lleva a cabo las tareas necesarias para conseguirlo, notificando

al sistema de visualización en cuanto existan resultados.

El sistema de gestión de imagen es representado únicamente por la clase *JP2Image*, encargada de toda la gestión de la imagen.

La clase *JP2Image* incluye además dos subsistemas, el sistema de comunicación y el sistema de descompresión.

Al recibir la clase *JP2Image* una petición del sistema de visualización, esta analizará la estructura del *codestream* de la imagen, empleando la clase *Kdu_codestream*, para determinar qué paquetes son necesarios leer del servidor. De los paquetes necesarios, se descartan aquellos que ya existan en la *cache*, mantenida por la clase *JP2Cache*, que a su vez deriva de la clase *Kdu_cache*. Esta clase será la empleada para inicializar la clase *Kdu_codestream* y ésta última la empleará para acceder a los datos del *codestream* de la imagen. Los paquetes que son necesarios se comunican al sistema de comunicación para que lleve a cabo las peticiones HTTP correspondientes al servidor. El sistema de comunicación va almacenando directamente los paquetes recibidos del servidor en la *cache*, en la clase *JP2Cache*.

La clase *JP2Image* posee un objeto de la clase *BufferedImage* donde almacenar la reconstrucción de la ROI.

Al establecer la ROI y la resolución, no sólo se determinan los paquetes necesarios a leer, sino que se determinan también que regiones son necesarias descomprimir, basándose en la ROI anterior. Estas regiones son comunicadas al sistema de descompresión para que las vaya descomprimiendo. Para la descompresión, este sistema necesita del *codestream* de la imagen, cuyo contenido irá siendo actualizado por el sistema de comunicación. En el objeto de la clase *Kdu_codestream* se establecerá que resolución es la necesaria.

La figura 5.3 representa de forma general la estructura del sistema de gestión de imagen, y su relación con el resto de sistemas o entidades.

5.4.3. Sistema de descompresión

El sistema de descompresión es un subsistema del sistema de gestión de imagen. Su misión es descomprimir aquellas regiones de la imagen que éste último requiera. La clase que identifica al sistema de descompresión es la clase *JP2Render*.

La figura 5.4 representa de forma general la estructura del sistema de descompresión, y su relación con el resto de sistemas o entidades.

Cada región se especifica mediante un objeto de la clase *Region* y se almacena en una lista, representada por un objeto de la clase *RegionsList*, la cual deriva a su vez de la clase *ArrayList*. Ésta es una lista FIFO.

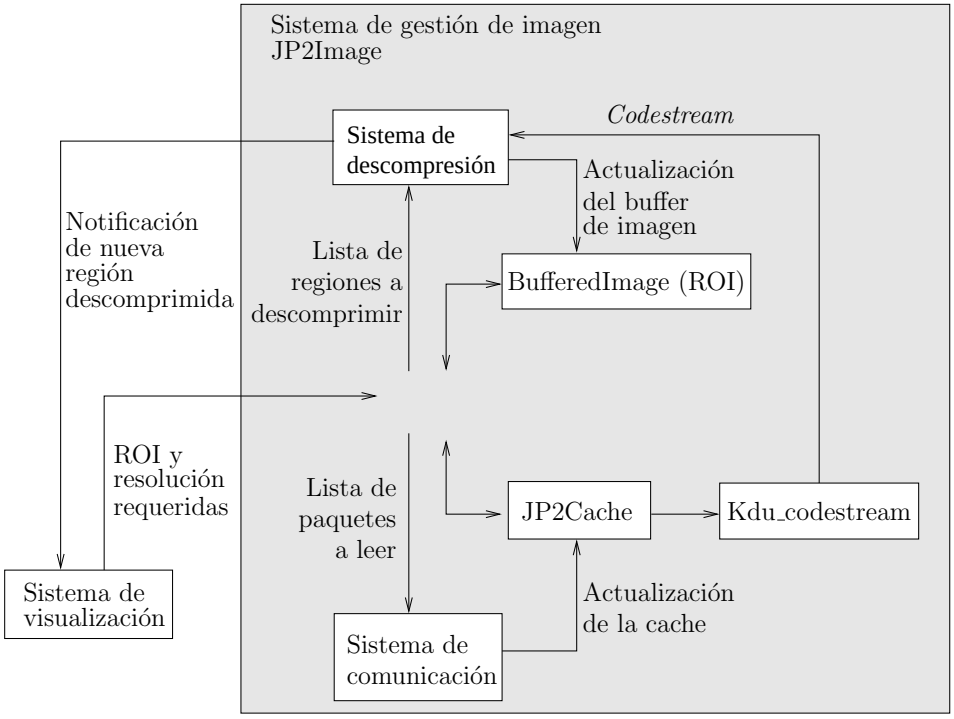

Figura 5.3: Estructura del sistema de gestión de imagen.

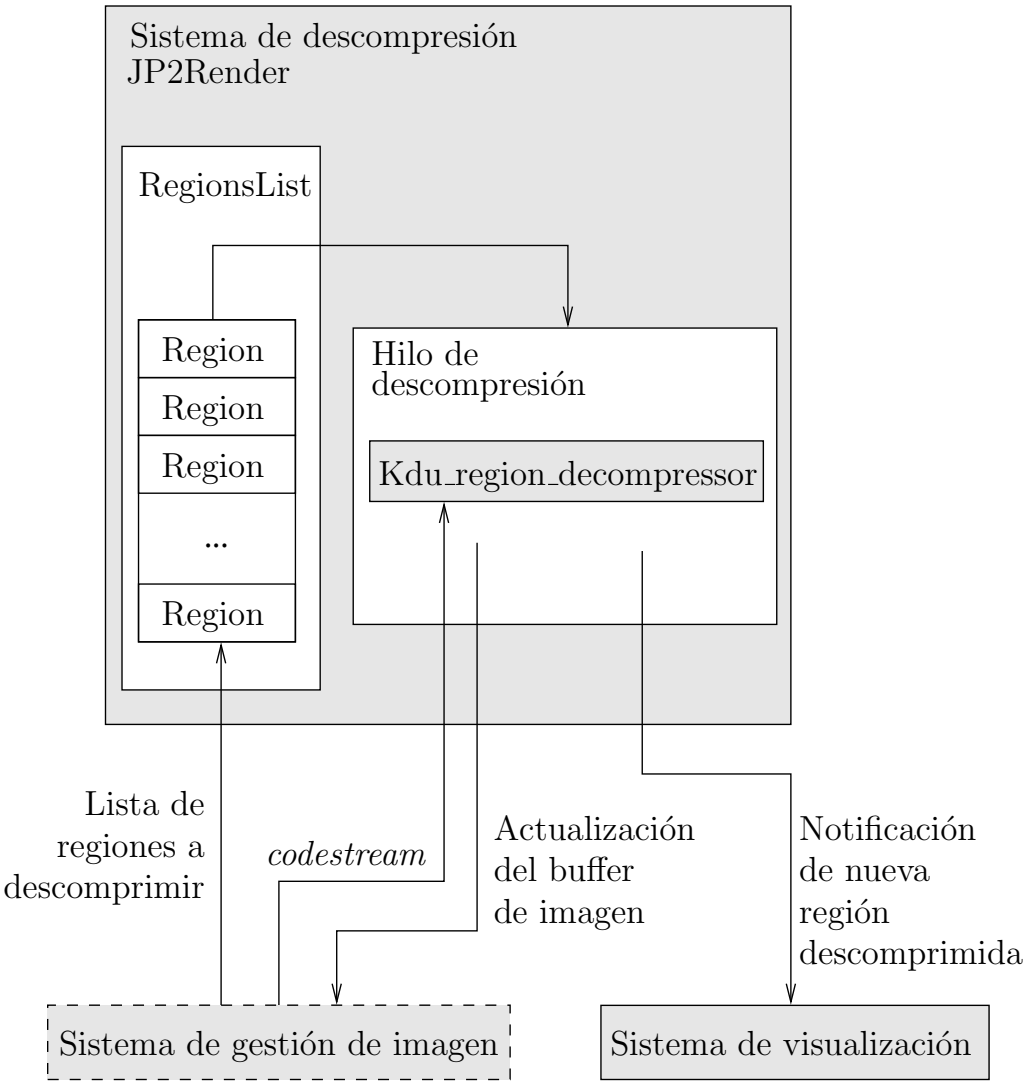

Figura 5.4: Estructura del sistema de descompresión.

