Improving Web proxy caching on browsing JPEG 2000 remote images with JPIP

J.P. ORTIZ, V.G. RUIZ, I. GARCIA
Computer Architecture and Electronics Department
University of Almeria
04120 Almeria, Spain
jportiz@dali.ace.ual.es, vruiz@ace.ual.es, inma@ace.ual.es

ABSTRACT
This paper presents a set of minimal modifications of the JPIP standard architecture to improve the Web proxy caching in those applications designed for progressive and interactive remote browsing of JPEG 2000 images, using the JPIP architecture with HTTP as base protocol, and employing Internet connections through proxies. Results show that the images reconstruction velocity is highly incremented in a progressive way.

KEY WORDS
JPEG 2000, JPIP, HTTP, proxy, cache, data-bin.

1 Introduction

JPEG 2000 is a recent image compression standard [1][2][3], developed by the Joint Photographic Expert Group (JPEG), and it is based on the Discrete Wavelet Transform (DWT) and the Embedded Block Coding with Optimized Truncation (EBCOT) [4]. One of its main features is its high scalability, allowing quality, resolution and spatial scalability.

The basic format of a JPEG 2000 compressed image is the code-stream, compound of a set of markers that reference useful information (compression parameters, image features, beginning or end of a packet, etc.), and a set of variable length packets, each of them containing the compression result of a specific area of the original image.

The order in which code-stream packets are stored determines the kind of progression when decompressing the image. This order is indicated by referencing the 4 levels of scalability that the standard affords, so, for example, the RLCP order indicates that packets are stored by resolution (R), by quality layer (L), by component (C) and by precinct (P). This progression determines the default progression existing in the code-stream, but it is possible to access to the desired packets independently, and extract them in any order.

In order to offer a higher flexibility and functionality to image files, JPEG 2000 defines a highly configurable and extending file format in which, besides being able to encapsulate one or more code-streams, permits to include diverse additional information.

The high scalability offered by the JPEG 2000 standard makes it ideal for progressive and interactive remote images browsing applications, in which a client, due to the user's interaction, carries out requests to a server asking for a specific region, within a specific resolution level, and having the ability to indicate others parameters like the number of quality layers or the number of desired components.

With the aim to broach the implementation of this kind of systems, Part 9 of the JPEG 2000 standard proposes a server/client architecture and a communication protocol called JPIP [5][6]. In this architecture, that can be observed in Figure 1, the client indicates to the server a region and the highest resolution level associated to a window of interest (WOI) within a remote image. The JPIP server would answer the client the necessary information to reconstruct in the client the requested WOI, that the server would collect from the local image.

JPIP protocol can be implemented over different protocols, being HTTP/1.1 protocol [7] one of the most interesting, since it easily allows to encapsulate requests and responses in HTTP messages, being able to exploiting the Web infrastructure.

The Web infrastructure offers many advantages in relation to another more specific, as for example, the caching system, which can be used not only at a client level, but also at the level of the different proxies that exist through the communication between clients and servers.

The JPIP protocol, running on HTTP, does not exploit all the performance of the Web caching system. The here proposed system tries to exploit to the maximum proxies caches in applications for progressive and interactive remote browsing of JPEG 2000 images.

The rest of this paper is structured as follows: in the Section 2, the current working JPIP architecture is detailed, showing its deficiency for the task of exploiting the Web caching system, and proposing a solution for it; in the Section 3, a possible implementation of the suggested system is explained; in the Section 4, the achieved results with the proposed system are exposed, and finally, in the Section 5, we conclude with the pertinent conclusions about our proposal.

Table 1. Data-bin types

Type	Information
Precinct data-bin	Precinct data
Tile-header data-bin	All tile-part headers concatenated of a tile
Tile data-bin	All tile-parts concatenated of a tile
Main-header data-bin	Main header
Metadata-bin	Collection of boxes of a JPEG 2000 family file

2 The proposed system

The JPIP protocol divides any JPEG 2000 image I (either in raw format or in a more complex format) into a set of N parts called data-bins, $I = \{d_1, d_2, \ldots, d_N\}$, so that, when a client performs a request in a instant t, about a specific window of interest of the image I, $WOI_t^{(I)}$, the information that the server sends in response, $r(WOI_t^{(I)})$, is the subset of image data-bins that permit to reconstruct that petition, $F(WOI_t^{(I)})$. Therefore,

$$r(WOI_t^{(I)}) = F(WOI_t^{(I)}) = \{d_l, \ldots, d_k\} \subset I \quad (1)$$

As it can be observed in Figure 1, the server can optionally maintain a client cache model, in order not to send information (data-bins) that it has already sent. It is only incumbent upon the requests of a same client.

