TERASERVER: A SCALABLE TERABYTE CONTINUOUS MEDIA SERVER

THÈSE N^{O} 2415 (2001)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR

Joaquín Tárraga Giménez

Ingénieur informaticien, Faculté d'Informatique, Université
Polytechnique de Valence, Espagne
de nationalité espagnole

Lausanne, EPFL
2001

To my wife, Maria Carmen,
my best friend

Table of Contents

Acknowledgements

I wish to express my deepest gratitude to Professor Roger David Hersch, the head of the Peripheral Systems Laboratory (LSP) of the EPFL for trusting me and giving me the opportunity to complete this work in his laboratory, for his continuous support and guidance throughout this research and for spending hours reading and commenting earlier drafts of this dissertation.

I would like to give special thanks to Dr Benoît Gennart for his patience, for his technical expertise, for his help in correcting bugs, and for all the enlightening discussions we had which helped me clarify my ideas.

Dr Vincent Messerli and Dr Benoît Gennart are the main authors of the CAP/PS^2 parallelization framework. Without their own work, the present research would not have been possible.

I would like to thank the members of the Peripheral System Laboratory, among them Dr Oscar Figueiredo, Marc Mazzariol, Jean-Christophe Bessaud, Dr Patrick Emmel, Sebastian Gerlach, Olivier Courtois, for their technical collaboration and support and for their friendship.

Many thanks to Yvette Fishman and Fabienne Allaire-Séchaud for their daily help, and for their invaluable contribution in making our working environment so pleasant.

Thanks to my family and friends, in Spain and in Switzerland, for their patience and understanding, especially in the last period of my work.

Finally I would like to thank the Swiss National Fund and EPFL for funding this research.

Summary

Parallel servers for I/O and compute intensive continuous media applications are difficult to develop. A server application comprises many threads located in different address spaces as well as files striped over multiple disks located on different computers. This research presents new methods and tools to build parallel I/O- and compute-intensive continuous media server applications based on commodity components (PCs, SCSI disks, Fast Ethernet, Windows NT/2000). This dissertation describes a customizable parallel stream server based on a computer-aided parallelization tool (CAP) and on a library of striped file components enabling the combination of pipelined parallel disk access and processing operations. The parallel stream server offers a library of continuous media services (e.g. admission control, resource reservation, disk scheduling) to support parallel streaming. CAP allows to combine these continuous media services with application-specific stream operations in order to create efficient pipelined parallel continous media servers.

As an example of such a parallel continuous media server, this thesis presents the 4D beating heart slice server application, which supports the visualization at a specified rate of freely oriented slices from a 4D beating heart volume (one 3D volume per time sample). This server application requires both a high I/O throughput for accessing from disks the set of 4D sub-volumes intersecting the desired slices and a large amount of processing power to extract these slices and to resample them into the display grid. Parallel processing on several PCs and parallel access to many independent disks offers the potential of scalable processing power and scalable disk access bandwith. The presented 4D beating heart application suggests that powerful continuous media server applications can be built on top of a cluster of PCs connected to SCSI disks.

In order to extent the storage capacities of multi-PC multi-disk servers, this thesis also develops paradigms allowing to integrate optical jukeboxes into cluster-based server architectures. A scalable server based on optical disk jukeboxes offers highly accessible terabyte storage capacities for digital libraries, image and multimedia repositories with significant cost advantages. Server scalability can be achieved by increasing the number of optical disks, drive units, robotic devices and server PCs. The master server PC incorporates a shadow directory tree containing the directory trees of all present optical disk files. In order to offer higher throughputs, the scalable terabyte server allows to access large files striped over multiple optical disks loaded into different optical disk drives. Using magnetic disks as a cache for optical disk files allows to significantly increase the number of files that can be served at the same time.

Finally, this thesis presents a terabyte server comprising one master PC and two slave PCs, each slave PC connected to one NSM Satellite jukebox, each one comprising four optical disk read-

ing units able to operate simultaneously. By running the 4D beating heart slice server application as well as Visible Human slice server application, the terabyte server offers advanced services in addition to its highly accessible terabyte storage capacities. The terabyte server has also been interfaced to a Web server and offers its multiple services to Internet clients.

Part of this work has been published at the ACM Multimedia 1999 conference.

Résumé

Il est difficile de développer des serveurs parallèles pour applications multimedia à flux continu nécessitant performances et débits d’entrée-sortie élevés. Un serveur d'applications parallèles comporte plusieurs processus répartis dans différents espaces d'adressage ainsi que des fichiers distribués sur des disques différents (striped files). Nous proposons une nouvelle approche pour développer des applications de media en flux continu effectuant en parallèle des opérations de traitement et d'entrée-sorties sur une grappe d’ordinateurs (PC cluster). A cette fin, nous présentons un serveur de flux parallèle basé sur l'outil CAP (Computer-Aided Parallelization) et sur une librairie de composants réutilisables pour l'accès à des fichiers parallèles. Ce serveur permet de combiner des tâches de traitement et d'entrée-sorties de manière pipeline parallèle. Le serveur parallèle incorpore une librairie de services pour la gestion de media en flux continu. Ces services comprennent l’admission de requêtes, l’allocation et la gestion de ressources ainsi que l’ordonnancement des accès aux disques. CAP permet de combiner ces services avec les besoins spécifiques d'une application afin de créer des nouveaux services multimedia parallèles à flux continus.

Comme exemple de service d'application parallèle à flux continu exigeant puissance de traitement et débits d’entrée-sortie élevés, cette thèse présente la réalisation d'un serveur de coupes d'un cœur qui bat. Ce service permet de visualiser à la cadence spécifiée des coupes obliques extraites des volumes 3D du coeur en mouvement. Le service “Beating heart” montre qu’il est possible de développer des applications performantes de média parallèle à flux continu sur grappe d’ordinateurs PC connectés à des disques SCSI.

