A Distributed Approach to Retrieving JPEG Pictures in Portable Hypermedia Documents

Dick C.A. Bulterman and Dik T. Winter

CWI (Centrum voor Wiskunde en Informatica)
P.O. Box 4079
1009 AB Amsterdam
The Netherlands

E-mail: `Dick.Bulterman@cwi.nl`

Abstract

In this paper, we single out one of the problem aspects of multimedia: how can one efficiently store high-quality picture information in a manner that does not make its retrieval characteristics incompatible with the needs of a (distributed) multimedia application. These needs include: timely access to data, predictable access to common (i.e., network) resources and ease in user specification of information. Our approach to solving this problem is to define adaptive data objects that adjust the amount and type of information given to an application as a function of resource availability. The key to our approach is that we transparently adapt the information presented to the application based on a set of pre-specified conditions that were defined by the application at author time. We discuss this work in the context of a parallel JPEG image decoder that provides adaptive images (with respect to data content and image representation) based on a transparent client/server negotiation scheme. Our work is based on the Amsterdam Multimedia Framework (AMF), a partitioning of control operations for supporting distributed multimedia. The parallel JPEG algorithm, AMF and the negotiated control algorithm are explained.

1 Introduction

The promise of multimedia computing is that it will enrich the quality of the user/computer interface. The reality of multimedia computing is that providing "good" multimedia presentations is a difficult task that few users (and fewer computing platforms) can effectively master. This is especially true of distributed multimedia systems in which multiple independent applications need to get at information objects located on several sets of semi-autonomous hosts. Here, information needs to be retrieved in a format suitable for the target system within the timing constraints of a particular application.

One simple model of activity in a distributed environment is that of a client that requests multimedia data from a server. If that data consists of an image encoded in the JPEG format [1], the server would have two options in the way that it processed the client's request: it could decode the image locally and send the result over the network to the client, or it could send the encoded image to the client, who would decode it upon receipt. The advantage of the server-based decoding is that it justifies the development of special-purpose software and hardware for speeding up the decoding process; it also allows a server to restrict the data it sends to the needs of the client. (That is, if the client only supported one-bit images, the server could transform the image and send only the reduced data to the client. This assumes, of course, that the server knows something about the configuration of the client.) The advantage of client-based decoding is that it can reduce the load on the environment's transmission architecture, at the expense of increasing processing on the client—something that could be useful if the system we heavily loaded.

If we take this simple client/server model and expand it to an environment in which many applications need to access remote facilities over a common network, the resource allocation decision-making process becomes much more complex. We represent this situation in the Amsterdam Multimedia Framework (AMF) [2], as is shown in Fig. 1. In this framework, many applications (AP) communicate with *adaptive information objects* (AIOs) via an infrastructure that is managed by a set of local operating systems and one (distributed) global operating system. The LOS's and GOS handle resource allocation, while the APs and AIOs request and deliver information, respectively. Note that the AMF does not solve the multimedia data transfer problem, it simply characterizes the components in a multimedia environment and it indicates their interactions. Individual models still need to be developed that implement the general functionality of the framework.

In this paper, we discuss an implementation model that explores the adaptive nature of the AIO. One of the things that makes the AIOs adaptive is an ability to deliver different data representations of the same "information object" based on, among other things, the load of the server controlling the object, the load on the client requesting the object, the load on the resources connecting the two, and the general characteristics of the client. We consider how such an adaptive process works in the context of an AIO that provides access to a set of JPEG-encoded images. We start by reviewing characteristics of JPEG images. We then discuss the structure of an AIO that implements, among other forms, a parallel implementation of a JPEG decoding algorithm. Next, we discuss a transparent client/server model for negotiating access to various representations of the JPEG image based on the availability of network resources. We conclude with a discussion of the open problems in this area, and a discussion of our plans for future work.

