Efficient and scalable open-source JPIP server for streaming of large volumes of image data

J.P. García-Ortiz, C. Martin, V.G. Ruiz, J.J. Sánchez-Hernández, I. García
Comp. Architecture and Electronics Dept.
University of Almería, Spain

D. Müller
European Space Agency, ESTEC
Noordwijk, Netherlands

***Abstract*—This paper[1] presents a new efficient and highly scalable open-source development of a JPIP server, financed by the European Space Agency, that allows the streaming of large volumes of JPEG 2000 imagery. Although its was mainly designed with the aim to serve the Sun data generated by the recently launched NASA SDO observatory, its features can be very interesting for many other remote image/video browsing contexts. The results show that it is currently one of the best implementations of JPIP server, both in terms of demanded computing resources and quality of the reconstructions in the clients.**

I. Introduction

Some of the powerful features offered by the novel JPEG 2000 multi-part standard [1] are lossless/lossy compression, random access to the compressed streams, incremental decoding and high degree of spatial and quality scalability. These characteristics have led it to obtain the recognition as a state-of-the-art solution among applications for remote browsing of high-resolution images.

JPEG 2000, in combination with the JPIP protocol [2] defined in its Part 9 [3], has already been successfully used in many scientific areas (e.g. tele-microscopy [4] or tele-medicine [5]), and it has a significant potential in any other one where large volumes of image data need to be streamed, like for example Google Earth/Maps.

A noticeable example in astronomy is the JHelioviewer project [6], developed by the European Space Agency (ESA) in collaboration with the National Aeronautics and Space Administration (NASA). Its main goal is to deploy an interactive, data browsing and analyzing platform to accommodate the staggering data volume of 1.4 TB of images per day that will be returned by the Solar Dynamics Observatory [7]. Among other data products, SDO will provide full-disk images of the Sun taken every 10 seconds in eight different ultraviolet spectral bands with a resolution of 4096×4096 pixels.

The client side of this project is being developed in Java, which code is stored under a open-source license in Launchpad. It uses the Kakadu [8] JPEG 2000 library, developed by D. Taubman. This library is currently one of the best and most used JPEG 2000 implementations because it is highly optimized. Moreover, although it is commercial software, its binaries can be redistributed without restrictions for open-source solutions.

The Kakadu package also contains some demo applications, including a completely functional JPIP server. Even though this server has been improved significantly throughout all the library versions, it still suffers from scalability and stability restrictions. In the case of the JHelioviewer project, where a high load of data transmission and client connections is expected, the solution provided by the Kakadu library does not offer enough performance for very large imagery systems, as it is shown later in this paper.

Although there are other freely available implementations of the JPIP, none of them is capable of complying with the necessary requirements imposed by the JHelioviewer project. One of them is the open-source OpenJPEG JPEG 2000 library which was developed under the 2KAN project [9]. In this library there is an implementation of a JPIP server, called OpenJPIP [10], but unfortunately, by the time being, it only supports tile-based streaming, which is only recommendable for certain specific applications, not the case of the JHelioviewer project. Most importantly, OpenJPIP does not implement a fully server architecture, like the Kakadu server. It is designed as a CGI module for an existing Web server. This implies that the development does not tackle the server performance or scalability at all, depending these issues of the used base system. These restrictions led it to be discarded as well for the JHelioviewer project. Other implementation that was taken into account was the CADI software [11] developed by the Group on Interactive Coding of Images (GICI) at the Universitat Autonoma de Barcelona. In this case, this solution was discarded because it is written in Java, a programming language that is not as efficient as C++ to control the CPU and the memory usages of the server host.

In this context where there is an absence of JPIP server implementations capable to comply with the project requirements, ESA decided to finance the development of a new efficient and scalable open-source JPIP server. This paper presents this application, which first version is currently available on Launchpad [12].

The rest of the paper is organized as follows: in Section II the problem in question is analyzed in detail. Section III is dedicated to explaining the proposed solution, which is later evaluated in Section IV. The paper ends with some conclusions and future work (Section V).

[1]This work has been funded by grants from the Spanish Ministry of Science and Innovation (TIN2008-01117) and Junta de Andalucía (P08-TIC-3518), in part financed by the European Regional Development Fund (ERDF).