La clase *JP2Render* lanza un hilo de ejecución que permanecerá continuamente esperando a que haya regiones para descomprimir en la lista de la clase *RegionsList*. En el momento en el que haya una nueva región a descomprimir, empleará para su descompresión un objeto de la clase *Kdu_region_decompressor*, y el *codestream* de la imagen, al cual se accede mediante el objeto de la clase *Kdu_codestream*, de la clase *JP2Image*. Al sistema de descompresión no es necesario pasarle la resolución, ya que empleará la establecida en *Kdu_codestream*.

Conforme se vayan obteniendo porciones de cada región, éstas se irán almacenando directamente en el *buffer* de la imagen, en el objeto de la clase *BufferedImage* de la clase *JP2Image*, a la vez que se lo notifica al sistema de visualización para realice la correspondiente actualización.

5.4.4. Sistema de comunicación

El sistema de comunicación es el encargado de leer a través del protocolo HTTP/1.1 los paquetes necesarios para construir la ROI requerida a la resolución indicada. La clase que identifica al sistema de comunicación es la clase *JP2Reader*.

La figura 5.5 representa de forma general la estructura del sistema de comunicación, y su relación con el resto de sistemas o entidades.

Los paquetes que requiere el sistema de gestión de imagen son indicados mediante objetos de la clase *PacketData*. Estos objetos se almacenan en una lista FIFO representada por un objeto de la clase *PacketsList*, la cual deriva de la clase *ArrayList*.

La clase *JP2Reader* lanza dos hilos de ejecución, uno para escritura y otro para lectura. El de escritura va recorriendo la lista de la clase *PacketsList* y enviando al servidor las peticiones correspondientes. El hilo de lectura va leyendo paquete a paquete del servidor según especifique la información de cada paquete en la lista de la clase *PacketsList*. Conforme se vaya recibiendo cada paquete, éste se va almacenando directamente en la *cache* de la clase *JP2Image*.

La comunicación con el servidor HTTP se realiza mediante un objeto de la clase *HTTPReader*. Esta clase permite realizar *byte-ranging* con el protocolo HTTP/1.1 y admite mensajes *"chunkeados"*.

5.5. Funcionamiento

En la sección anterior se dio una breve descripción de la estructura del código de la aplicación desarrollada, así como una pequeña introducción

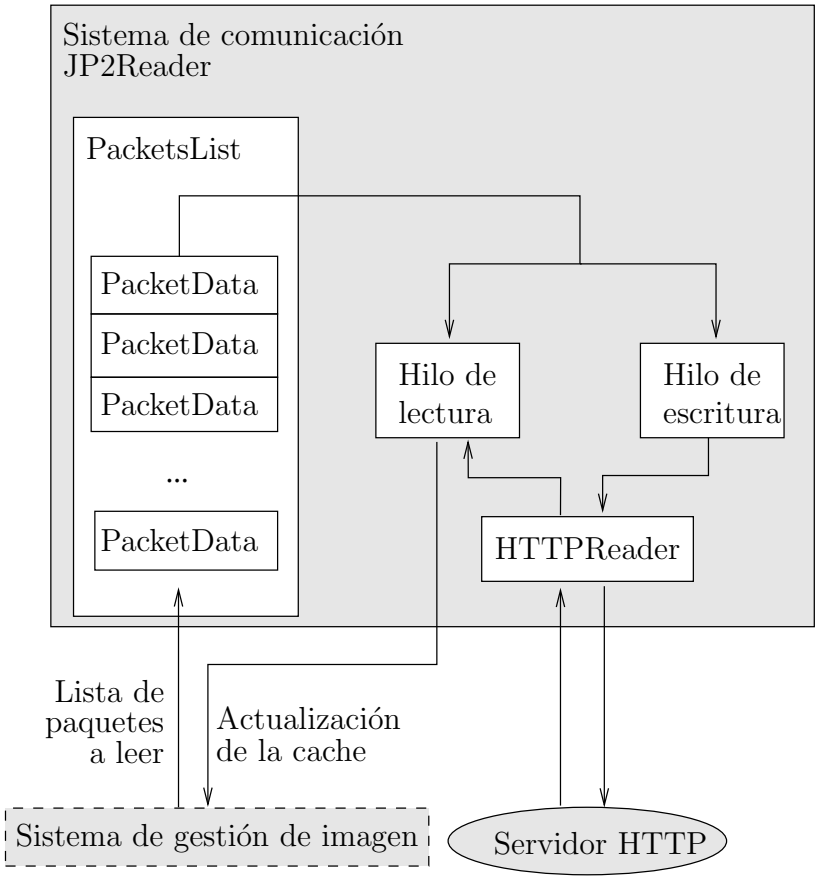

Figura 5.5: Estructura del sistema de comunicación.

al funcionamiento de cada uno de los sistemas que la componen. En esta sección se entrará más en detalle en el funcionamiento de cada una de las tareas que puede llevar a cabo la aplicación. Cada tarea es desarrollada por uno o varios de los sistemas de la aplicación.

5.5.1. Apertura de la imagen

Cuando el usuario introduce la dirección de una nueva imagen y pulsa la tecla INTRO, o simplemente pulsa sobre el botón Recargar, se procede a realizar la apertura de la imagen cuya URL viene indicada por la cadena que haya en el cuadro de texto *Dirección*.

Si la cadena no forma una URL correcta, o el protocolo indicado no es el HTTP, se asumirá que se desea visualizar una imagen local, cuyo proceso a seguir no se describirá en la presente documentación, al no ser el cometido del proyecto. Si la cadena forma una URL correcta y el protocolo indicado es el HTTP, se procederá a la apertura normal de la imagen remota indicada. Este proceso no ha sido referenciado en las figuras de la sección anterior.

El proceso de apertura de imagen comienza al llamar el objeto de la clase *ImagePanel* al método *open()* de la clase JP2Image, es decir, el sistema de visualización le solicita al sistema de gestión de imagen la apertura de una nueva imagen. A este método se le pasa la cadena (*String*) con la URL de la imagen. El código del método *open()* crea un objeto nuevo de la clase *Kdu_codestream*, y otro también de la clase *JP2Cache*. A continuación, se crea provisionalmente un objeto de la clase *HTTPReader* que nos permitirá leer la cabecera principal del *codestream* de la imagen remota. Esta lectura no se realizará en un hilo a parte, puesto que no es posible llevar a cabo ninguna tarea posterior sin acabar antes la lectura de la cabecera principal, ni empleando *pipelining*, ya que cada lectura depende de la anterior.

El algoritmo de la lectura de la cabecera principal es como el siguiente:

```
marcador = httpReader.read(2);
si (marcador != SOC) error;
cache.addToMainHeader(marcador);

repetir {
  marcador = httpReader.read(2);
  si (marcador == SOT) salir del bucle;
  si no {
    marcadorLen = httpReader.read(2);
```

```
    si (marcador == COM) o
       (marcador == PLM) o
       (marcador == PPM) httpReader.skip(marcadorLen-2);
    si no {
      datos = httpReader.read(marcadorLen-2);
      si (marcador == TLM) tlmRecord.addSegment(datos);
      si no {
        cache.addToMainHeader(marcador);
        cache.addToMainHeader(marcadorLen);
        cache.addToMainheader(datos);
    }
  }
}

cache.completeMainHeader()
```

En el algoritmo, se asume que las variables *marcador* y *marcadorLen* son del tipo *byte[]*, el objeto *httpReader* es de la clase *HTTPReader*, el objeto *tlmRecord* es de la clase *TLMRecord* y el objeto *cache* de la clase *JP2Cache*.