The different types of data-bins existing can be seen in Table 1. It has to be pointed up that the most important data-bins are the concerning to the code-stream packets (Precinct data-bin).

The Web infrastructure caching system that we mean to use is maintained by the existing proxies in the connection between client ant server. A proxy is a special server that acts as intermediary between one or more clients and one or more servers (as much the clients as the servers can be themselves proxies), in order to offer one or more, among others, these functions:

- Protocol conversion, when the client and the server use different protocols.
- Security control, allowing to establish a set of security rules in connections, in one or other direction.
- To provide with a caching system, profiting from redundant requests of the clients, to reduce the responses latency. Many times the requests of one or more clients are, either equals or they share information, and they can be solved by using proxy cache, without turning to the server.

With regard to the JPIP protocol, it is true that a proxy could store in its cache the requests results, but in order to this one could use the cache information with other different requests, these one had to be identical to the first one. This is rather inefficient, since many of the different requests for a same image will share information. Currently, for two different requests from the same image, from two different clients, that require a common set of data-bins, the server response for the last request is complete, although existed a proxy/cache hierarchy in the communication between client and server. More concisely,

$$\begin{array}{l} P(WOI_{t+\Delta t}^{(I)}) \neq \phi \Rightarrow \\ r(WOI_{t+\Delta t}^{(I)}) = F(WOI_{t+\Delta t}^{(I)}) \end{array} \quad (2)$$

where $P(WOI_{t+\Delta t}^{(I)}) = F(WOI_t^{(I)}) \cap F(WOI_{t+\Delta t}^{(I)})$ is the set of common data-bins for the different requests, that could be saved in the proxies.

The most efficient way would be that the cache of the existing proxies in the communication should avoid a server complete response, profiting from the common data-bins stored due to previous requests, i.e.,

$$\begin{array}{l} P(WOI_{t+\Delta t}^{(I)}) \neq \phi \Rightarrow \\ r(WOI_{t+\Delta t}^{(I)}) = F(WOI_{t+\Delta t}^{(I)}) - P(WOI_{t+\Delta t}^{(I)}) \end{array} \quad (3)$$

Although actually it is not in this way, we will assume that two requests made to the same server relating to the same image in different moments of time, if they are equals, then the requests results are equal too, and therefore,

$$WOI_i^{(I)} = WOI_j^{(I)} \Rightarrow F(WOI_i^{(I)}) = F(WOI_j^{(I)}) \quad (4)$$

It is true that, due to the features of the standard, it is possible to implement an application for remote browsing of images, without necessarily using the JPIP protocol. For example, it would be possible only to use the HTTP/1.1 protocol and, using byte-ranging, a client would access to those necessary data-bins for a certain WOI. The problem is that the most of proxies do not cache byte-ranges efficiently, because of which the most of them, either are ignored, that is, they pass through proxies without caching, or, when receiving a byte-ranging request, they ask the server for the whole associated file and, once received, they sent the client the requested byte-ranges. We can not assure that the possible proxies existing between us and the final server make an efficient caching of the byte-ranges.

The first solution could be to divide a WOI request into a request of data-bins references, and a set of requests, one for each data-bin reference. Thus, the client would send to the server the WOI that it needs to browse, and the server would return the necessary data-bin references. The client would ask the server for one by one every data-bin, in different requests. This would make that, when encapsulating every request in HTTP messages, proxies could cache them in an independent way, making (3) possible. It would only have to add to the JPIP server the possibility

Figure 1. Client-Server JPIP architecture.

of answering the requests of (i) the references of the necessary data-bins for a WOI and (ii) a specific data-bin. Besides, the JPIP server had to implement the HTTP protocol caching system, handling the different associated headers (`Cache-Control`, `If-Modified-Since`, etc.).

This first solution, a priori, would efficiently use the caching system of the Web infrastructure, besides it would not need the own caching system of the protocol JPIP. However, this solution has a overload problem because of the own ASCII headers of the HTTP protocol. Depending on the type of the messages, an HTTP message can have 200 additional bytes on average, due to the ASCII headers. To reconstruct an area of a JPEG 2000 image of 2954×1976, with 7 resolution levels and 10 quality layers, with a precinct size of 128×128, we need a total of 1260 packets, for what we would have 252000 overload bytes on average when effecting the corresponding data-bins requests.

Besides, this overload per request is not homogeneous, so that for very large data-bins, the overload is small, but for very small data-bins (for example, the data-bin associated to an empty packet, of a 1 byte), the overload can be higher than 20000%.