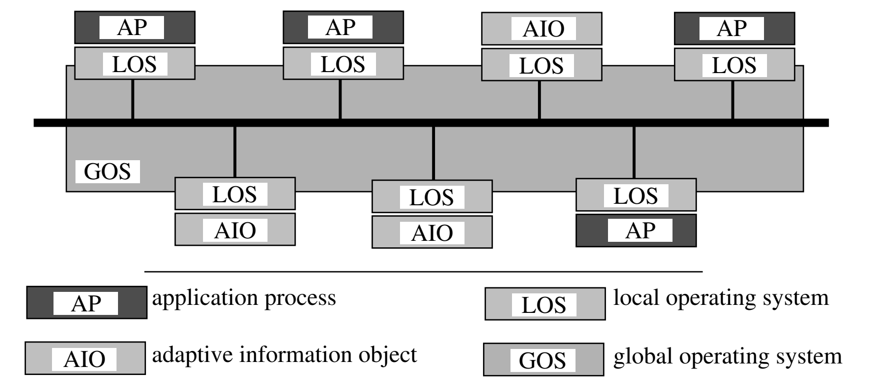


Figure 1 *AMF "active" components.*

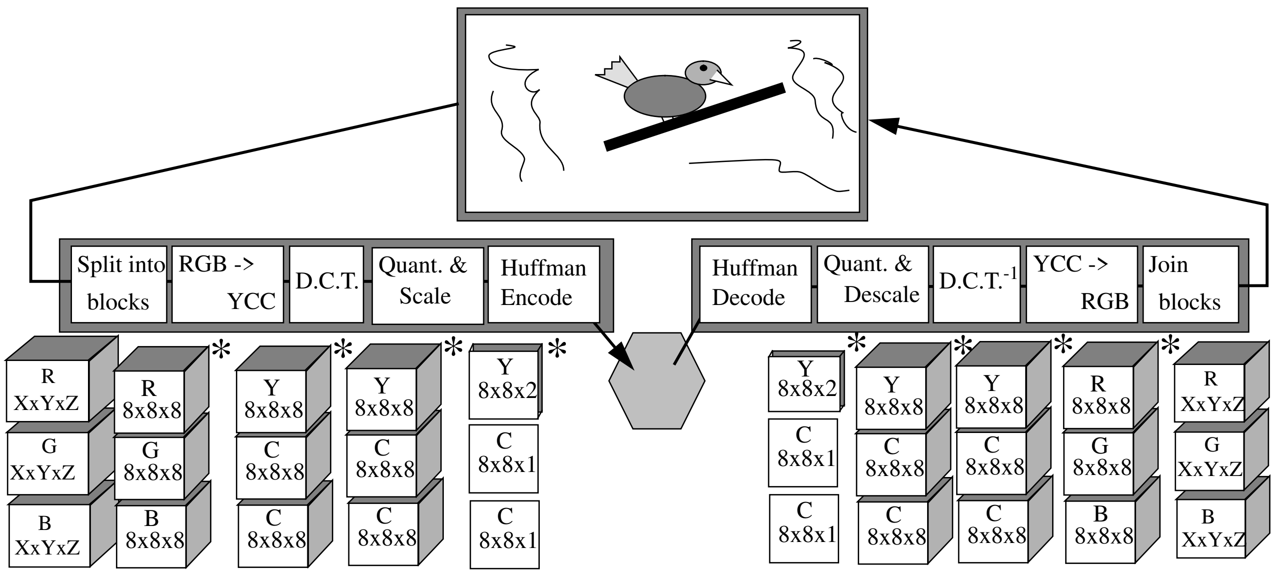


Figure 2 *Encoding and decoding JPEG images.*

2 Processing JPEG Pictures

In order to understand the structure of a JPEG AIO, we briefly discuss JPEG picture encoding and decoding in this section. We begin with the general algorithm and then show how this has been parallelized to form the core of a JPEG AIO.

General algorithm

The general JPEG algorithm is illustrated in Fig. 2. (This algorithm is derived from a public-domain JPEG implementation [3].) An image is split into a number of $8\times8\times8$ blocks, each of which undergoes a color conversion, a discrete cosine transformation and a quantification and scale operation, reducing the image to a set of 8*8*4 bit blocks. These blocks are then Huffman encoded and stored. Image decoding is performed by doing the inverse of these operations. The efficiency of the image encoding is a function of the number of colors used, the makeup of the image and the filtering techniques used. For an image of $1K\times1K\times24$, a compression by a factor of 25-30 is typical. Note that image encoding is a time-consuming process that is typically performed by an image supplier. Image decoding is usually done by an image user. We concentrate on decoding in this paper.

Parallelizing JPEG

Our interest in JPEG processing is the construction of a JPEG AIO that can serve as a central decoding resource to a diverse set of applications, running on a diverse set of host computers. Given this perspective, we felt it was useful to experiment with a parallelized version of JPEG, implemented on a Cray-S/MP computer server.[1] Our parallel

1. The Cray S/MP is a computer that supports multiple Sparc-based front-ends and an array of Intel i860-based APP's as a computing backend. The configuration we used had one Sparc front-end CPU and a 7×4 array of APP's.

decomposition was based on the following distribution of effort in decoding a test set of JPEG images:

Step	*% Effort*	*Step*	*% Effort*	*Step*	*% Effort*
DCT^{-1}	36	YCC->RGB	16	Descaling	8
Joining Blocks	18	Huffman Decoding	11	Miscl. I/O	6

The resulting decoder decomposition is shown in Fig. 3. The version is not fully optimized, but it provides a relatively fast implementation of a JPEG decoder that is an attractive alternative to doing decoding on a local workstation.