Afin d'accroître la capacité de stockage du serveur (multi-PC, multi-disques), cette thèse énonce les paradigmes qui permettent d'intégrer des jukeboxes à disques optiques. Un serveur extensible basé sur de tels jukeboxes permet d'étendre la capacité de stockage au delà du teraoctet pour des serveurs de librairies digitales, des serveurs d’images et des serveurs multimedia tout en réduisant les coûts. Le serveur peut être étendu en augmentant le nombre de disques optiques, d’unités de lecture, de robots jukebox et d’ordinateurs PC serveurs. Le PC serveur maître contient le répertoire hiérarchique de tous les répertoires de chacun des disques optiques (shadow directory tree). Afin d'offrir un débit optimal, le serveur accède à des fichiers distribué sur plusieurs disques optiques (striped files). En utilisant des disques magnétiques comme cache pour les fichiers stockés sur disques optiques, plus de fichiers peuvent être servis en même temps. Nous analysons les performances d’un serveur composé d'un PC maître et de PC esclaves connectés chacun à un NSM Satellite jukebox incorporant des unités de lecture capables de travailler simultanément. Le serveur de coupes du cœur qui bat ainsi que le serveur de coupes du Visible Human s’exécutent sur ce serveur et démontrent sa capacité à offrir services avancés

et accès parallèle à des larges volumes de données. Le serveur de stockage à base de jukebox multiples a également été interfacé à un serveur Web afin d'offrir ses services sur Internet.

Une partie de cette thèse a été publiée a la conférence internationale ACM Multimedia 1999.

1 Introduction

Since the early nineties, there has been considerable interest in developing concepts and computer architectures for continuous media applications, especially for video-on-demand (VoD) services. Recent research has focussed either on the design of single computer video servers [57, 67] or on the design of large parallel video servers able to serve thousands of client viewers [8, 9, 55]. Besides declustering video streams across several disks, these servers generally incorporate dedicated admission control, real-time disk scheduling algorithms, disk access deadlines, multi-stream synchronization mechanisms, quality of service management and resource reservation ensuring that the hard real-time video stream delivery constraints (guaranteed throughput, bounded jitter) can be met. Although VoD servers may require a certain processing power (e.g. to transcode the video stream from one compression standard to another, or to compute from an original video stream a derived stream with smaller size frames and a lower frame rate), they are mainly characteristized by their high I/O bandwidth requirements.

Thanks to the increased processing power of microprocessors, new continuous media applications requiring as much processing power as I/O bandwidth can be conceived. For example, multimedia applications should allow to navigate in real-time across large 3D volumes or extract a stream of slices from a 3D image volume varying in time. As an example of such an application, this thesis presents the *4D beating heart slice server application* [81], which supports the visualization at a specified rate of freely oriented slices extracted from a 4D beating heart volume (one 3D volume per time sample). This server application requires both a high I/O throughput for accessing from disks the set of 4D sub-volumes intersecting the desired slices and a large amount of processing power to extract these slices and to resample them into a display grid. With the emergence of fast commodity networks such as Fast Ethernet (100 Mbits/s), running parallel server applications on a cluster of PCs is now possible. Parallel processing on several PCs and parallel access to many independent disks offers the potential of scalable processing power and scalable disk access bandwidth.

Creating a parallel server application requires the creation and management of processes, threads and communication channels. In addition, accessing simultaneously and in parallel many disks located on different computers requires appropriate parallel file system support, i.e., means of striping a global file onto a set of disks located on different computers and of managing meta-information. To minimize communications, processing operations should be executed on processors which are close to the disks where the data resides. An efficient utilization of the resources offered by the system (processors, disk, network) can be achieved by enabling the combination of pipelined parallel disk accesses and processing operations.

1 Introduction

In order to build parallel continuous media servers, we use a computer aided parallelization tool (CAP)[1] [35] and a library of striped file components[2] [60] that have been created for facilitating the development of parallel server applications running on distributed memory multi-processors, e.g. PCs connected by Fast Ethernet. We generate parallel server applications from a high-level description of threads, operations (sequential C++ functions) and from high-level parallel constructs specifying the flow of parameters and data between operations. Each thread incorporates an input token queue ensuring that communication occurs in parallel with computation. In addition, disk access operations are executed asynchronously with the help of callback functions.

This research presents new methods and tools to build parallel I/O- and compute- intensive continuous media server applications based on commodity components (PCs, SCSI disks, Fast Ethernet, Windows NT/2000). This dissertation describes a customizable parallel stream server, based on CAP and on the striped file component library. The parallel stream server offers a library of continuous media services (i.e. admission control, resource reservation, disk scheduling) to support parallel streaming. CAP allows to combine these continuous media services with application-specific stream operations in order to create efficient pipelined parallel continuous media servers, for example the 4D beating heart slice stream server described in Chapter 4.

In order to extend the storage capabilities of multi-PC multi-disk servers, we also develop the paradigms allowing to integrate optical jukeboxes into cluster-based server architectures. For large data server systems, optical jukebox storage systems are superior in terms of cost to storage systems based entirely on magnetic disks. Furthermore, optical jukeboxes offer significantly lower access times compared with robotic tape libraries.

In this work, we are interested in showing to which extent a server made of PCs, disks and multiple optical jukeboxes has the potential of becoming a scalable terabyte server. In particular, we will show how scalability can be achieved with an optical jukebox, i.e. how it can be expanded by increasing the number of optical disks, drive units, robotic devices and server PCs. This work analyses the potential bottlenecks of a jukebox server and computes its overall throughput, taking into account the robotic device, the optical disk drives as well as the client request arrival rate. An analytic performance model is described, extended by a simulation model and verified by performance measurements.

This dissertation is structured as follows. In the Chapter 2, we briefly describe the CAP Computer-Aided Parallelization tool and the library of striped file components facilitating the development of parallel applications running on distributed memory multi-processors multi-disk architectures. Chapter 3 describes the parallel stream server library for developing parallel I/O- and compute-intensive continuous media applications. It includes admission control, resource allocation, data streaming and synchronization issues. Chapter 4 describes the *4D beat-*

1. The CAP Computer-Aided Parallelization tool was conceived by Dr Benoit Gennart at the Peripheral System Laboratory, Computer Science Department, EPFL.
2. The library of striped file components was developed by Dr Vincent Messerli at the same place.

ing heart slice server application, a parallel continuous media server based on the parallel stream server library, the CAP tool and the library of striped file components. This server supports the extraction of freely oriented slices from a 4D beating heart volume (one 3D volume per time sample). Chapter 5 describes a scalable server architecture based on optical disk jukeboxes as an extension to the multi-PC multi-disk architectures. Chapter 6 describes a WEB server based on the multi-PC multi-jukebox architecture and evaluates its performance. Chapter 7 present the conclusions.