We find the solution by grouping data-bins in blocks. For each image I, we define a minimum block size $s^{(I)}$, and we group the image data-bins in blocks, so that each block contains the minimum number of data-bins that the total size of all contained data-bins is equal or greater than $s^{(I)}$. Realize that we only can assure this for all blocks except the last one.

For a minimum block size equals to 0, we would find that a block would be identical to a data-bin. The minimum block size would be chosen depending on the average size of the image data-bins.

When making a request for block, instead of for data-bin, we control the overload produced by request due to the HTTP headers. For any response to a request of any block, we would have a maximum overload of $((100h)/(s^{(I)}))\%$, being h the minimum average size of the headers.

This overload assumes that, when requesting a specific block, all blocks included are necessary, although many times it is not true. In any case, it would comply with (3) at block level, with a time delay reduction in responses.

The most important data-bin is the one associated to the packets, so if we would want the blocks to include the maximum number of interesting data-bins, it would be recommendable that the last progression dimension was by precinct, P (for example, RLCP), that is, the packets would be organized by lines, within the code-stream.

3 Implementation

The implementation of this proposed solution is not intended for replacing the JPIP's current, but for being a complement to use in those applications in which it is required to use efficiently the caching system of the Web infrastructure. Next we propose the basic modifications to do to a classic JPIP server in order to implement the proposed solution.

Firstly, we have to modify the server for accepting a new type of request, with which a client can ask for the references of the necessary blocks to reconstruct a specific WOI. For this, we will include the parameter `request`, with the value `blocks`, in a request. So, for example, a request of a client asking for the references of the necessary blocks to reconstruct a WOI with a size of 310×310, with an origin $(10, 10)$, and at a resolution level within an area of 512×512, for the image `rimage.jp2`, would be as follows:

```
GET http://jp2.server/rimage.jp2?roff=
10,10&rsiz=310,310&fsiz=512,512&request=
blocks HTTP/1.1↩
Host: jp2.client↩
↩
```

The server, when the `request` parameter was found, with the `blocks` value, it would answer the references of the necessary blocks to reconstruct the required WOI. The client can also indicate the wanted minimum block size, in bytes, with the `sblock` parameter. This size could be either accepted or not by the JPIP server, being the server able to modify it if it is necessary, notifying the client this modification by means of `JPIP-sblock` header. If the client does not specify any block size, the server will use a default size and will notify the client of it.

Next we can see an example, with the same last request, but specifying 512 bytes as the minimum block size required:

```
GET http://jp2.server/rimage.jp2?roff=
10,10&rsiz=310,310&fsiz=512,512&request=
blocks&sblock=512 HTTP/1.1↩
Host: jp2.client↩
↩
```

The server could answer a response as follows, in which the server changes the minimum block size to 256 bytes:

```
HTTP/1.1 200 OK, with modifications↩
JPIP-sblock: 256↩
↩
...
```

If the server did not find the `request` parameter, its behavior would be the standard. If it is found, it must not include headers for disabling proxies caching, like `Cache-Control: no-cache`, but it must be able to interact with the Web caching system, supporting headers like `If-modified-since`, and ignoring the headers of the caching system of the JPIP protocol.

The content of a server response is a set of references of those necessary blocks to reconstruct the requested WOI. Every block reference is an index, starting at 0, with a VBAS structure (variable-length byte-aligned segment), defined in JPIP. This structure, which can be observed in Figure 2, allows to store a number B, which binary representation would have a length of L_B bits, in a total of $\lceil L_B/7 \rceil$ bytes.

Figure 2. VBAS structure.

This first server response will not be cached, due to the fact two identical requests can produce different server responses (server could modify any request parameter). Because of this, it would be more interesting that the server response was as smaller as possible, performing the compression. It is possible in the HTTP protocol by means of the `Content-enconding` header. With this header, the server could compress the response content, indicating which algorithm, within the set of supported algorithms of the protocol, has used (for example, it could indicate `deflate`, `gzip`, etc.).

In Figure 3 we can see the JPIP client structure, with the proposed modification. It needs two sockets for running, S_1 and S_2. It would send references requests through S_1 and, when it is received any reference, it would send the associated block request through the socket S_2. In the proposed client structure, the own pipe-lining of the HTTP/1.1 is fully exploited, allowing to make requests and to receive responses both in parallel.

Figure 3. JPIP client proposed.