3 Integrating JPEG Processing in a Distributed Environment

In the context of an AIO model, the JPEG decoder needs to perform several functions beyond the “simple” decoding of JPEG images. For example, some of the requesting workstations may have hardware decoding facilities, and may therefore only need the raw image. Other workstations may not have full 24-bit color facilities, so the resulting image should be mapped to the facilities of the workstations. More importantly, there may be times when the network load would make the delivery of full-scale decompositions impractical—at such times, it would be useful if an alternative, low-impact version of the picture were available.

Each of these types of transformations could be handled explicitly by an application that checked various implementation parameters as part of a data-request operation. For some transformations, this makes sense: an application may want to select a particular representation at a particular time. Often however, the application may not want to worry about the state of the system when that application is run. Consider the case of an application that needs to run on two instance of one vendor’s workstation; if one instance has hardware JPEG support and the other one doesn’t, the application may feel that it is up to the “system” to decide if a decoded or encoded JPEG image is retrieved from a server. The same is true if an alternative representation of a particular picture needs to be sent because of bandwidth limitations on the network—the application *could* figure this out, but it probably should not have to do so.

One of the goals of our work has been to investigate application specification techniques in which “the system” selects the representation of a data object *transparently* for the

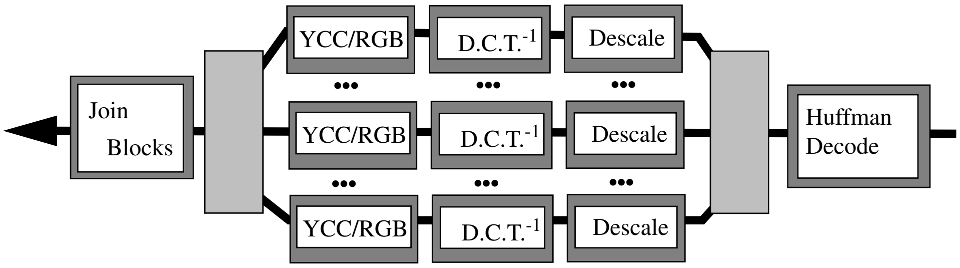


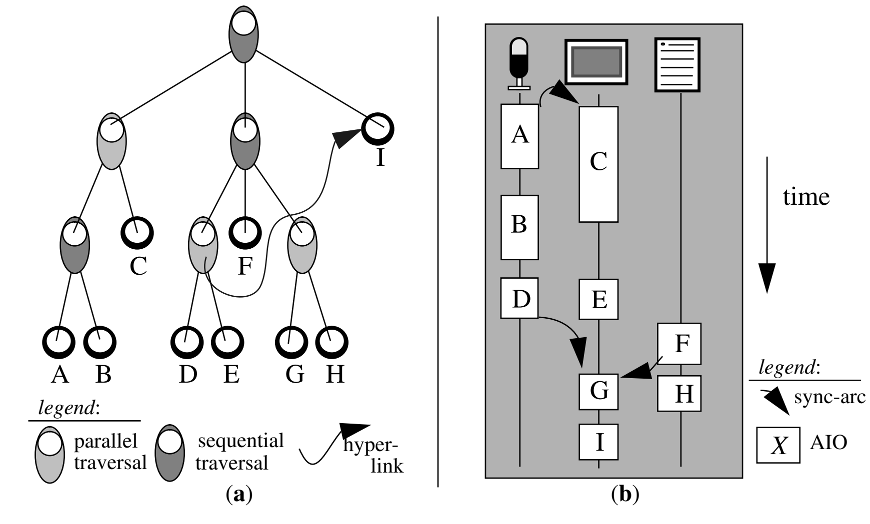
Figure 3 *Parallelizing JPEG image decoding.*

application. Ultimately, this selection should be performed by the AIO, LOS and GOS, under general control of the application (see Fig. 1); currently, we are investigating this transparency using a client/server negotiation model that is hidden from the application, but which sits on top of a operating system and networked infrastructure.

In this section, we review our current results in providing AIO functionality. We start by showing how applications specify information objects and then discuss a negotiation model that selects an “appropriate” data object for the application. We finish the section with a review of the performance consequences of our approach.