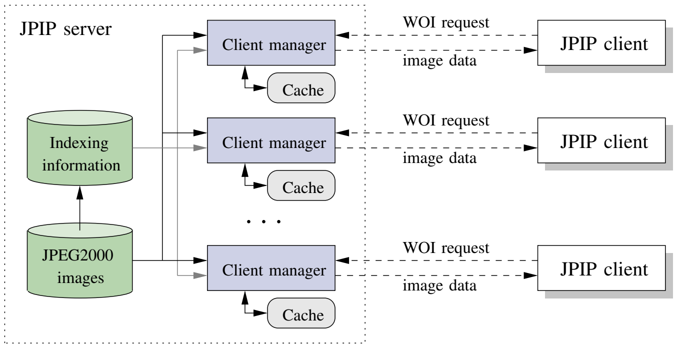

Fig. 1. Common functional architecture of a JPIP communication system, focused on the server side.

II. Server architectures

Figure 1 shows the common functional architecture of a JPIP communication system, focused on the server side. For each client connection, a new communication session is established and handled by a client manager. Although the JPIP protocol allows stateless connections, they are not commonly used.

Within the session a client can open different channels, one for each remote image to explore. Clients explore the desired images by means of WOI (Window Of Interest) requests. A WOI is usually identified by a rectangular region and a zoom or resolution level. The client manager extracts from the associated JPEG 2000 image those parts related to the WOI and send them to the client.

The client usually imposes a length limit for the server responses with the aim to control the communication flow. The complete response of a WOI is thus completed in different message exchanges, repeating the same request several times. This is possible thanks to the cache model maintained by each client manager, which records which image parts have been already sent.

For this data extraction the server needs an indexing information that is generated parsing the image files. The performance of the server is directly affected by how this information is generated. For example, it is possible, for each WOI request, to parse always the entire image file looking for the required parts. This does not consume memory, but it involves a considerable processing and disk load. On a completely contrary approach, it is possible to pre-build a complete index file and load it completely before attending a WOI request of an image. This reduces at the maximum the processing and disk load, but consumes too much memory.

The hybrid approach achieves a good relation between the memory consumption and the CPU/disk usage. It consists of pre-building some little indexing files, which contain the references of the main parts of the image, and then parsing on demand the images depending on the client requests. The index of each image is thus built on memory, on demand.

The implementation of a JPIP server must consider that the indexing information is shared by all the client managers. Depending on how it is generated, the sharing mechanism becomes more or less complex. In the case of complete pre-build index files, the access of the client managers is only for reading. However, for the hybrid approach, the client managers access to the index information for reading as well as for writing.

At the moment of writing this paper there is not any published work related to either JPIP server implementations or architectures. This is why it has not been possible to include any reference or comparison to previous related works. Nevertheless, the architecture of a JPIP server is quite similar to the one of a common Web server. For example, as it was mentioned in the introduction, the OpenJPIP server has been implemented as a layer for a Web server. Next some existing works related to Web server architectures are analyzed.

There are multiple possible approaches for implementing a web server, as it is commented in [13]. The multi-processes (MP) and multi-threads (MT) are the most common ones.

With MP each client is handled by a different process. The main advantage of this approach is the stability: if one process crashes, the other ones are no affected at all. On the contrary, this solution achieves a lower performance than MT in terms of memory consumption, operating system load (creating and killing processes) and inter-processes communication. The mechanisms for sharing information between processes in the existing platforms are usually less efficient that in the case of threads.

The most used Web server nowadays, Apache [14], adopts the MP approach in the Unix version 1.3 and in the version 2.0's multiprocessing (MPM) prefork module.

The MT approach allows an easy and natural way of programming a server, becoming simple and efficient the sharing and communication between threads. However it suffers from stability since if a thread crashes, the entire server process gets down, stopping also all the other threads. The Kakadu server [8], for example, adopts this approach.

With the aim to achieve a balance between the two previous solutions, some implementations use an hybrid approach (MP+MT), dividing the server in several processes, and each process in several threads. This increases the stability and obtains a performance near the MT solution. The Apache 2.0 Worker MPM implements it. The main drawback of this approach is inherited from the MP one, that is, the sharing and communication between the processes, and hence between the threads among processes.

Apart from the MP, MT or MP+MT, there are many other different proposals studied by the research community, some of them compared in terms of performance in the work of D. Pariag et al. [15]. A particular and very referenced proposal is the Flash server of V.S. Pai et al. [16]. It implements an AMPED (Asynchronous Multi-Process Event Driven) architecture that avoids the use of blocking I/O operations, and hence reducing the associated idle times. Although it showed promising results, the work of Gyu Sang Choi et al. [17] demostrated that it suffers from scalable performance, in relation to a MT solution, on multi-processor machines.