As soon as the client receives a block reference through the socket S_1, it sends a request about the referenced block, through the socket S_2. In order to request a specific block, it is necessary to use the `block` parameter, taking value of the wanted block index. In the following example we can observe a client request about the block number 20, of a remote JPEG 2000 image called `rimage.jp2`:

```
GET http://jp2.server/rimage.jp2?block=
20&sblock=256 HTTP/1.1↩
Host: jp2.client↩
↩
```

It is necessary that the client indicates in every one of its requests the minimum block size used, which will be employed by the server to group the image data-bins in blocks, and indexing them. This size must be the same that the signaled by the server as result of the first client request, about the references of the blocks.

Every read block contains a set of contiguous image data-bins. In order to offer the highest compatibility, the format of these data-bins is the same that the format employed in the JPIP protocol. The order in which the server groups data-bins in blocks, depending on the minimum block size, is completely free, although it would be more efficient to use an order that allows to improve the progressive visualization in the client. In the same way, the order of the references sent by the server, due to a WOI request, is free too, although, for example, for a simple codestream, the best one would be that the block which contains the main header data-bin was firstly sent (and it would be

recommendable that its position within the block was the first one); it would be also recommended to use an order in which the packet data-bins follow a quality progression, like, for example, LRCP.

The server must comply with that, having the same minimum block size and the same block number, two different requests would produce the same set of data-bins. This must be complied strictly to avoid incoherence in the cache of the proxies.

Figure 4. Architecture used in the experimental tests.

4 Results

For the realization of the tests, it has been employed the architecture that we can see in Figure 4. In that architecture we can observe that the connection between the clients and the server is through one (or more) Web proxies, with a caching system. Several WOIs, all about the same image, are visualized in different moments of time, in different clients. The JPEG 2000 remote image here used is 2954×1976, with 7 resolution levels and 10 quality layers. In order to simplify, the image format will be the raw format (.j2c), and it will not contain neither tile nor tile-parts.

The image progression will be LRCP. The data-bins order will be as follows: the first data-bin will be the main header data-bin, and the next data-bins will be the packets data-bins, in the same order as the image progression.

As minimum block size will take 512 bytes, and, having an average header size of 100 bytes per HTTP message, we will have a maximum overload of 20%.

For recording the showed results, firstly it is visualized in one of the clients a specific WOI, with a size of 512×512, $(100, 100)$ as origin, at the maximum resolution level. After this visualization, another different client will visualize a WOI with a size of 512×512, with $(356, 356)$ as origin, and at the same resolution level. These two visualizations will be made for each system to compare, two for the classic JPIP system, and two for the proposed system.

In Figure 5 we can see the relation between the PSNR (in dB) of the WOI and the time (in seconds), in every evaluated system, for the second visualization. We have supposed here that the server connection bandwidth is 4KB/s on average. On the other hand, for the connection bandwidth of the first proxy in the client/server connection, is only limited by the architecture of the local net, where the first proxy is situated. For the tests, we have supposed that this bandwidth is 4MB/s.

We can see that the velocity for the reconstruction of the WOI, for the second visualization in the proposed system, is quite higher than the classic JPIP system. This velocity would be incremented as much as more requests, about the same image, are made.

Figure 5. Rate-distortion results from experimental tests.

The time delay, due to the reception of the blocks references, has not been analyzed, for it is minimum, because of, mainly, these two reasons: (i) the requests of blocks are made in parallel with the reception of the block references, and (ii) it is possible to compress, as we said in previous sections, the first server response, exploiting the feature of all the algorithms proposed in the HTTP/1.1 protocol that allows to decompress the data “on-the-fly”.

5 Conclusions

For architectures similar to the architecture employed for the realization of the tests, the classic JPIP system is rather inefficient, because it does not exploit the redundancy existing in the requests of different clients for WOIs of a set of remote images. This redundancy can be absorbed quite efficiently by the Web proxies using the proposed system. This proposed system coexists with the classic JPIP system, and it requires minimal modifications.

Acknowledgments: This work has been partially supported by the Spanish CICYT through grants TIC99-0361 and TIC2002-00228.

References

[1] ISO/IEC 15444. *Information Technology – JPEG2000 Image Coding System – Part 1: Core coding system*, 2000.

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

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

[4] D.S. Taubman. High Performance Scalable Image Compression with EBCOT. *IEEE Transactions on image processing*, pages 1158–1170, July 2000.

[5] ISO/IEC 15444. *Information Technology – JPEG2000 Image Coding System – Part 9: Interactive tools, APIs and protocols*, 2003.

[6] D.S. Taubman and M.W. Marcellin. JPEG2000: Standard for Interactive Imaging. *Proceedings of the IEEE*, 90(8):1336–1356, August 2002.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. http://www.ietf.org/rfc/rfc2616.txt, June 1999.