La clase *HTTPReader* mantiene un puntero de archivo, inicializado a cero, de forma que permite realizar lecturas consecutivas, como se indica en el algoritmo. Estas lecturas se realizan mediante el método *byte[] read(int nbytes)*.

Dicho método devuelve un *array* (*byte[]*) con los datos leídos. Esta lectura incluye la conexión con el servidor HTTP, si no existe la conexión aún, el envío de la petición HTTP con el *byte-range* apropiado, y la lectura de la respuesta.

La clase *JP2Cache* deriva de la clase *Kdu_cache*. Como se indicó en el capítulo dedicado al paquete Kakadu, esta clase permite disponer de una *cache*. La *cache*, al estar pensada para desarrollar sistemas clientes/servidor empleando el protocolo JPIP, está organizada en *data-bins*. El método de la clase *Kdu_cache* que permite añadir contenido en bytes a un *data-bin* determinado es:

```
void Add_to_databin(int databin_class, long codestream_id,
                    long databin_id, byte[] data, int offset,
                    int num_bytes, boolean is_final,
                    boolean add_as_most_recent,
                    boolean mark_if_augmented)
```

El parámetro *databin_class* indica qué tipo de *data-bin* es, pudiendo tomar el valor *KDU_MAIN_HEADER_DATABIN* para la cabecera principal, *KDU_TILE_HEADER_DATABIN* para la cabecera de un *tile* o *KDU_PRECINCT_DATABIN* para los datos de un precinto; *codestream* contiene el identificador de *codestream*, cuando trabajamos con varios *codestreams* simultáneamente, que no es nuestro caso, por el que siempre valdrá cero; *databin_id* identifica al *data-bin*, útil únicamente para el caso de los *data-bins* de las cabeceras de los *tiles* y de los precintos, ya que para el caso de la cabecera principal valdrá cero; *data* contiene los datos a añadir al *data-bin*; *offset* posee la posición dentro del *buffer* del *data-bin* a partir de la cual copia los datos de data; *num_bytes* contiene el número de bytes a copiar; *is_final* indicará si el *data-bin* se completa al añadir los datos de data o no. El resto de parámetros no los emplearemos y los dejaremos con el valor *false*.

El uso de este método es un poco engorroso, por lo que la clase JP2Cache encapsula las llamadas a este método en una serie de métodos, cuya implementación sería:

```
void addToMainHeader(byte datos[])
{
  void Add_to_databin(
    KDU_MAIN_HEADER_DATABIN, 0, 0,
    datos,
    Get_databin_length(KDU_MAIN_HEADER_DATABIN, 0, 0, null),
    datos.length,
    false,
    false, false);
}

void completeMainHeader()
{
  void Add_to_databin(
    KDU_MAIN_HEADER_DATABIN, 0, 0,
    null,
    Get_databin_length(KDU_MAIN_HEADER_DATABIN, 0, 0, null),
    0,
    true,
    false, false);
}
```

```
void addToTileHeader(byte datos[], int tileID) {
  void Add_to_databin(
    KDU_TILE_HEADER_DATABIN, 0, tileID,
    datos,
    Get_databin_length(KDU_TILE_HEADER_DATABIN, 0, 0, null),
    datos.length,
    false,
    false, false);
}

void completeTileHeader(int tileID) {
  void Add_to_databin(
    KDU_TILE_HEADER_DATABIN, 0, tileID,
    null,
    Get_databin_length(KDU_TILE_HEADER_DATABIN, 0, 0, null),
    0,
    true,
    false, false);
}

void addToPrecinct(byte datos[], int precID) {
  void Add_to_databin(
    KDU_PRECINCT_DATABIN, 0, precID,
    datos,
    Get_databin_length(KDU_PRECINCT_DATABIN, 0, 0, null),
    datos.length,
    false,
    false, false);
}

void completePrecinct(int precID) {
  void Add_to_databin(
    KDU_PRECINCT_DATABIN, 0, precID,
    null,
    Get_databin_length(KDU_PRECINCT_DATABIN, 0, 0, null),
    0,
    true,
    false, false);
}
```

El algoritmo mostrado anteriormente sirve para leer la cabecera principal del *codestream* y almacenarlo en el *data-bin* correspondiente de la *cache*. Nótese que esta lectura salta los marcadores PPM, PLM y COM y no los incluye en la *cache*. El motivo del marcador COM es obvio, al contener un mero comentario, de principio sin función alguna. Los marcadores PPM Y PLM, no son utilizados en la aplicación, y su inclusión o no influye en el rendimiento de la biblioteca de Kakadu, por lo que son ignorados. Además, suelen tener un tamaño considerable.

El marcador TLM tampoco es incluido en la *cache*, pero sí leído y almacenado en un objeto de la clase *TLMRecord*. La clase *TLMRecord* permite almacenar los marcadores TLM de un *codestream* para después poder ser procesados, con el fin de obtener la información de cada *tile-part*, esto es, el *offset* y la longitud de cada uno. Para añadir el contenido de un marcador TLM a un objeto de esta clase empleamos el método *addSegment(byte data[])*. No se incluye ni los 2 bytes del marcador ni los dos bytes de la longitud del marcador.

El bucle de lectura de la cabecera principal finaliza cuando encuentra el primer marcador SOT. Entonces se le debe indicar a la *cache* que el *data-bin* de la cabecera principal está completo, ya que si no, el comportamiento de la clase *Kdu_codestream* no será el correcto. Este se realiza llamando al método *Add_to_databin()* con el parámetro *is_final* a *true* y *num_bytes* igual a cero. Esta llamada está encapsulada en el método *completeMainHeader()* de la clase *JP2Cache*.

Una vez leída la cabecera principal se procederá a leer las cabeceras de todos los *tile-parts* del primer *tile*. Recordemos que la imagen debería constar de un único *tile*.

El algoritmo para la lectura de las cabeceras sería similar al siguiente:

```
para i desde 0 hasta (tlmRecord.numTileParts(0)-1) {
  httpReader.seek(tlmRecord.getTilePartOffset(0, i));
  marcador = httpReader.read(2);
  si (marcador != SOT) error;
  marcadorLen = httpReader.read(2);
  httpReader.skip(marcadorLen-2);
  repetir {
    marcador = httpReader.read(2);
    si (marcador == SOD) salir bucle;
    si no {
      marcadorLen = httpReader.read(2)
      si (marcador == PPT) o
```

```
          (marcador == POC) o
          (marcador == COM) httpReader.skip(marcadorLen-2);
        si no {
          si (marcador == PLT) {
            pltRecord.addSegment(i, httpReader.getPos(), marcadorLen-2);
            httpReader.skip(marcadorLen-2);

          } si no {
            cache.addToTileHeader(0, marcador);
            cache.addToTileHeader(0, marcadorLen);
            cache.addToTileHeader(0, datos);
          }
        }
      }
    }
}

cache.completeTileHeader(0)
```

Con el método *numTileParts()* de la clase *TLMRecord* obtenemos el número de *tile-parts* del primer, y en teoría, único *tile*. Con el método *getTilePartOffset()* obtenemos el *offset* dentro del archivo donde comienza cada *tile-part*, para poder leer la cabecera asociada.

El *data-bin* correspondiente a la cabecera de un *tile* debe contener las cabeceras de todos los *tile-parts* que lo componen, quitando los marcadores SOT y SOD. En nuestro caso eliminamos también los marcadores COM, PPT y POC, ya que de todos modos serán ignorados.

El marcador PLT de cada *tile-part* tampoco se almacena, pero se guarda la referencia del mismo, esto es, se guarda el *offset*, la longitud y el número de *tile-part* al que pertenece. Se almacena en un objeto de la clase *PLTRecord*. Esta clase nos permitirá después analizar el contenido de dichos marcadores con el fin de obtener la longitud de cada paquete de cada *tile-part*.