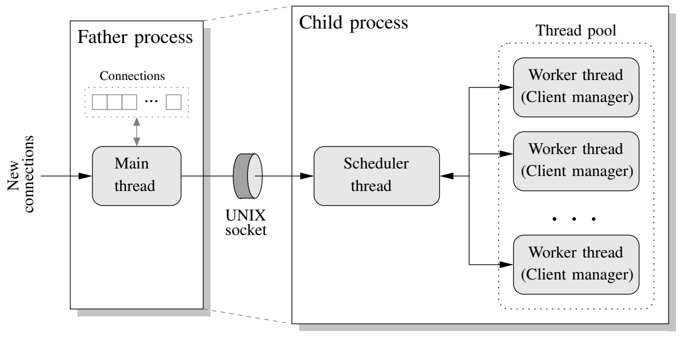

Fig. 2. Schematic representation of the ESA JPIP server architecture.

III. Proposal description

The open source project here presented was carried out with the aim to implement, using the C++ programming language, an efficient and very stable/scalable solution of a JPIP server. It has been specifically designed for Unix systems in order to fully profit from its characteristics, discarding a portable design which might compromise the efficiency.

The ESA JPIP server is capable to handle the following JPEG 2000 image files: raw J2C, JP2 and JPX with or without hyperlinks. The main requirement for the image files is that they must contain PLT markers, defined in the standard, with the information about the length of all the packets. These markers allow the server to build the indexing information of the different parts of the image without decoding. It simplifies the code and avoids to use any JPEG 2000 engine.

Almost any packet progression is allowed for compressing the images, but the RPCL progression is strongly recommended to be used for achieving an efficient performance, because of the organization of the packets in the file. Others implementations, like Kakadu, recommend this progression as well.

Fig. 2 shows a schematic representation of the server architecture. It consists of a hybrid model combining both process and thread approaches. Nevertheless, it is not a classical hybrid MP+MT model, as it is implemented in Apache for example, but it is more a pure MT approach with the minimum MP support for achieving a good robustness. There are only two processes, hereinafter called father and child. The second one is who maintains all the working threads.

The father process creates the child process by forking and watches it. It also has the listening socket of the server to accept new incoming connections. When a new client connection is established by the father process, it sends this connection to the child process through a UNIX-domain socket and records it in a vector where all the opened connections are recorded as well. If the father detects that the child has finished (e.g. due to a crash) it creates a new child process forking itself. Taking into account that it inherits the vector of the current opened connections, it can continue handling them without interruptions for the clients.

The child process provides all the functionality to handle the client connections. It contains a scheduler thread for reading the new connections sent by the father through the UNIX-domain socket. The scheduler thread assigns each connection to a working thread available in the maintained thread pool.

Each working thread implements the necessary functions, explained in Section II, associated to the client manager module shown in Fig. 1. The indexing information can be easily shared in memory by all the threads, without the efficiency restrictions when dealing with processes.

In order to generate the indexing information of the images a hybrid approach has been adopted. When an image is going to be served for its first time, a little associated cache file is created with the index of the main parts of the image, mainly the position of the header and the PLT markers. This cache file is loaded the next times the same image is served again.

With the help of this cache file, the indexing information is generated by the server in memory on demand depending on which regions are explored by the clients. Actually, the more resolution levels the user explores of an image, the larger becomes the related index data. The space required for this data has been reduced considerably, requiring the minimum possible number of bytes for each index item.

The index of each opened image for being served is stored as a node in a double-linked list shared by all the threads. Each node may also have references to other nodes of the list. For instance, in the case of a JPX file with hyperlinks, it is represented by a node which points to a set of other nodes, each one associated to each hyperlink of the file.

The access control of the threads to this shared information has been implemented using two different mechanisms. A general mutex locking mechanism has been adopted for reading/modifying the list. These operations are fast and only performed when opening/closing images. For each node a reader/writer locking mechanism has been used, in order to control the access to the indexing information of each image. This mechanism gives a priority to the readers higher than to the writers. This corresponds to the server behaviour because the read operation is the most common, while the write operation is performed just when incrementing the indexing information for a new resolution level. All these locking mechanisms are available by means of the POSIX pthread library.

This architecture provides a fault-tolerant and robust approach for the server, as well as it offers a good performance. The multi-threading solution implemented in the child process is efficient in terms of memory consumption and fast sharing/locking mechanisms. Having separated the client handling code from the father process provides robustness and security. If the child process crashes, the father process will be able to launch a new child process, keeping all the opened client connections (clients do not notice anything).

The image data is transmitted efficiently. The precincts that geometrically overlap by the requested WOI are sent always first, following the LRCP progression. This is the optimal one in terms of rate/distortion, as it was analyzed in [18]. As it is later shown in the evaluation, this achieves a significant gain in rate/distortion.

IV. Evaluation

The development has been evaluated being compared to the JPIP server provided by the commercial Kakadu package [8], currently the most referenced JPEG 2000 solution due to its good performance.