1 Introduction

2 Parallelization tools for I/O and Compute Intensive Applications

2.1 Introduction

This chapter describes two powerful tools for parallelizating I/O and compute intensive applications: the CAP computer-aided parallelization tool and the library of striped file components (PS^2). The CAP tool enables application programmers to create separately the serial program parts and specify the schedule of the program at a high-level of abstraction. This high-level parallel CAP program description specifies a macro-dataflow, i.e., a flow of data and parameters between operations running on the same or on different processors. CAP is designed to implement highly pipelined-parallel programs that are short and efficient. The library of striped file components (PS^2) enables application programmers to access files that are declustered across multiple disks distributed over several server nodes. The extensible parallel application development system comprises a set of compute threads that programmers can freely customize by incorporating application- or library-specific processing operations. Application programmers can, thanks to the CAP formalism, easily and elegantly extend the functionality of PS^2 by combining the predefined low-level striped file access components with application specific or library-specific processing operations in order to yield efficient pipelined parallel I/O and compute intensive applications.

2.2 The Computer-Aided Parallelization tool

The CAP language is a general-purpose parallel extension of C++ enabling application programmers to express the behavior of parallel program at a high-level of abstraction. Application programmers specify the set of threads (*a CAP process hierarchy*), which are present in the application, the processing operations offered by these threads, and the flow of data and parameters (*macro dataflow*) between operations. This specification completely defines how operations running on the same or on different computers are scheduled and which data and parameters each operation receives as input values and produces as output values. A macro dataflow described in CAP is graphically represented by a directed acyclic graph (DAG). Directed acyclic graphs can describe a large class of algorithms. Furthermore, they ensure that the generated code is deadlock free.

The CAP methodology consists of dividing a complex operation into several suboperations with data dependencies, and to assign each of the suboperations to a thread. The CAP programmer specifies in CAP the data dependencies between the suboperations, and assigns explicitly each suboperation to a thread. CAP operations are defined by a single input, a single output, and the computation that generates the output from the input. Input and output of operations are called *tokens* and are defined as C++ classes with serialization routines that enable the tokens to be packed or serialized, transferred across the network, and unpacked or deserialized. Communication occurs only when the output token of an operation is transferred to the input of another operation. An operation specified in CAP as a schedule of suboperations is called a *high level operation*. A high level operation specifies the assignment of suboperations to threads, and the data dependencies between suboperations. When two consecutive operations are assigned to different threads, the tokens are *redirected* from one thread to the other. As a result, high level operations also specify communications and synchronizations between s*equential operations*. A sequential operation, specified as a C/C++ routine, computes its output based on its input. A sequential operation is performed by a single thread and cannot incorporate any communication.

Each parallel CAP construct consists of a split function splitting an input request into sub-requests sent in a pipelined parallel manner to the operations of the available threads and of a merge function collecting the results. The merge function also acts as a synchronization means terminating the parallel CAP construct's execution and passing its result to the higher level program after the arrival of all sub-results (Fig. 2-1).

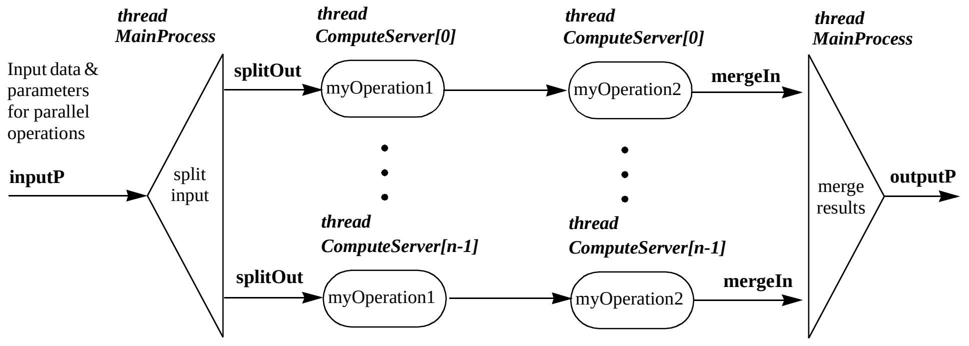
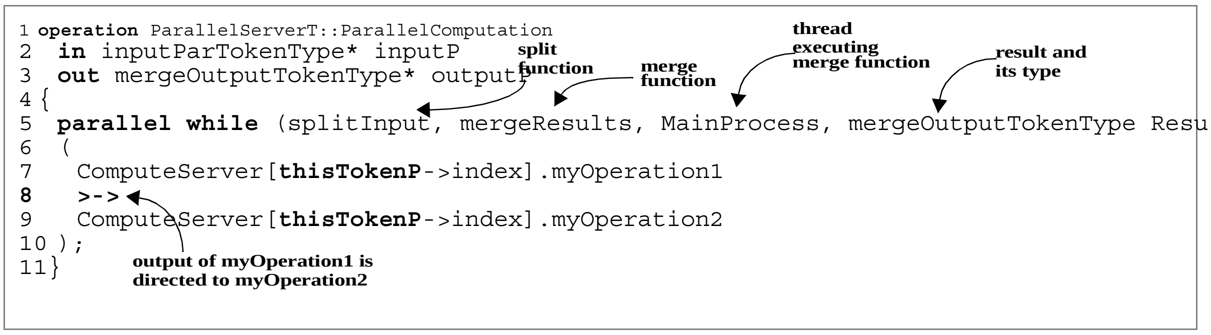


Figure 2-1: Graphical CAP specification: parallel operations are displayed as parallel horizontal branches, pipelined operations are operations located in the same horizontal branch

Parallel constructs are defined as high level operations and can be included in other higher level operations to ensure compositionality. The *parallel while* construct corresponding to the example of Fig. 2-1 has the syntax shown in Fig. 2-2. Besides *split* and *merge* functions, it incorporates two successive pipelined *operations* (*myOperation1* followed *by myOperation2*).