Una vez leída la cabecera principal y la cabecera del primer *tile*, estamos en condiciones de inicializar el objeto de la clase *Kdu_codestream* con el objeto de la clase *JP2Cache*, que contiene la *cache* con todos los datos leídos. Esta inicialización la realizamos llamando al método *Create()*. Le indicamos a esta clase también que trabaje haciendo los recursos persistentes con el método *Set_persistent()*.

Finalmente, es necesario realizar una serie de comprobaciones para asegurarnos de que el *codestream* de la imagen es compatible con la aplicación,

es decir:

- Que tenga un sólo *tile*: El subsistema de parámetros de la biblioteca Kakadu, a través del marcador SIZ almacenado, nos da el tamaño de los *tiles*, (t_w, t_h), y el tamaño de la imagen, (i_w, i_h), de forma que habrá un sólo *tile* si $t_w \geq i_w$ y $t_h \geq i_h$.
- Que tenga un *tile-part* por resolución: No podemos asegurar esto con certeza, a menos que recorramos el archivo entero, pero pero podemos comprobar al menos que existen tantos *tile-parts*, como número de resoluciones haya.
- Que tenga la progresión RLCP: Accediendo al subsistema de parámetros de Kakadu, en concreto al marcador COD, a través de la clase *Kdu_codestream*, podemos determinar la progresión del *codestream* y comprobar si realmente es RLCP o no.
- Que disponga de marcadores TLM: El objeto de la clase *TLMRecord* debe tener al menos un marcador almacenado.
- Que disponga de marcadores PLT: El objeto de la clase *PLTRecord* debe tener al menos tantos marcadores como *tile-parts* existen.

Si el *codestream* no cumple alguno de estos requisitos, se le notificará al usuario un mensaje de error.

Una vez abierta la imagen, el sistema de gestión de imagen está preparado para aceptar peticiones del sistema de visualización. Inicialmente no se establece ninguna ROI, por lo que los subsistemas de descompresión y comunicación permanecen inactivos, a la vez que el *buffer* para la imagen del objeto de la clase *BufferedImage* estará vacío. Es necesario mencionar que antes de proceder con la apertura de la imagen, el sistema de gestión de imagen debe finalizar la ejecución del subsistema de descompresión y del subsistema de comunicación, si éstos están en ejecución.

5.5.2. Establecimiento de la ROI

Cuando el usuario interacciona con la imagen, a través del sistema de visualización, éste le comunica al sistema de gestión de imagen la ROI y la resolución necesaria.

Si no se ha establecido con anterioridad ninguna ROI, simplemente se crea el *buffer* de imagen, que será un objeto de la clase *BufferedImage*, con las dimensiones de la ROI. En el que caso de que ya se hubiera establecido

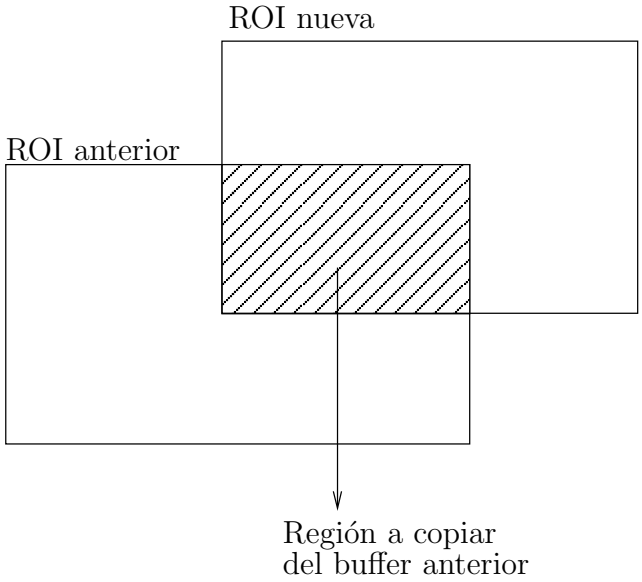

Figura 5.6: Cambio de la ROI.

una ROI anteriormente, se creará un *buffer* de imagen nuevo para la nueva ROI, pero, si la nueva ROI intersecciona con la anterior ROI, se copiará la intersección del *buffer* anterior al nuevo *buffer*. Este proceso se ilustra en la figura 5.6.

La región de interés y la resolución deseada se la comunicamos al objeto de la clase *Kdu_codestream* mediante el siguiente método:

```
void Apply_input_restrictions(int first_component,
                              int max_components, int discard_levels,
                              int max_layers,
                              Kdu_dims region_of_interest)
```

Este método, como su propio nombre indica, se emplea para restringir el acceso al *codestream* de la imagen, pudiendo así limitar el acceso a un número de resoluciones, a un numero de capas o a un numero de componentes. El parámetro *first_component* identifica el índice (empezando desde 0) de la primera componente a partir de la cual se permitirá el acceso; *max_components* es el número de componentes permitidas, empezando por la componente indicada en *first_component*, pero si es 0, todas las componentes son permitidas; *discard_levels* es el número de resoluciones superiores a las que no se

tendrá acceso; *max_layers* es el número máximo de capas de calidad permitidas y si es 0, todas las capas son permitidas; *region_of_interest* define una área a la cual restringir el acceso, no pudiendo descomprimir ninguna otra parte de la imagen.

En nuestro caso, llamaríamos a este método con los siguientes parámetros:

```
codestream.Apply_input_restrictions(0, 0, resolucion, 0, ROI)
```

Con ello permitiríamos el acceso a todos los componentes y todas las capas de calidad disponibles, restringiríamos el acceso a tantas resoluciones, empezando por la mayor, como la variable *resolucion* indique, y para dicha resolución, restringiríamos el acceso a la región establecida por la variable *ROI*, la cual asumimos que es de la clase *Kdu_dims*. Nótese que en la aplicación desarrollada, el concepto de resolución es diferente del concepto de resolución del estándar JPEG2000. En nuestro caso, la resolución cero sería la máxima resolución, que sería la imagen original. Al establecer la llamada anterior, cuando el sistema de descompresión reconstruya la imagen, se obtendrá una imagen del tamaño de ROI y con la resolución, según el estándar JPEG2000, de $D_{t,c} - resolucion$.

El sistema de gestión de imagen debe determinar qué áreas de la ROI debe reconstruir. En primer lugar, obtendrá la partición en regiones rectangulares de la región resultante al quitarle a la nueva ROI la parte de la intersección de ella misma con la anterior ROI. La figura 5.7 muestra un ejemplo gráfico de dicha partición. La partición crea regiones rectangulares, ya que el descompresor no admite otro tipo de regiones. Estas regiones las introduce en la lista del sistema de descompresión.

En la lista de regiones a descomprimir del sistema de descompresión, cada región se almacena como un objeto de la clase *Region*. Esta clase deriva de la clase *Rectangle*, heredando así las variables miembro *x*, *y*, *width* y *height*.

El sistema de gestión solicita a la lista de regiones del sistema de descompresión que actualice su contenido en función de la ROI. Esto se realizará llamando al método *actualize(Rectangle ROI)* de la clase *RegionsList*, clase a la que pertenece el objeto que representa la lista de regiones. Este método recorre la lista y actualiza cada región con la intersección de ella con la ROI. Si no intersecciona una región con la ROI simplemente es eliminada de la lista.