Specifying Adaptive Information Objects

There are many ways that an application program could refer to an (adaptive) information object: the object could be saved as a file or it could be referenced as a database object. The approach that we take is to have the application author define a document specification that gives a document logical hierarchy and a mapping of abstract objects to AIOs that reside on particular servers. (See Fig. 4.) This approach allows a user to decouple the logical relationship among information objects from their particular data characteristics. In Fig.4(a), we see an application that refers to 8 information objects (leaf nodes A-I). In 4(b), the nodes are mapped to virtual channels, and placed on an activation timeline. (Note that our authoring system does most of this timeline placement automatically [4,5].) In addition,4(b) shows a number of synchronization arcs that are used to indicate (relative) presentation constraints of a particular object.



a: A CMIF specification describes a document as a *hierarchy* of components, each of which is of a specific (possibly composite) media type. The hierarchy is traversed either sequentially or in parallel. Note that hyperlinks and other user interactions such as *fast forward, reverse, replay*, can change the order of tree traversal dynamically.

b: The hierarchy in (a) is mapped to a set of (virtual) *channels,* which relate the logical components to (virtual) output devices. Each logical block is placed on a channel and is associated with an AIO. Fine-grain synchronization can be specified using *synchronization arcs*, shown between nodes A/B, D/G and F/G. At run-time, channels may be active or inactive, influencing document synchronization.

Figure 4 *CMIF in a nutshell.*

It is our long-term intention that the specification will get passed among all of the components of the AMF, each of which will pick out the information it needs to support the application (and all others active in the environment at the same time). In the current version of our work, this global function is replaced by a separate client and server pair. This pair negotiate the format of the information to be used to satisfy a particular object reference based on the characteristics of the target system, the load on the network, the types of alternative representations that the client will accept, etc. In order to model the intended AIO role, the client and server act transparently to the application program in negotiating for representations from the AIO. This transparent interaction is important because it offers an opportunity for the system to respond quickly to transient conditions in the environment. (Consider the alternative: the AIO, upon sensing an unacceptable load on the network, messaged the application, which in turn queried the user, who then would select an alternative representation, which would be communicated to the AIO by the client ... . In this case, the original transient state of the environment that prompted the AIO's initial concern would probably have totally changed!)

Instead of interacting with the user at the moment a problem was sensed, our approach allows the user to prespecify a number of alternative actions that could be taken under a set of potential resource-use situations. (In all probability, most of the actions would be set by default values for the options.) This is done by specifying the constraints under which the applications operated in one of CMIF's synchronization arcs (see Fig. 4(b)).

Three example constraint/options pairs relevant to JPEG images are:

- *processing of base request*: if the request can not be satisfied under the specified timing constraints of the application (given a particular network loading), the request can be skipped or the rest of the presentation can be delayed;
- *selection of image representation*: if alternatives exist, must this reference to the object always be an image or is an alternative data form (such as a text or audio description) also acceptable;
- *selection of image resolution*: if the network bandwidth is limited, must the maximum resolution always be sent, or can the highest resolution available "under the circumstances" be substituted;

This list is only partial: other options could include the server's ability to match the image with the hardware on the target system or the use of local/remote decoding.

Given a well-defined set of options, a protocol could be defined that implements the selection of an image under the circumstances that exist at the moment the image is required, rather than at the time the application was authored.

A Transparent Client/Server Negotiation Model

The general negotiation model we use is shown in Fig. 5, The process starts when the *player* [6] encounters a request for an image from the application. (This request is the result of referencing an AIO node that corresponds to a JPEG-encoded object on a particular server.) Client code on the application's hosts sends a message to the server *(a)* requesting a particular object, plus it sends a vector of application-specified constraints (as specified by the application), as well as a state vector indicating resource characteristics of the client's machine (including resolution, color-map depth, etc.). The server analyzes the current state of the interconnection environment plus the load on the server, and makes a first request to the AIO for a particular type of representation of the object. If the AIO responds with the requested data object, it is immediately returned to the

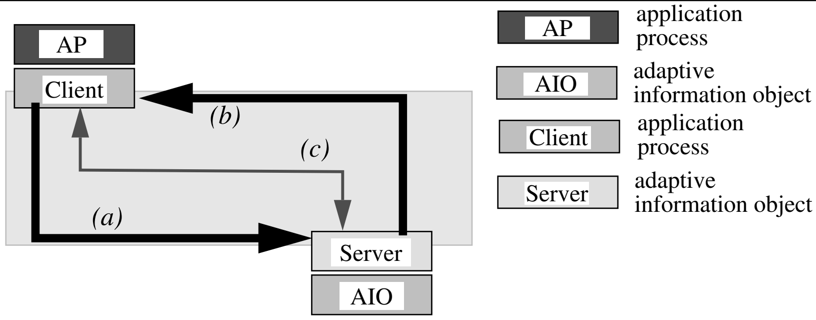


Figure 5 *The AP, AIO and surrogate client and server components.*

application *(b)*. (This will be the usual case.) If, on the other hand, this is not possible, the AIO may offer an alternative representation, based on the application's constraint vector. If the AIO cannot support any of the options, it can return a status result that is passed back to the client *(c)*; the client can either decide how to act on the status directly, or it can cause the application to query the user for a selection.