The *split* function is called repeatedly to split the input data into subparts which are distributed to the different compute server thread operations (*ComputeServer[i].myOperation1*). Each operation running in a different thread (*ComputeServer[i]*) receives as input the subpart sent by the split function or by the previous operation, processes this subpart and returns its subresult either to the next operation or to the merge function. The parallel construct specifies explicitly in which thread the merge function is executed (often in the same thread as the split function). It receives a number of subresults equal to the number of subparts sent by the split function. For



```
operation ParallelServerT::ParallelComputation
 in inputParTokenType* inputP
 out mergeOutputTokenType* outputP
{
 parallel while (splitInput, mergeResults, MainProcess, mergeOutputTokenType Resu
 (
  ComputeServer[thisTokenP->index].myOperation1
  >->
  ComputeServer[thisTokenP->index].myOperation2
 );
}
```

split function
merge function
thread executing merge function
result and its type
output of myOperation1 is directed to myOperation2

Figure 2-2: Syntax of a parallel while construct

further information on the CAP preprocessor language syntax, please consult the CAP tutorial [35]. The CAP specification of a parallel program is described in a simple formal language, an extension of C++. This specification is translated into a C++ source program, which, after compilation, runs on multiple processors according to a *configuration map* specifying the mapping of CAP threads onto the set of available processes and PCs. The macro data flow model which underlies the CAP approach has also been used successfully by the creators of the MENTAT parallel programming language [41, 42].

Thanks to the automatic compilation of the parallel application, the application programmer does not need to explicitly program the protocols to exchange data between parallel threads and to ensure their synchronizations. CAP also handles for a large part memory management and communication protocols, freeing the programmer from low level issues. CAP's runtime system ensures that tokens are transferred from one address space to another in a completely asynchronous manner (socket-based communication over TCP-IP). This ensures that correct pipelining is achieved, i.e. that data is transferred through the network or read from disks while previous data is being processed. In addition, CAP offers a high-level mechanism to regulating the flow of tokens circulating within a parallel operation in order to avoid that a split function generates tokens without stopping[1]. The flow-control mechanism guarantees that the split function generates tokens at the same rate a the merge function consumes them. Another important feature of the CAP tool is load-balancing. CAP allows the programmer to specify, at a high-level of abstraction, *static* or *dynamic* load-balancing[2] directives. The CAP kernel interprets these directives and balances the load of the specified tasks among the different processors.

The CAP approach works at a higher abstraction level than the commonly used parallel programming systems based on message passing (e.g. MPI [62], MPI-2 [63], PVM [5]). CAP

1. If the split function generates thousands of large tokens, the processor will be overloaded executing continuously the split function, memory will overflow by storing the generated tokens, and the network interface will saturate sending all these tokens. This will result in a noticeable performance degradation and later in a program crash with a "no more memory left" error message.
2. Static load-balancing consists of assigning statically some work to processors. In contrast to the static load-balancing, dynamic load-balancing distributes the load at run time, so when a processor runs out of work, it should get more work from another processor.

enables expressing explicitly the desired high-level parallel constructs. Due to the clean separation between parallel construct specification and the remaining sequential program parts, CAP programs are easier to debug and to maintain than programs which mix sequential instructions and message-passing function calls.

CAP distinguishes itself from DCOM or CORBA by the fact that data and parameters circulate in an asynchronous manner between operations. CAP programs therefore make a much better utilization of the underlying hardware, specially for parallel I/O and compute intensive applications.

```
process StripedFileServerT
{
subprocesses:
  InterfaceServerT InterfaceServer;
  ExtentFileServerT ExtentFileServer;
  ExtentServerT ExtentServer;
  ComputeServerT ComputeServer;
operations:
  OpenStripedFile in FileNameT InputP out FileDescriptorT OutputP;
  CloseStripedFile in FileDescriptorT InputP out ErrorT OutputP;
}; // end process StripedFileServerT

process ExtentServerT
{
operations:
 ReadExtent in ReadExtentRequestT InputP out ErrorT OutputP;
 WriteExtent in WriteExtentRequestT InputP out ErrorT OutputP;
}; // end process ExtentServerT
...
```

Listing 1.

(library of striped file components)

```
process StripedStreamServerT
{
subprocesses:
  ExtentStreamServerT ExtentStreamServer;
  HPComputeServerT HPComputeServer;
operations:
  OpenStripedStream in FileNameT InputP out StreamDescriptorT OutputP;
  CloseStripedStream in StreamDescriptorT InputP out ErrorT OutputP;
}; // end process StripedFileServerT
...
operation StripedStreamServerT::OpenStripedStream
in FileNameT InputP
out StreamDescriptorT OutputP
{
 ... // argument checking
 >->
 StripedFileServer.OpenStripedFile
 >->
... //do some work
}; // end operation OpenStripedStream
leaf operation ExtentServerT::ReadExtentWithDiskScheduling
in ExtentRequestT InputP
out ExtentT OutputP
{
 ... //read the extent according to the disk scheduling algorithm
}; // end leaf operation ReadExtentWithDiskScheduling
```

Listing 2.

(library of striped stream components

Figure 2-3: Creating modular and extensible parallel applications using CAP

The CAP computer-aided parallelization tool was mainly conceived by Dr. Benoit Gennart at EPFL's Peripheral Systems Laboratory. I developed the first CAP-oriented message passing system within the PVM environment [34][1]. In the context of this thesis, I have contributed to CAP by specifying features to be incorporated into CAP such as the possibility of creating multiple CAP process hierarchies and the specification of a new parallel CAP construct (the *parallel-while suspend* CAP construct) allowing tokens to be suspended during a period of time specified by a user-defined function. Using this CAP construct, application programmers can ensure the isochrone behavior of a continuous media server, i.e. the ability of delivering at constant time intervals the corresponding amounts of stream data. On the other hand, thanks to CAP's ability of creating different CAP process hierarchies, we can create modular and extensible parallel applications. For instance, we can create a first library comprising a first CAP process hierarchy (set of threads and operations) that implement a striped file access component (listing 1 of Fig. 2-3). Then, we can create a second library comprising a second CAP process hierarchy with a new set of threads and that implements specific operations for streaming data from striped files (listing 2 of Fig. 2-3). These specific operations are built up from the operations defined in the first CAP process hierarchy (e.g. in Fig. 2-3, lines 11-20 of listing 2). In addition, in the second library we can define operations for the first CAP process hierarchy without modifying the first library (e.g. in Fig. 2-3, lines 21-26 of listing 2).