Una vez definidas las regiones que son necesarias descomprimir, es preciso obtener que paquetes son los que se necesitan leer con el fin de poder descomprimir dichas regiones. Para ello emplearemos el objeto de la clase

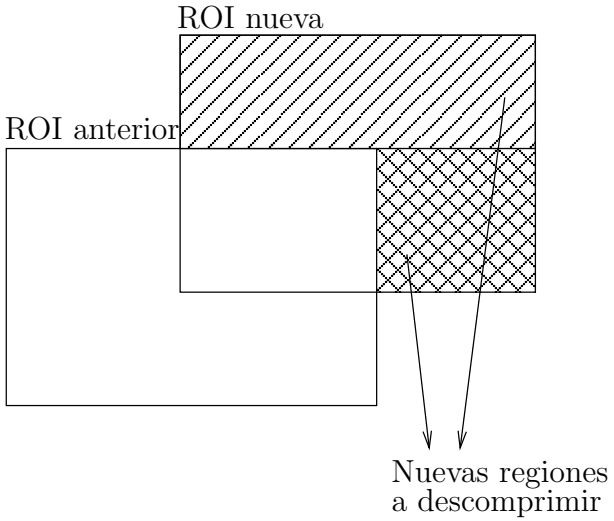

Figura 5.7: Regiones a descomprimir de una ROI.

Kdu_codestream. Al haber restringido el área a la cual tenemos acceso, mediante el método *Apply_input_restrictions()*, la clase *Kdu_codestream* no nos permitirá acceder a ninguna partición (*tile*, componente, resolución, precinto), que no forme parte de dicha área.

El algoritmo para determinar que paquetes serían los necesarios sería similar al siguiente:

```
Kdu_tile tile = codestream.Open_tile(new Kdu_coords(0, 0));

para l desde 0 hasta (numero_de_capas-1) {
  para c desde 0 hasta (tile.Get_num_components()-1) {
    Kdu_tile_comp tile_comp = tile.Access_component(c);
    para r desde 0 hasta (tile_comp.Get_num_resolutions()-1) {
      Kdu_resolution res = tile_comp.Access_resolution(r);
      Kdu_dims precintos = res.Get_valid_precincts();

      int pxInicio = precintos.Access_pos().Get_x();
      int pyInicio = precintos.Access_pos().Get_y();
      int pxFin    = precintos.Access_pos().Get_x() +
                     precintos.Access_size().Get_x() - 1;
      int pyFin    = precintos.Access_pos().Get_x() +
                     precintos.Access_size().Get_x() - 1;
```

```
      para py = pyInicio hasta pyFin
        para px = pxInicio hasta pxFin {

          // Incluir la informacion del
          // paquete (l, c, r, px, py)
        }
    }
  }
}

tile.Close();
```

En primer lugar, abrimos el único *tile* que debe existir en la imagen, con el método *Open_tile()* de la clase *Kdu_codestream*. A continuación vamos recorriendo todas las componentes, y todas las resoluciones de cada componente. En cada resolución de cada componente, solicitamos los precintos a los cuales tenemos acceso, mediante el método *Get_valid_precincts()*. Este devuelve un objeto de la clase *Kdu_dims*, el cual almacena el índice del precinto situado en la esquina superior izquierda del área permitida en el objeto de la clase *Kdu_coords* que devuelve el método *Access_pos()*. El número de precintos horizontales y verticales permitidos viene indicado en el objeto de la clase *Kdu_coords* devuelto por el método *Access_size()*. Cada precinto *(px, py)*, de una resolución r, de la componente c, de la capa l, define un paquete que es necesario para la descompresión, y cuya información se debe de incluir en la lista de paquetes del sistema de comunicación. Como se puede observar, la información de los paquetes se va incluyendo por capas de calidad.

La información de un paquete requerido se almacena en la lista de paquetes del sistema de comunicación en un objeto de la clase *PacketData*. Esta clase contiene tres variables miembro: *precinctID*, identificador de precinto al cual pertenece el paquete, *offset*, posición del comienzo del paquete dentro del archivo, y *length*, longitud en bytes del paquete.

El identificador del precinto es un concepto definido en la biblioteca Kakadu, necesario para referenciar el *data-bin* asociado a un precinto. Es decir, para utilizar método *Add_to_databin()*, anteriormente mencionado, para añadir información al *data-bin* asociado a un precinto determinado, necesitaremos indicar el identificador de dicho precinto en el parámetro *databin_id*. La forma de calcular dicho identificador no será necesario conocerla, ya que el método *Get_precinct_id()* de la clase *Kdu_resolution* nos devuelve el iden-

tificador de un precinto, pasándole la posición del mismo (en el algoritmo anterior, los valores *px* y *py*).

La posición y longitud de cada paquete las podremos obtener gracias a las referencias de los marcadores PLT que hemos almacenado en la apertura de la imagen. La clase *PLTRecord* es la que ha almacenado esta información, y será la que nos permita conocer la información de un determinado paquete. El método *int getOffset(int res, int layer, int comp, int px, int py)* de la clase *PLTRecord* nos devuelve el *offset* de un paquete, identificado por la resolución *res*, la capa *layer*, el componente *comp*, y la posición del precinto al que pertenece *(px, py)*. El método *int getLength(int res, int layer, int comp, int px, int py)* nos devuelve la longitud de paquete. Si llamamos a alguno de los dos métodos anteriores, y la clase *PLTRecord* no posee la información correspondiente, será porque todavía no ha leído los marcadores correspondientes a la resolución indicada. Si no los ha leído, construirá un objeto *HTTPReader* y los solicitará al servidor.

Al incluir la información de los paquetes requeridos en la lista del sistema de comunicación, se debe consultar la *cache* para ver qué paquetes se han leído ya. El *data-bin* de un precinto incluye todos los paquetes asociados al precinto, ordenados por capa de calidad. Sabiendo esto, obteniendo la longitud del *data-bin* asociado a un precinto determinado, podremos saber qué paquetes están ya incluidos, con el fin de no solicitarlos de nuevo al servidor.

Una vez concluidas todas estas etapas, el sistema de gestión de imagen pondrá en marcha el sistema de descompresión y el sistema de comunicación.

5.5.3. Lectura de los paquetes

La lectura de los paquetes indicados por el sistema de gestión de imagen la realiza el sistema de comunicación. Este sistema, identificado por la clase *JP2Reader*, crea un objeto de la clase *HTTPReader*, y dos hilos de ejecución, uno para la lectura y otro para la escritura.

El objeto de la clase *HTTPReader* nos permitirá comunicarnos con el servidor empleando el protocolo HTTP/1.1. Este clase nos permite leer *byte-ranges* de un determinado archivo y admite mensajes "*chunkeados*" del servidor. El constructor de la clase acepta como único parámetro una URL del recurso sobre el cual interaccionar, por lo que, para cada recurso o archivo remoto, habrá que crear un objeto de la clase *HTTPReader* diferente. Los métodos principales de esta clase son los siguientes:

- *void addRange(int offset, int length)*: Con esta función añadimos un

byte-range para ser incluido en la próxima petición HTTP que se haga con el método *request()*. Este *byte-range* se almacena en una lista interna de la clase.

- *byte[] readData(int numBytes)*: Lee un número de bytes igual a *numBytes*. El hilo que llama a este método se queda bloqueado hasta que llegue esa cantidad de bytes del servidor o se produzca algún error, o bien en forma de excepción o bien en forma de respuesta del servidor. El número de bytes indicado no incluye ninguna cabecera, y puede incluir además varios mensajes de respuesta consecutivos. Esto quiere decir que todas las respuestas del servidor se leerán secuencialmente, siendo transparente el hecho de poder existir cabeceras o mensajes "*chunkeados*". Para que haya datos para leer, se habrá tenido antes que llamar al método *request()* al menos una vez.

- *void request()*: Realiza una petición al servidor HTTP incluyendo todos los *byte-ranges* que se hayan definido anteriormente llamando al método *addRange()*. Este método simplemente realiza la petición, no espera a la respuesta, que será el cometido del método *readData()*. Al llamar a éste método se vacía la lista interna de *byte-ranges*.

- *byte[] read(int numBytes)*: Este método ya se ha visto anteriormente y simplemente llama en primer lugar al método *addRange()* para especificar el *byte-range* necesario, en función de la posición actual, después llama al método *request()* para realizar la petición y finalmente al método *readData()* para esperar la respuesta. Permite realizar lecturas secuenciales debido a que la clase *HTTPReader* mantiene un puntero para el recurso asociado. Este puntero es modificado por este método, pero no es modificado por el método *readData()*.