The aim of the first test carried out was to compare the both solutions in terms of memory consumption and CPU usage. 20 linked JPX files were created with 1000 different frames each one corresponding to 4096×4096 SDO Sun images. Every 5 seconds a flash crowd of 100 connections was established. Each connection is related to a JPX image from the available set (20), and it simulates a client that plays the video during 30 seconds, requesting 15 sequential images in each query, exploring all the 1000 images. After 30 seconds, all the connections are closed, releasing the related channels as well. The used WOI was 1024×1024 in the same resolution level. This scenario was running during a week.

Although there are not many simultaneous clients in this experimental scenario, the server load is quite high, due to the large amount of image data that needs to be handled, distributed in many different files. Moreover, this scenario is a common situation within the context of the JHelioviewer application.

When generating the JPEG 2000 images used in this experiment the PLT markers have been included and the following compression parameters have been used: 8 quality layers, 8 resolution levels and RPCL as progression order. Precincts have been used with a size of 128×128.

Table I shows the average and standard deviation of the results of this test. As it can be seen, the Kakadu server needs around 2 GB of RAM, while the ESA server only needs 30 MB. Moreover, the std. deviation says that the memory consumption Kakadu server is quite more variable than the ESA server, which on the contrary maintains the same memory consumption almost all the time.

The CPU usage is similar in the two solutions, although the Kakadu server seems to need less. Nevertheless, the logs have shown many records with 0 usage, which, considering that the logs have been recorded every 5 seconds, only means that the Kakadu server generates delays attending the connections. This would be coherent with the differences in the standard deviation.

The average throughput, in terms of responses by second, has been also recorded in this scenario, when all the 100 clients are communicating. The Kakadu server achieves a value around of 232, while the ESA server raises up to 1068.

	Memory (MB)		CPU (%)	
	Ave.	Dev.	Ave.	Dev.
ESA server	30.17	1.77	213.25	76.05
Kakadu server	1871.46	345.56	176.54	128.04

TABLE I
Results of the benchmarking.

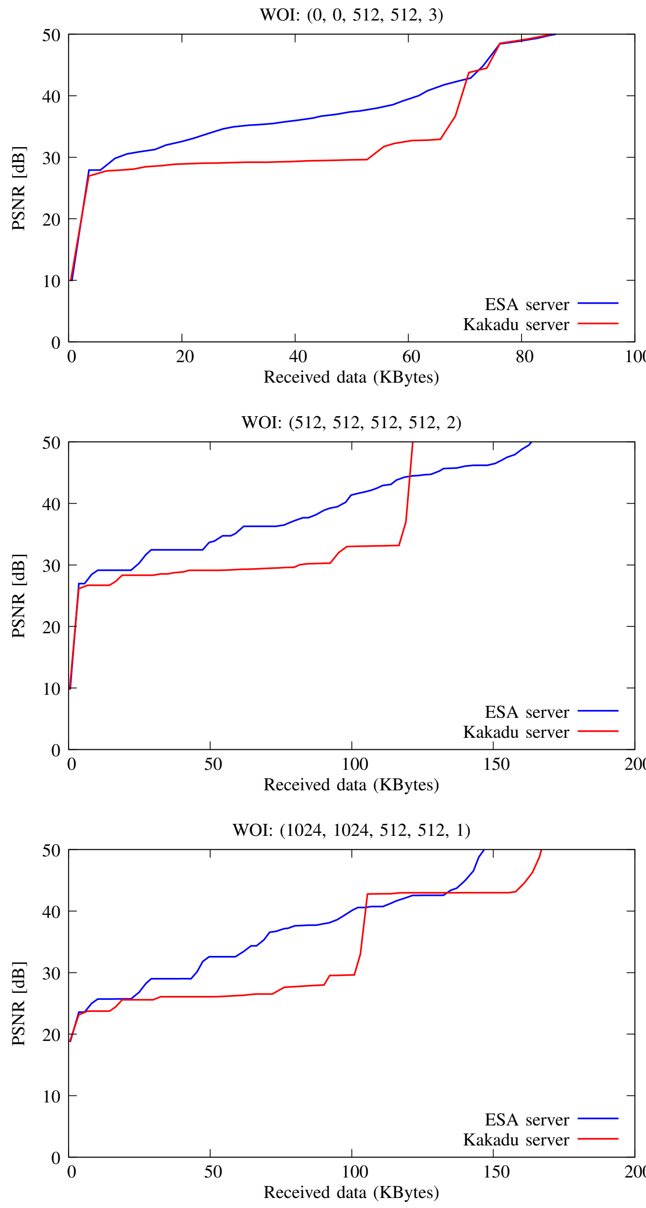

Fig. 3. Comparison in terms of PSNR vs. data received between the Kakadu server and the ESA server for a sequence of three WOIs.