2.3 The library of striped file components

The library of striped file components (PS^2) [60] enables application or library programmers to access files that are declustered across multiple disks distributed over several server nodes. PS^2 does not impose any declustering strategy on an application's data. Instead, it provides application or library programmers with the ability to easily and efficiently program their declustering strategy and to control the degree of parallelism exercised on every subsequent access according to their own needs using the CAP language[2]. This control is particularly important when implementing I/O-optimal algorithms [20]. To allow this behavior, a *striped file*, i.e. a file whose data is declustered across multiple disks, is composed of one or more *extent files*, i.e. local files or subfiles which may be directly addressed by the application using a 32-bit value called the *extent file index*. Each extent file resides entirely on a single disk. When a striped file is created, the programmer is asked to specify how many extent files the striped file contains and on which disks the extent files will be created. The number of extent files and their locations remain fixed throughout the life of the striped file. The *striping factor* of a striped file represents the number of extent files it contains, or in other words the number of disks across which it is declustered.

1. The current CAP-oriented message passing is based on threads and socket communication and was developed by Dr Vincent Messerli.
2. Of course, many application programmers will not want to handle low-level details such as the declustering algorithm and will even not program with the CAP language. We therefore anticipate that most end users will use high-level C/C++ libraries, e.g. a parallel streaming library that provides a variety of abstractions with appropriate declustering strategies, but hide the details of these strategies from the application programmers.

Each extent file is structured as a collection of *extents*. An extent within a particular extent file is addressable by a 32-bit value called the *local extent index*. An extent is a variable size block of data representing the unit of I/O transfer, i.e. applications may read or write entire extents only. In order to offer added flexibility for building C/C++ libraries on top of PS^2 library, an extent is decomposed into a variable size header and a variable size body. Unlike the number of extent files in a parallel file, the number of extents in an extent file is not fixed. Libraries and applications may add extents to or remove extents from an extent file at any time. There is no requirement that all extent files have the same number of extents, or that all extents have the same size.

Fig. 2-4 shows the two-dimensional striped file structure of PS^2. A particular extent is addressed using two 32-bit unsigned values: an extent file index ranges from 0 to N-1, where N is the striping factor, and a local extent index within that extent file which ranges from 0 to 2^{32}-1. Extents within an extent file are not necessarily stored by successive extent indices, i.e. an extent file may contain extents with scattered indices.

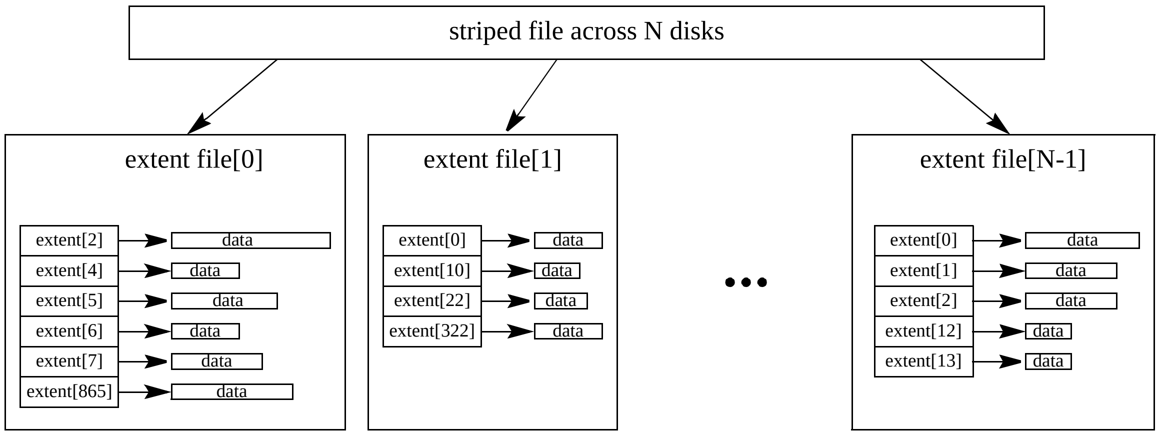


Figure 2-4: A striped file is composed of one or more extent files. Each extent file is structured as a collection of extents. An particular extent is directly addressed by its extent file index and its local extent index.

For the striped file management, the PS^2 library comprises the following threads:

- an *InterfaceServer* thread responsible for coordinating striped file directory operations[1], i.e. open striped file, close striped file, create striped file, delete striped file, create directory, delete directory and list directory,

1. Striped file operations implicate the coordination of the corresponding extent file operations, e.g. opening a striped file requires opening all the extent files making up that striped file.

- several *ExtentFileServer* threads offering extent file directory operations, i.e. open extent file, close extent file, create extent file, delete extent file, create extent file directory, delete extent file directory and list extent file directory,
- and several *ExtentServer* threads offering extent access operations, i.e. read extent, write extent and delete extent.

In addition to the file I/O threads, the PS^2 library offers several compute threads (called *ComputeServer* threads). These compute threads do not offer any default processing operations. They are customized, i.e. library programmers can freely extend the functionalities of compute threads by incorporating library-specific processing operations. Fig. 2-5 shows an example of how the PS^2 threads can be mapped onto a server architecture comprising N server nodes.