Al tener dividida la clase *HTTPReader* la petición de la lectura, podemos hacer *pipelining* en el protocolo HTTP/1.1. Por ello se crean dos hilos de ejecución, el de escritura, encargado de realizar las peticiones HTTP al servidor, y el de lectura, que se encarga de leer los mensajes de respuesta. Ambos hilos emplean un mismo objeto de la clase *HTTPReader* para comunicarse con el servidor. El hilo de escritura hará uso de los métodos *addRange()* y *request()*, mientras que el hilo de lectura hará uso del método *readData()*.

El hilo de escritura va leyendo los objetos de la clase *PacketData* almacenados en la lista de la clase *PacketsList*, lista en la cual deposita el sistema

de gestión de imagen los paquetes que necesita. Como vimos, la clase *PacketData* almacena el *offset* y la longitud (*length*) de cada paquete. El hilo de escritura juntará en un mismo byte-range aquellos paquetes que sean contiguos, haciendo que, al haber impuesto la progresión RLCP al *codestream* de las imágenes, pida los paquetes de los precintos línea a línea. Es necesario limitar el número de *byte-ranges* a incluir en una misma petición HTTP, ya que los servidores pueden tener limitada la longitud de cada cabecera (recordemos que los *byte-ranges* se especifican en HTTP mediante una cabecera ASCII). En la presente aplicación se ha limitado a 20 el número de *byte-ranges* por petición. El hilo de escritura finalizará cuando acabe de realizar todas las peticiones necesarias, conforme a lo incluido en la lista de la clase *PacketsList*, o cuando se produzca algún error. Es necesario mencionar que los paquetes de longitud igual a 1 no se solicitarán al servidor, debido a que en el contexto del *codestream* JPEG2000, un paquete de 1 byte de longitud, es un paquete "vacío", teniendo ese byte un valor constante.

El hilo de lectura irá leyendo los paquetes especificados en la lista de la clase *PacketsList*. La lectura, gracias a la clase *HTTPReader*, no tendrá en cuenta si los paquetes se han incluido en la misma petición o en peticiones diferentes, o de si la respuesta ha sido *chunkeada* o no. Al leer cada paquete, se elimina el objeto *PacketData* asociado de la lista *PacketsList*, y se incluye la información en la *cache* (*JP2Cache*) mantenida por el sistema de gestión de imagen. Como vimos para la apertura de la imagen, para añadir la información del paquete emplearemos el método *Add_to_databin()*. Se debe tener en cuenta no leer los paquetes de un byte de tamaño, ya que no se solicitaron al servidor. No obstante es preciso incluirlos en la *cache*. Estos paquetes son bytes cuyo primer bit está a cero (el resto de los bits pueden tener cualquier valor). El hilo de lectura finalizará cuando acabe de leer todos los objetos *PacketData*, o cuando se produzca algún error. Es el hilo encargado de cerrar la comunicación con el servidor, tanto si ha acabado o no el hilo de escritura.

Cuando el sistema de gestión de imagen requiere detener el sistema de comunicación, o bien porque se ha abierto una imagen nueva o bien porque se ha cambiado de ROI, se llamará al método *stop()* de la clase *JP2Reader*. Este método elimina todos los objetos de la clase *PacketData* de la lista de la clase *PacketsList*, y cierra la conexión con el servidor, si aún está abierta, es decir, llamada al método *close()* del objeto de la clase *HTTPReader*. Esto provoca que el hilo que aún permanezca en ejecución finalice debido a una excepción. Se dice que el sistema de comunicación ha finalizado, cuando los dos hilos de ejecución están parados.

Es imprescindible para el correcto uso de la *cache*, que el hilo de lectura almacene los paquetes completos, que no se pueda detener su ejecución

quedándose en la *cache* almacenada una porción de un determinado paquete. Esto es garantizado al ser la llamada que realiza el hilo de lectura al método *readData()* de la clase *HTTPReader* una llamada bloqueante.

El orden que siguen tanto el hilo de escritura como el de lectura para recorrer la lista de la clase *PacketsList* es el mismo.

5.5.4. Descompresión

El sistema de descompresión está representado por la clase *JP2Render*. Su misión es descomprimir las regiones almacenadas en la lista de clase *RegionsList*. Para la descompresión emplea un objeto de la clase *Kdu_region_decompressor*. Esta clase permite descomprimir regiones rectangulares de una imagen.

Para comenzar la descompresión de una región, empleamos el siguiente método de la clase *Kdu_region_decompressor*:

```
boolean Start(Kdu_codestream codestream,
              Kdu_channel_mapping mapping, int single_component,
              int discard_levels, int max_layers, Kdu_dims region,
              Kdu_coords sampling, boolean precise)
```

Este método únicamente inicializa el descompresor. Los parámetros que nos interesan son: *codestream*, que referencia al *codestream* de la imagen; *discard_levels*, que indica el número de niveles de transformada *wavelet* a descartar (la resolución); y región que es la región propiamente dicha de la imagen a descomprimir.

Para descomprimir, empleamos el siguiente método:

```
boolean Process(int[] buffer, Kdu_coords buffer_origin,
                int row_gap, int suggested_increment,
                int max_region_pixels, Kdu_dims incomplete_region,
                Kdu_dims new_region)
```

A este método se le pasa, principalmente, el *buffer* donde almacenar la región descomprimida. A la hora de descomprimir la región pasada al método *Start()*, el método *Process()* tendrá en cuenta el tamaño del *buffer* pasado (*buffer.length*), de forma que descomprimirá una subregión de la región indicada haciendo que el tamaño total de los datos descomprimidos sea menor o igual en el tamaño del *buffer* pasado. Esto obligará a que, en función del *buffer* utilizado, necesitemos más o menos llamadas sucesivas al método *Process()*. Este método devuelve en el parámetro *incomplete_region*

la parte de la región indicada inicialmente al método *Start()* que aún queda por descomprimir. En el parámetro *new_region* devuelve que subregión ha podido descomprimir, en función del tamaño del *buffer* pasado. En el caso de que tras la llamada a *Process()* se haya descomprimido ya la totalidad de la región inicialmente pasada a *Start()*, el método devolverá *false*, y como es lógico, *incomplete_region* contendrá una región vacía. En caso contrario, devolverá *true*.

Finalmente, cuando tras sucesivas llamadas al método *Process()*, éste devuelve *false* para indicar que la descompresión de la región de interés ha finalizado, se deberá llamar al método *Finish()*, para liberar los recursos empleados por el descompresor.

El sistema de descompresión al iniciarse, crea un hilo de ejecución que recorre la lista *RegionsList* descomprimiendo, con los métodos anteriormente mencionados de la clase *Kdu_region_decompressor*, cada lista almacenada en cada objeto del tipo *Region* de la lista. Como hemos visto, la descompresión de cada región supone una o más llamadas al método *Process()*. Cada subregión descomprimida tras cada llamada a dicho método, le es comunicada al sistema de visualización para que éste actualice su representación en pantalla.

Cuando se termina de descomprimir una región, se comprueba si el sistema de comunicación ha finalizado. Si ha finalizado, la región es eliminada de la lista. Si el sistema de comunicación no ha finalizado aún, la región se elimina de la primera posición de la lista y se coloca en la última posición. Esto hace que el sistema de descompresión permanezca descomprimiendo las regiones solicitadas por el sistema de gestión de imágenes una y otra vez mientras que el sistema de comunicación no finalice.