In the context of the remote browsing systems, where a JPIP server is located, it is also very interesting to evaluate the quality of the reconstruction of the served WOI, measured by means of the PSNR [dB] versus the data received in the client side. This measurement is related to the user experience because the user always wants to see the best quality as soon as possible.

The sequence of WOIs $(x, y, width, height, res.level)$ that has been used is as follows: $(0, 0, 512, 512, 3)$, $(512, 512, 512, 512, 2)$, $(1024, 1024, 512, 512, 1)$. It corresponds to a common user sequence using the JHelioviewer application and making zoom in a corner. Fig. IV shows the rate-distortion curve generated by ESA server and Kakadu server.

As it can be observed, the ESA server gives better results. This is consequence of the way it transmits the image data, explained in the previous section. The Kakadu server seems to use a different packet progression.

In the hardest scenario where the ESA server was tested 1500 simultaneous connections were attended, serving a total of 1500000 image files (generated with the compression parameters commented before) in parallel. The Kakadu server is not able to support this load.

V. Conclusions

The evaluation results show that the ESA JPIP server is better than the server provided by the Kakadu package, in terms of scalability (memory consumption and CPU usage) as well as in terms of rate-distortion. Its open-source license allows to be used, maintained and improved freely by the Internet community. Therefore, this development is currently one of the best options in those contexts where a JPIP server is required.

References

[1] International Organization for Standardization, "Information Technology - JPEG 2000 Image Coding System - Core Coding System," ISO/IEC 15444-1:2004, September 2004.

[2] D. S. Taubman and R. Prandolim, "Architecture, Philosophy and Perfomance of JPIP: Internet Protocol Standard for JPEG2000," in *International Symposium on Visual Communications and Image Processing*, Julio 2003, vol. 5150, pp. 649–663.

[3] International Organization for Standardization, "Information Technology - JPEG 2000 Image Coding System - Interactivity Tools, APIs and Protocols," ISO/IEC 15444-9:2005, November 2005.

[4] V. Tuominen and J. Isola, "The application of JPEG 2000 in virtual microscopy," *Journal of Digital Imaging*, 2007.

[5] K. Krishnan, M.W. Marcellin, A. Bilgin, and M.S. Nadar, "Efficient transmission of compressed data for remote volume visualization," *IEEE Transactions on Medical Imaging*, vol. 25, pp. 1189–1199, September 2006.

[6] D. Müeller, B. Fleck, G. Dimitoglou, B. W. Caplins, D. E. Amadigwe, J. P. Garcia Ortiz, A. Alexanderian B. Wamsler, V. Keith Hughitt, and J. Ireland, "JHelioviewer: Visualizing large sets of solar images using JPEG 2000," *Computing in Science and Engineering*, vol. 11, no. 5, pp. 38–47, September 2009.

[7] W. Pesnell, "The Solar Dynamics Observatory: Your eye on the Sun," in *37th COSPAR Scientific Assembly*, 2008, vol. 37 of *COSPAR, Plenary Meeting*, pp. 2412–+.

[8] "Kakadu JPEG 2000 SDK," http://www.kakadusoftware.com.

[9] "The 2kan project," http://www.2kan.org.

[10] "OpenJPIP - Open-source C-Library for JPEG 2000," http://code.google.com/p/openjpeg/wiki/JPIP.

[11] "Cadi software," http://gici.uab.es/CADI.

[12] "ESA JPIP Server," https://launchpad.net/esajpip.

[13] D. Carrera, V. Beltran, J. Torres, and E. Ayguade, "A hybrid web server architecture for e-commerce applications," in *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on*, july 2005, vol. 1, pp. 182–188.

[14] "Apache HTTP server project," http://httpd.apache.org.

[15] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton, "Comparing the performance of web server architectures," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 231–243, March 2007.

[16] V.S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An efficient and portable Web server," in *USENIX 1999 Annual Technical Conference*, June 1999.

[17] Gyu Sang Choi, Jin-Ha Kim, Deniz Ersoz, and Chita R. Das, "A multi-threaded pipelined web server architecture for smp/soc machines," in *Proceedings of the 14th international conference on World Wide Web*, New York, NY, USA, 2005, pp. 730–739, ACM.

[18] J.P.G. Ortiz, V.G. Ruiz, M.F. Lopez, and I. Garcia, "Interactive transmission of JPEG2000 images using web proxy caching," *IEEE Transactions on Multimedia*, vol. 10, June 2008.