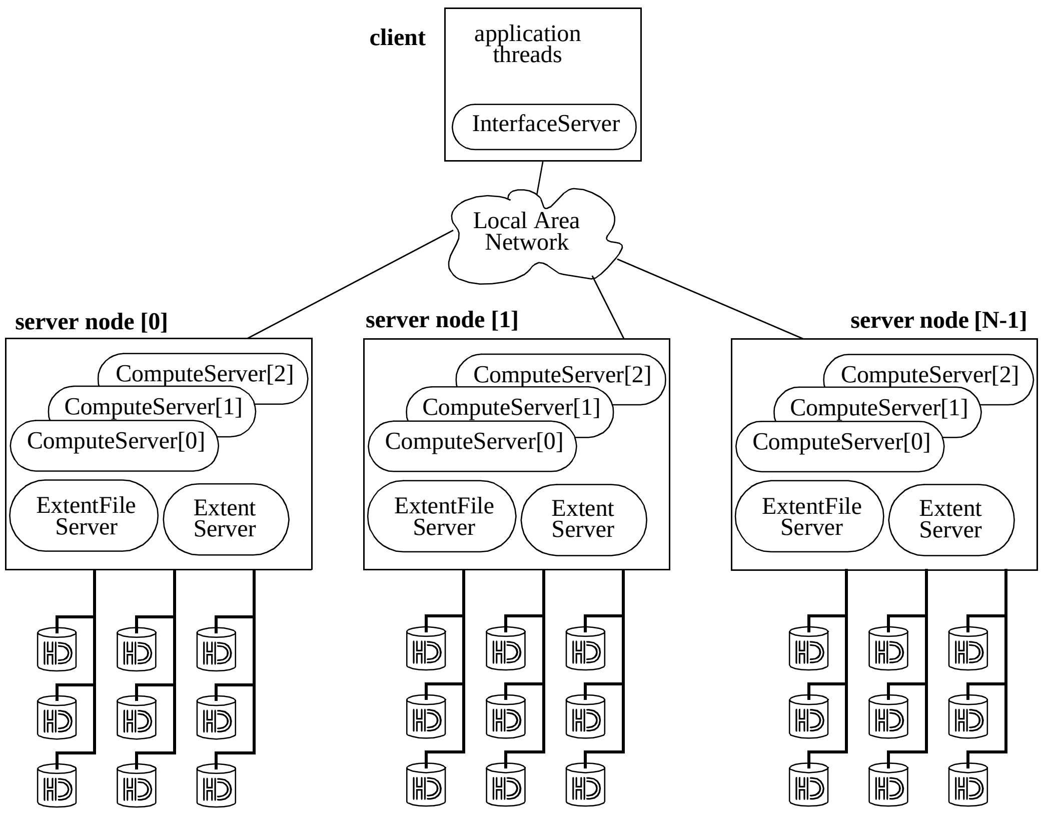


Figure 2-5: PS^2 threads and how they are mapped onto a server architecture comprising N server nodes.

The file I/O threads and the compute threads are distributed between the server nodes so as each contributing server node comprises at least one *ExtentServer* thread performing parallel disk accesses on local disks and at least one compute thread running library-specific processing operations. Library programmers can, thanks to the CAP formalism, easily and elegantly extend the functionalities of the PS^2 process hierarchy by combining the predefined low-level striped file access components offered by the *ExtentServer* threads (i.e. read and write extents) with the library-specific processing operations in order to yield efficient pipelined parallel I/O- and compute- intensive CAP operations. Then, these parallel CAP operations can be incorporated into C/C++ high-level libraries offering specific abstractions of striped files and processing routines on those abstractions.

To minimize the amount of data transferred between server nodes and client nodes, library-specific processing operations can, thanks to the PS^2 flexibility and the CAP formalism, be directly executed on the server nodes where data resides. A compute thread can process data that is read by its companion extent server thread located on the same server node in order to avoid superfluous data communication. Accordingly, developing a new parallel I/O- and compute-intensive application with a large data set declustered across multiple disks may consist of

1. dividing the application data set into extents,
2. striping these data extents across several disks,
3. on the server nodes, reading data extents from multiple disks,
4. on the server nodes, performing a library-specific processing operations on the extents previously read, e.g. extracting the desired information from the extents,
5. transmitting the processed data extracted from extents from the server nodes to the client node,
6. on the client node, merging the processed data extents into the application’s buffer.

Thanks to the CAP methodology, disk access operations offered by the *ExtentServer* threads (i.e. reading extents), library-specific processing operations performed by the *ComputeServer* threads (i.e. processing data extents), data communications (i.e. transmitting processed data extents to the client node), and collecting the results (i.e. merging the processed data extents) are executed in a pipelined parallel manner (Fig. 2-6).

In a pipelined parallel execution (Fig. 2-6), pipelining is achieved at three levels:

- A library-specific processing operation is performed by the server node on one data extent while the server node reads the next data extents,
- a processed data extent is asynchronously sent across the network to the client node while the next data extent is processed,
- a processed data extent is merged by the client node while the next processed data extent is asynchronously transmitted across the network from the server node to the client node.

Parallelization occurs at two levels:

- several data extents are simultaneously read from different disks; the number of disks in the server nodes can be increased to improve I/O throughput,

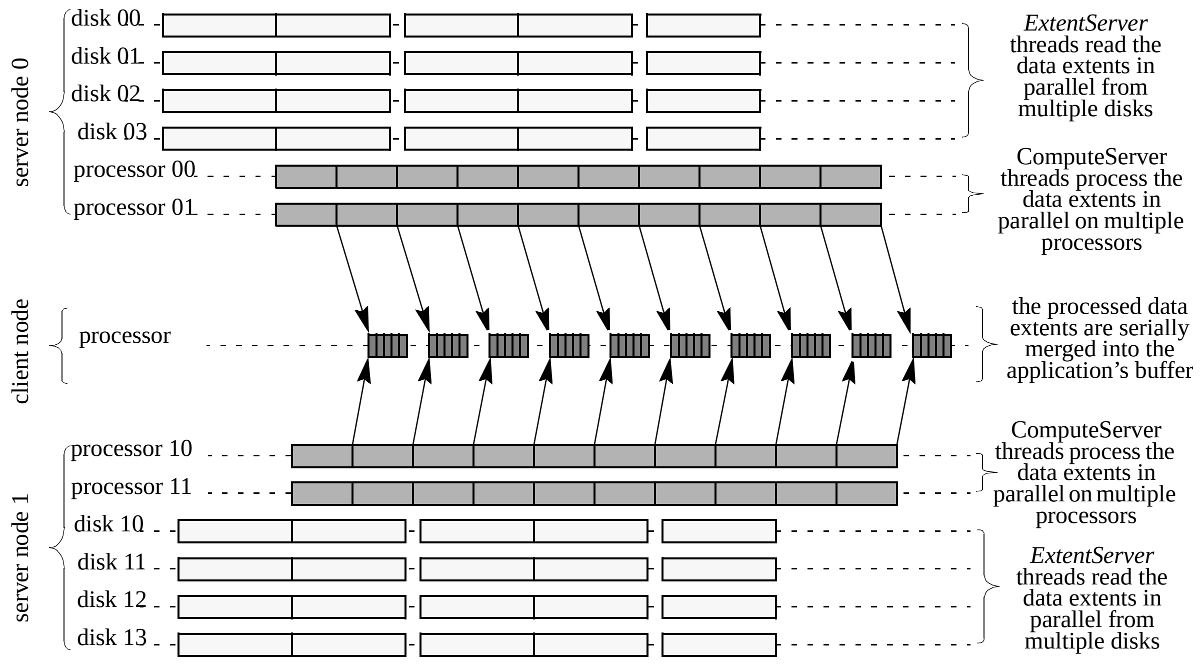

server node 0
disk 00
disk 01
disk 02
disk 03
processor 00
processor 01
ExtentServer threads read the data extents in parallel from multiple disks
ComputeServer threads process the data extents in parallel on multiple processors
client node
processor
the processed data extents are serially merged into the application's buffer
server node 1
processor 10
processor 11
disk 10
disk 11
disk 12
disk 13
ComputeServer threads process the data extents in parallel on multiple processors
ExtentServer threads read the data extents in parallel from multiple disks