5.5.5. Creación de la vista en miniatura

Cuando se pulsa el botón "Vista en miniatura", se crea una subventana (*JInternalFrame*) dentro del panel de visualización, representado por la clase *ImagePanel* (*JDesktopPane*). El sistema de visualización le notifica entonces al sistema de gestión de imagen que lo que requiere ahora es la visualización en miniatura de la imagen. El sistema de gestión de imagen asumirá esta petición como una petición normal y corriente en la que la resolución de la imagen es aquella cuya dimensión sea menor o igual a 200×200, y la ROI igual al tamaño completo de la imagen a dicha resolución. El sistema de visualización, cuando éste primero le va notificando las regiones descomprimidas, las va mostrando, en vez de en el panel principal, en la ventana dedicada a la vista en miniatura. En el momento en el que el usuario vuelva

Figura 5.8: Interfaz de usuario.

a interaccionar con la imagen (cambie la ROI o la resolución), se detendrá el proceso de creación de la vista en miniatura. Cuando se reinicie este proceso, volviendo a pulsar sobre el botón "Vista en miniatura", se continuará el proceso por donde se dejó.

5.6. Interfaz de usuario

La figura 5.8 muestra una captura de pantalla de la aplicación desarrollada, en la que se puede ver la interfaz de usuario, estando señalados los principales elementos.

La aplicación consta únicamente de una ventana redimensionable. La ventana se divide en cuatro zonas principales: la barra de botones, el cuadro de dirección, la zona de visualización y la barra de estado. Estas zonas se sitúan de arriba a abajo, ocupando cada una de ellas todo el ancho de la ventana.

La barra de botones posee cuatro botones, que son, de izquierda a derecha:

- *Abrir imagen local*: Aunque la aplicación ha sido desarrollada para trabajar principalmente con imágenes remotas, se ha incluido la opción de poder visualizar imágenes locales. En este caso, la única restricción existente para los archivos de imagen es la que impone la biblioteca Kakadu, es decir, se admitirán los archivos J2C y JP2.

- *Refrescar*: Este botón permite volver a cargar la imagen actualmente visualizada desde el inicio. Su comportamiento es similar al que se produce cuando se abre una imagen, pero en este caso es la misma imagen. Si situamos el cursor en el cuadro de dirección y pulsamos la tecla INTRO se produce el mismo efecto.

- *Zoom*: Estando pulsado este botón, el usuario puede realizar *zooms* en la imagen.

- *Vista en miniatura:* Al pulsar este botón, se muestra la ventana con la vista en miniatura de la imagen, si aún no ha sido mostrada.

El cuadro de dirección es donde especificamos la URL de la imagen a cargar. Una vez que escribimos la URL de la imagen deberemos pulsar la tecla INTRO.

La zona de visualización es la zona principal de la aplicación. Es donde se visualiza la ROI actual, y es donde el usuario puede interaccionar con la misma. Por defecto, el usuario interacciona con la zona de visualización para cambiar la posición de la ROI. Esto lo puede hacer mediante las barras de desplazamiento, existiendo una vertical y otra horizontal, o bien mediante el cursor del ratón: si pulsa con el ratón en la imagen y, sin soltar el ratón, desplaza el puntero, se desplazará la ROI según el movimiento del puntero. Si el usuario pulsa el botón de *Zoom*, entonces será capaz de cambiar la resolución que desea de la imagen, mediante el ratón: al pulsar el botón izquierdo se aumenta una resolución, y al pulsar el botón derecho disminuye una resolución. Recordemos que el usuario no se puede salir del rango de resoluciones que admita la imagen. Al cambiar de resolución, la ROI tendrá el mismo tamaño, pero se desplazará para que su centro se sitúe donde ha pulsado en usuario al hacer *zoom*.

Cuando se pulsa sobre el botón "Vista en miniatura", en la zona de visualización aparece una pequeña ventana, donde aparece la imagen con la que se está trabajando a baja resolución, en concreto, a la resolución en la que la imagen quepa en un cuadrado de 200×200. Esta ventana no es redimensionable. Sobre la vista en miniatura se representa un rectángulo rojo que identificará la posición y tamaño de la ROI actual sobre la imagen completa. El usuario puede mover dicho rectángulo, arrastrándolo con el ratón, para mover la ROI.

La barra de estado visualiza información de interés, como es el ancho y el alto de la imagen, la escala a la cual se está trabajando, y si la imagen no es local, la cantidad de bytes que se han recibido hasta el momento.

5.7. Evaluación

Para la evaluación del rendimiento de la aplicación, se ha realizado una comparativa de la misma con la aplicación *kdu_show* del paquete de Kakadu, empleando el protocolo JPIP.

Tanto la aplicación desarrollada como *kdu_show* visualizarán progresivamente la imagen *boatsip.jp2*, que tiene un tamaño de 2045×2045, no posee *tiles*, posee un tamaño de precinto de 128×128 y un tamaño de *code-block* de 64×64. Para el caso de nuestra aplicación, la imagen se ha pasado a formato J2C, y se le ha impuesto una progresión RLCP y el particionamiento del

Figura 5.9: Comparativa entre el sistema desarrollado y el protocolo JPIP, empleando el paquete de Kakadu.

tile en *tile-parts*, uno por cada resolución. Ambas aplicaciones visualizarán la imagen a una escala del 25 %. Se han hecho pruebas con otras imágenes, pero el resultado obtenido ha sido muy similar.

Se han realizados dos gráficas, una para cada aplicación, en las que se puede ver el PSNR, en el eje vertical, por *kilobytes* recibidos, en el eje horizontal. Estas dos gráficas aparecen en la figura 5.9.

La medida PSNR calculada para imágenes con 8 bits/componente, y expresada en decibelios, se calcula como:

$$\text{PSNR[dB]} = 10 \log \frac{255^2}{\frac{1}{N}\sum_{i=1}^{N}(s[i] - \widehat{s}[i])^2}$$

donde $s[i]$ y $\widehat{s}[i]$ denotan a un punto RGB de la imagen original y reconstruida, respectivamente. N es el número de puntos de la imagen.

Como se puede apreciar, *kdu_show*, empleando el protocolo JPIP, necesita leer menor cantidad de bytes para mostrar una reconstrucción aceptable de la imagen, que la aplicación desarrollada en el presente proyecto. Existe una diferencia media aproximada de $3dB$ en la distorsión de la imagen, con la misma cantidad de bytes leídos. El motivo es que al emplear el protocolo HTTP, debemos leer las cabeceras de cada mensaje, cuya longitud se ha tenido en cuenta al realizar las gráficas. Las cabeceras en el protocolo JPIP ocupan mucho menos espacio. No obstante hay que tener en cuenta que el servidor empleado para el caso del protocolo HTTP tiene limitada la longitud de cabecera, por lo que el número de *byte-ranges* que podemos incluir en una misma petición está limitado, en concreto, a aproximadamente unos 20 *byte-ranges*. Cuantos más *byte-ranges* permita un servidor HTTP en la misma petición, tendremos que leer menos cabeceras, al reducirse el número de peticiones necesarias. Además el servidor JPIP de Kakadu optimiza el orden de los paquetes a transmitir, como se verá en el apartado de posibles mejoras, minimizando la distorsión de la imagen en relación a los bytes transmitidos. De incluir ésta mejora en la presente aplicación, su curva se aproximaría a la del sistema JPIP.

5.8. Posibles mejoras

De las posibles mejoras que se le podrían aplicar a la aplicación desarrollada, las principales serían la de incluir soporte para archivos JP2 y la secuenciación óptima de paquetes.

5.8.1. Soporte para archivos JP2

Esta no supondría una mejora del rendimiento, pero sí le daría más flexibilidad a la aplicación. No se ha dado soporte a los archivos JP2 debido a su complejidad estructural, que haría ralentizar el proceso de lectura. Sin embargo, es posible que nos encontremos en la necesidad de tener que incluir información adicional a un archivo de imagen, como por ejemplo, una paleta de colores, información de interés relacionada con la imagen, etc. Esto no es posible hacerlo con los archivos J2C, los cuales tan sólo incluyen un *codestream* JPEG2000.