Note that if the AIO passes back an alternative representation of an image, the client must inform the player that different resources may be necessary to support the request. This does not happen by magic: the player must realize that a picture channel may need to (temporarily) be replaced by a sound or text channel to support the mapping.

Performance Impact

The processing of negotiated requests requires extra authoring time and execution-time overhead during accessing an object. For purely local objects, it is probably not worth the effort to select alternative representations, since resource availability will not vary widely during different runs of an application. For a distributed case, however, most varying resource conditions are beyond the control and knowledge of the application; this makes surrogate representation control attractive. In addition, the possibility that a particular application may be transported to a different host (with different local resource availability) is also present, providing a second incentive to study means of adapting data.

The following table shows two sets of performance figures for the decoding of images.

Size	*a: Workstation Decoding (Seconds)*	*b: Server Decoding (Seconds)*
125 x 125	0.39	0.25
423 x 613	5.10	3.80
681 x 900	9.27	7.12
1152 x 900	16.80	11.75

In (a), we see the time required to decode a series of 24-bit JPEG images on a workstation,[1] including transfer time from a server. Numbers in column (b) show the time required to decode an image on the server using the parallel implementation described

1. In this case, a Silicon Graphics Indigo with R3000 CPU and 16MB of main memory, using an Ethernet connection to the server.

above, including transfer time to the display host. While these times are by no means optimal, they do indicate that, in general, a special-purpose server can have considerable flexibility in deciding if a particular representation should be chosen and decoded. Even for small images, there is sufficient time to allow all negotiation to take place *and* decode and transfer an image. Note that these figures do not show the extra performance benefit of actually adapting the data for a given picture; under these circumstances, overall performance increases.

4 Directions for Future Work

The work presented in this paper represents a first step in integrating a negotiation-based adaptive object server into our multimedia environment. From these experiments, we can conclude that providing an adaptive object server can be useful if there is a large performance disparity between performing operations in a local environment vs. performing these operations at a central server. More importantly, however, this work gives us some insight into the ways that an application can pre-specify a number of options that could govern the operation of that application; given these options, the support environment can then adapt the presentation of media objects in a transparent manner.

In many ways, the processing of images presents an easy domain to model with an AIO: it is simple to build a text description describing the image, and it is (relatively) easy to produce reduced-resolution versions of any given image ‘on the fly.’ For other media domains, the task is harder: how does one build alternative representations of video or audio (other than by producing a text summary). In these cases, the best benefit of our adaptive strategy may be to define a structured interface to operations such as sub-sampling of data. Here, the greatest performance benefit will be that a user does not need to be interactively queried every time a dip in network resource availability occurs.

References

[1] *Gregory K. Wallace,* “The JPEG Still Picture Compression Standard,” Comm. ACM, V34 N4, April 1991, pp. 30-44.

[2] *Dick C. A. Bulterman*, “Synchronization of Multi-Sourced Multimedia Data for Heterogeneous Target Systems”, Proceedings of the 3rd International Workshop on Network and OS Support for Digital Audio/Video, Springer Verlag, 1992.

[3] *The Independent JPEG Group,* Public Domain JPEG implementation, Version 3B, 2-Aug-92. For information, contact: jpeg-info@uunet.uu.net.

[4] *Dick C. A. Bulterman, Guido van Rossum and Robert van Liere*, “A Structure for Transportable, Dynamic Multimedia Documents”, USENIX conference June 1991 Nashville TN, pp. 137 - 155.

[5] *Lynda Hardman, Dick C. A. Bulterman, Guido van Rossum*, “Structured Multimedia Authoring”, submitted to ACM Multimedia’93, USA 1993.

[6] *Guido van Rossum*, *Jack Jansen, Sjoerd Mullender and Dick C. A. Bulterman,* “CMIFed: A Presentation Environment for Portable Hypermedia Documents”, submitted to ACM Multimedia’93, USA 1993.