Figure 2-6: With PS^2 disk accesses, computations, and communications are executed in a pipelined parallel manner

- application-specific or library-specific processing operations on data extents are carried out in parallel by several server node processors; the number of server node processors can be increased to improve processing performance.

Provided that there are enough disks hooked onto the server nodes to meet the I/O requirement of an application, i.e. when the execution time of the parallel I/O- and compute- intensive operation is limited by the processing power at the server nodes (Fig. 2-6), a pipelined parallel execution enables hiding slow disk accesses and high-latency network communications. Therefore, PS^2 enables application programmers create I/O and processing intensive applications on striped data sets that have potentially the same execution time as if the application's data sets would be stored in large main memories instead of being striped across multiple disks hooked on several server nodes.

2.4 Summary

This chapter presented the CAP computer-aided tool simplifying the creation of pipelined parallel applications. CAP enables application programmers to specify the parallel application schedule at a higher level of abstraction than commonly used parallel programming systems based on message passing. The parallel application is concisely described as a set of threads, operations located within the threads and flow of data (tokens) between operations. Application programmers create separately the serial program parts and express the parallel schedule of the program with CAP constructs. Due to the clean separation between parallel construct specification and the remaining sequential program parts, CAP programs are easier to debug and to maintain. In addition, thanks to the automatic compilation of parallel applications, application programers do not need to explicitly program the protocols to exchange data between parallel threads and to ensure their synchronization. CAP was used for creating the library of striped file components (PS^2).

The striped file components library (PS^2) comprises a fixed set of low-level striped file system components as well as an extensible parallel processing system. The reusable low-level striped file system components and their corresponding I/O threads enable access to files that are declustered across multiple disks. Application or library programmers can, thanks to the CAP formalism, easily and elegantly extend the functionalities of the PS^2 library by combining the predefined low-level striped file access components with the application-specific or library-specific processing operations in order to yield efficient pipelined parallel I/O- and compute-intensive CAP operations. In Chapter 3 we will describe how the PS^2 functionalities are extended in order to provide parallel continuous media support on top of striped files.

The CAP computer-aided parallelization tool has been conceived by Dr. Benoit Gennart. The striped file components library (PS^2) has been conceived by Dr. Vincent Messerli. In the context of this thesis, I have contributed to CAP by specifying features to be incorporated into CAP. The CAP tool and PS^2 library have been improved thanks to the present thesis. In particular, I suggested new CAP constructs for supporting isochrone server access requests. In addition, the continuous media applications developed within the present thesis were extremely useful for testing and verifying CAP’s functionality and correctness.

3 Parallel Continuous Media Support

3.1 Introduction

This chapter describes a new approach for developing parallel server for I/O and compute intensive continuous media applications running on multi-PC multi-disk architectures (Fig. 2-5). A customizable parallel stream server, based on the CAP computer-aided parallelization tool (section 2.2) and on the PS^2 striped file component library (section 2.3), offers a library of basic continuous media services (e.g. disk scheduling algorithm, admission control and resource reservation mechanism,...). These continuous media services are media independent and are necessary to guarantee the delivery, at a specified rate, of continuous media data from the multiple server disks to the client. Thanks to the CAP tool, these basic services can be combined with media-dependent stream operations in order to yield efficient pipelined parallel continuous media servers, e.g. the *4D beating heart slice stream server* described in Chapter 4. Although the parallel stream server library is addressed to develop parallel I/O and compute intensive continuous media server, the library is general enough for developping any continuous media server, e.g. a video on demand server.

3.2 Related work

Issues related to the file system support for storing real-time video and audio streams individually on magnetic disks are explored by Rangan and Vin [70] and by Andersson et al. [4]. Scheduling and admission control algorithms ensure that performance guarantees of the processor and storage resources for real-time streams are fulfilled in the presence of non-real-time, sporadic traffic [69]. A multimedia storage system that runs under the IRIX, Solaris and Windows operating systems is presented by Martin et al. [57].

Many publications [78, 87, 32, 73, 46, 55] present the fundamental issues arising in multimedia server design, such as the placement policy of the multimedia data, the admission control algorithms, the disk scheduling algorithms and the paradigms to schedule retrieval of media streams. Reddy and Wyllie propose a disk arm-scheduling approach based on the earliest-deadline-first (EDF) algorithm for multimedia data [72]. The same authors have characterized the disk-level trade-offs in a multimedia server [73]. In order to support the high bandwidth requirement of the continuous media objects (e.g. the continuous display of a video object [31]) several techniques have been presented: compression (e.g. MPEG [54]), striping [68, 76, 83], declustering [37, 38, 12] or a combination of these techniques [6]. In [84], Vin and Rangan present a quantitative study of designing a multiuser HDTV server and discuss efficient techniques [71] for storing multiple HDTV videos on magnetic disks and serving multiple subscriber requests simultaneously under the constraint of guaranteeing HDTV playback rates. Jadav and

Choudhary [46] propose high-performance computers as the best solution for media-on-demand servers according to their analysis of server requirements and implementation problems. In order to improve the capacity and reliability of single computer video servers, Lee introduces a framework for the design of parallel video server architectures and examines mainly the server striping policies and the video delivery protocols [55].

A significant amount of research was performed on hierarchical storage systems [11, 24, 80, 39, 29, 77] for video-on-demand servers in order to offer high-capacity and low-cost storage. In such systems, video files are stored at the tertiary storage level (i.e. robotic libraries based on tapes or optical disks) and transferred to the secondary level (i.e. magnetic disks) before being transferred to the clients.