La aplicación utiliza la clase *Kdu_cache* de la biblioteca Kakadu para implementar el sistema de *cache*. Esta clase está preparada para trabajar con archivos JP2, de manera que existe un tipo de *data-bin* para albergar las cajas de un archivo JP2. Estos *data-bins* son llamados *meta data-bins*. En un mismo *meta data-bin* es posible albergar más de una caja, tal y como aparecen en el archivo.

La aplicación debería ir leyendo caja a caja de la imagen, almacenándola en la *cache*, hasta llegar a la caja del *codestream*, la cual se leería, una vez saltada la cabecera, de la misma forma que para un archivo J2C normal y corriente. Es preciso tener en cuenta que un archivo JP2 puede incluir más de una caja de *codestream*. En el caso de encontrarse con un archivo con esta peculiaridad, habría que decidir qué *codestream* visualizar, o dejar al usuario que lo decidiera. La clase *Kdu_cache*, por su parte, soporta perfectamente varios *codestreams*, debido a que a cada *data-bin* se le asigna una identificador de *codestream* (salvo en el caso de los *meta data-bins*, que no están asociados a ningún *codestream*).

En este caso, la inicialización del objeto de la clase *Kdu_codestream* no se realizaría pasándole directamente el objeto de la clase *Kdu_cache*. Se debería crear un objeto de la clase *Jp2_family_src*, que sería el que se inicializará con la *cache*, llamando al método *Open()*. Posteriormente se crearía otro objeto, pero de la clase *Jp2_source*, inicializándolo con el último objeto creado, también llamando a su método *Open()*. Sería este objeto el que le pasaríamos al método *Create()* de la clase *Kdu_codestream*.

Por lo demás, todo lo desarrollado para la presente aplicación se man-

tendría prácticamente igual.

5.8.2. Secuenciación óptima de paquetes

Esta sería una mejora consistiría en emplear una técnica que ya utiliza el paquete Kakadu para optimizar la transmisión de las imágenes a través del protocolo JPIP.

Cuando se requiere una ROI determinada a una resolución dada, la aplicación desarrollada para el presente proyecto solicitaría al servidor todos los paquetes de todos los precintos relevantes para dicha ROI de la imagen, desde la mínima resolución hasta la resolución dada, capa por capa.

Aunque sólo se solicitan los precintos necesarios para la reconstrucción de la ROI, a menudo de incluyen precintos que contribuyen a la reconstrucción de regiones de la imagen que no pertenecen a la ROI solicitada.

El paquete de Kakadu realiza dos cosas con el fin de optimizar la secuenciación de paquetes, a la hora de ser enviados por el servidor JPIP al cliente, y ordenarlos así por relevancia en relación a la reconstrucción de la ROI. La primera de ellas es que, para cada capa de calidad, los precintos son ordenados por su relevancia, en vez de por su resolución, como hace nuestra aplicación. La relevancia, ρ_i, del precinto i, es calculado con la siguiente fórmula:

$$\rho_i = \frac{\|A_i \cap W_{R_i}\|}{\|A_i\|}$$

donde A_i es la región espacial ocupada por el precinto i en su *tile*-componente-resolución, R_i, y W_{R_i} es la proyección de las muestras de las subbandas que pertenecen a la resolución R_i, y que contribuyen a la reconstrucción de la ROI en dicha resolución.

El servidor JPIP de Kakadu ordena los paquetes de los precintos a enviar al cliente, basándose en la fórmula anterior. Los paquetes siguen siendo enviados por capa, pero para cada capa, el orden de los paquetes ya no es por resolución.

La segunda fase de la optimización de la secuenciación de los paquetes sólo es posible si el archivo de imagen contiene almacenada la información de los umbrales de la pendiente de distorsión que fueron usados por el compresor para generar las capas del *codestream*. El compresor de Kakadu permite almacenar esta información en una marcador COM especial, incluido en la cabecera principal. Si el archivo de imagen posee dicha información, el servidor es capaz de ordenar los paquetes, ya no sólo por su relevancia en la misma capa, sino entre las distintas capas.

Figura 5.10: Impacto de la secuenciación óptima de paquetes.

Esta segunda fase de optimización se basa en asumir que el impacto de un paquete de un precinto i en la distorsión MSE (*Mean Squared Error*) de la ROI es igual a ρ_i veces el impacto que tiene este mismo paquete en el MSE de la imagen completa. Las contribuciones de cada *code-block* incluidas en la capa λ tienen unas pendientes de distorsión no más pequeñas que s_λ, donde s_λ sería uno de los valores almacenados por el compresor en el marcador COM especial mencionado anteriormente. Para modificar las pendientes de distorsión de los *code-blocks* para que reflejen la relevancia que tienen con respecto a la ROI sólo necesitaríamos multiplicarlos por ρ_i. Esto permite al servidor, a la hora de transmitir un precinto, empezar por los paquetes de las capas de mayor relevancia.

La figura 5.10 muestra una gráfica donde se puede comprobar el resultado de emplear la secuenciación óptima de paquetes, respecto a emplear la secuenciación tradicional.

Es evidente la mejora que supone esta técnica en el rendimiento del cliente, de forma que para el mismo número de bytes leídos, la distorsión de la imagen mostrada es menor.

El incluir esta técnica en el sistema propuesto no sería sencillo, ya que en Kakadu es el servidor el que realiza toda la ordenación de paquetes, debido a que tiene acceso a la imagen completa. En nuestro caso, el cliente no tiene acceso a la imagen completa, por lo que no puede calcular ρ_i. Esta información se debería incluir, al igual que hace Kakadu para la segunda parte de su optimización, en un marcador COM especial, el cual debería ser leído en el proceso de apertura de la imagen, antes de solicitar ningún paquete. Para ello sería necesario implementar una pequeña aplicación que incluyera en un archivo J2C dicho marcador.

Bibliografía

[1] M.D. Adams and F. Kossentini. JasPer: A software-based JPEG-2000 Codec implementation. In *IEEE Int.Conf. Image Processing*, volume II, pages 53–56, Vancouver, Canada, September 2000.

[2] S. Deshpande and Zeng W. Scalable Streaming of JPEG2000 Images using Hypertext Transfer Protocol. ACM Multimedia, October 2001.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC2068, January 1997.

[4] S. Holzner. *La biblia de Java 2.* ANAYA Multimedia, 1999.

[5] J. Li and H.-H. Sun. A Virtual Media Vmedia Access Protocol and Its Application in Interactive Image Browsing. Microsoft Research China.

[6] ISO/IEC JTC1/SC29/WG1 N1855. JPEG 2000 Part I: Final Draft International Standard. Technical report, ISO/IEC, August 2000.

[7] ISO/IEC JTC 1/SC 29/WG 1 N2904. JPEG 2000 Part 9: Interactivity tools, APIs and protocols – CD 1.0. Technical report, ISO/IEC, March 2003.

[8] A. Skodras, C. Christopoulos, and T. Ebrahimi. The JPEG 2000 Still Image Compression Standard. *IEEE Signal Processing Magazine*, pages 36–58, September 2001.

[9] G. Strang and T.Ñguyen. *Wavelets and Filter Banks.* Wellesley-Cambridge Press, 1996.

[10] D.S. Taubman. Remote Browsing of JPEG2000 Images. The University of New South Wales, Sydney, Australia.

[11] D.S. Taubman. High Perfomance Scalable Image Compression with EBCOT. *IEEE Transactions on image processing*, 9(7):1158–1170, July 2000.

[12] D.S. Taubman. Kakadu Survey Documentation. Technical report, The University of New South Wales, June 2002.

[13] D.S. Taubman. Proposal and Implementation of JPIP (JPEG2000 Internet Protocol) in Kakadu v3.3. Technical report, The University of New South Wales, August 2002.

[14] D.S. Taubman and M. Marcellin. *JPEG2000, Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, 2002.

[15] D.S. Taubman and R. Prandolim. Architecture, Philosophy and Perfomance of JPIP: Internet Protocol Standard for JPEG2000.