Several recent publications review and analyze different approaches for designing storage architectures for multimedia applications [15, 28, 43] based on a single- or a multi-computer system connected to storage devices through a storage area network (SAN).

Previous research projects show that a central issue is the high I/O bandwidth required by continuous media servers. Our effort is focussed on parallel continuous media applications where both processing power requirements and I/O bandwidth requirements are important. In order to achieve an efficient resource utilization these applications have to ensure that computations, disk accesses and network transfers occur simultaneously. In addition, in such multimedia applications, data to be served is not necessarily stored contiguously on the disk.

3.3 Objectives and design of the continuous media support

Our goal is to provide application programmers a framework for continuous media support in order to guarantee the delivery, at a desired rate, of continuous media data from the server disks of a multi-PC multi-disk server architecture. By using the library of striped file components (PS^2), continuous media data is segmented into storage units, called extents, that are stored on the extent files[1] distributed across server disks. Each stream is defined by a structure comprising the attributes for controlling the data delivery and the parameters specific to the media.

In order to offer parallel continuous media support in multi-PC multi-disk server architectures (Fig. 2-5), we extend the functionalities of the PS^2 library (described Chapter 2) by adding the basic services required to guarantee the streaming of data from the server disks to the client without violating the stream’s real-time contraints. The new services will be written with the CAP computer-aided parallelization tool (described in Chapter 2). Basic services to support continuous media access are (1) an admission control algorithm, (2) a mechanism of resource reservation, (3) isochrone media delivery by the server and (4) a disk scheduling algorithm. These basic services for continuous media are independent of the media and may be used by application programmers to implement specific continuous media servers. The challenge

1. Extent files are the subfiles making up the striped file (see section 2.3).

resides in combining basic continuous media services and parallel file striping. We describe how to integrate basic continuous media services in the library of parallel file striping components. This integration yields the parallel stream server library. Chapter 4 describes a specific parallel continuous media server, the *4D Beating Heart Slice Stream Server*, implemented with the parallel stream server library.

The parallel stream server library is application independent. Continuous media server programmers are free of deciding (1) how continuous media data is splitted into extents, (2) how extents are distributed across the server disks and (3) how the data is streamed from the server nodes to the client: according to the *client-pull*, *server-push* models or a combination of them. By programming data streaming with the CAP preprocessor language and by making use of the parallel stream library, we can obtain a high degree of pipelining and parallelism yielding therefore a high resource utilization rate.

Fig. 3-1 shows the overall design of the integration of the continuous media support within the CAP/PS^2 framework.

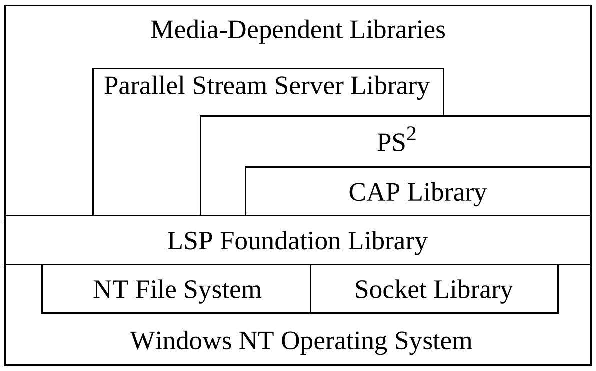

Media-Dependent Libraries
Parallel Stream Server Library
PS^2
CAP Library
LSP Foundation Library
NT File System
Socket Library
Windows NT Operating System

Figure 3-1: Integration of the continuous media support within the CAP/PS^2 framework

At the bottom, the Windows NT operating system is insulated from the above layers by the LSP Foundation library which provides basic data structures, memory pool allocation routines and OS independent interprocess communication primitives.

The CAP library (section 2.2) provides the necessary functions in order to implement the features of the CAP language, i.e. operation scheduling, thread and process management, synchronization and an interface for interprocess message-passing.

The PS^2 system (section 2.3) offers the necessary primitives to access striped files distributed over a cluster of networked computers each having several attached disks. The PS^2 system exports computation threads into which custom operations can be plugged, thus ensuring a very tight combination of disk access and computation operations [60].

The parallel stream server library provides general services for supporting continuous media access. The offered services are (1) an access control algorithm, (2) a resource reservation mechanism, (3) a disk scheduling policy based on the earliest deadline first algorithm (EDF) and on incremental disk track scanning (SCAN) and (4) a support for isochronous server mode behavior.

Finally, media-dependent libraries sit on top of these layers. Media-dependent libraries complete the parallel stream server library by defining the strategy for partitioning the stream data into extents, the distribution of the extents across the server disks and by implementing the parallel operation for streaming the media data from the server disks to the client. An example of a media-dependent library is the 4D beating heart slice stream server library described in Chapter 4.

3.4 The parallel stream server library

The parallel stream server library offers a set of threads with appropriate operations for providing continuous media support for streaming data from the server disks to the client. Application programmers are free to incorporate specific operations to the existing threads and to combine them with the predefined operations to implement their parallel continuous media servers.

The parallel stream server is composed by the following threads:

- The *InterfaceServer* thread is part of the PS^2 library offering operations to coordinate the parallel file operations (section 2.3). For supporting continuous media, new functionalities have been added to this thread. The *InterfaceServer* thread is responsible for coordinating the parallel operations for opening and closing parallel streams. It also applies the admission control algorithm to evaluate the availability of server resources and to determine whether a new request stream can be accepted. The *InterfaceServer* thread reserves the necessary resources for serving each accepted stream.

- The *ExtentFileServer* threads are part of the PS^2 library offering operations to open, close and delete extent files (section 2.3). For supporting continuous media, these threads are responsible of managing the data structure representing the accepted stream object. When opening a parallel stream, a stream object is created on each server node involved in the stream so that subsequent operations may refer to that stream. When closing the parallel stream, the stream object from each concerned server node is destroyed.

- The *ExtentServer* threads are part of the PS^2 library offering operations to read, write and delete extents on the server disks (section 2.3). For supporting continuous media accesses, a new operation to read extents according to their deadlines has been added. Before reading asynchronously the extents, this new extent read operation schedules the extent requests according to the SCAN/EDF algorithm [72] in order to satisfy a certain deadline.

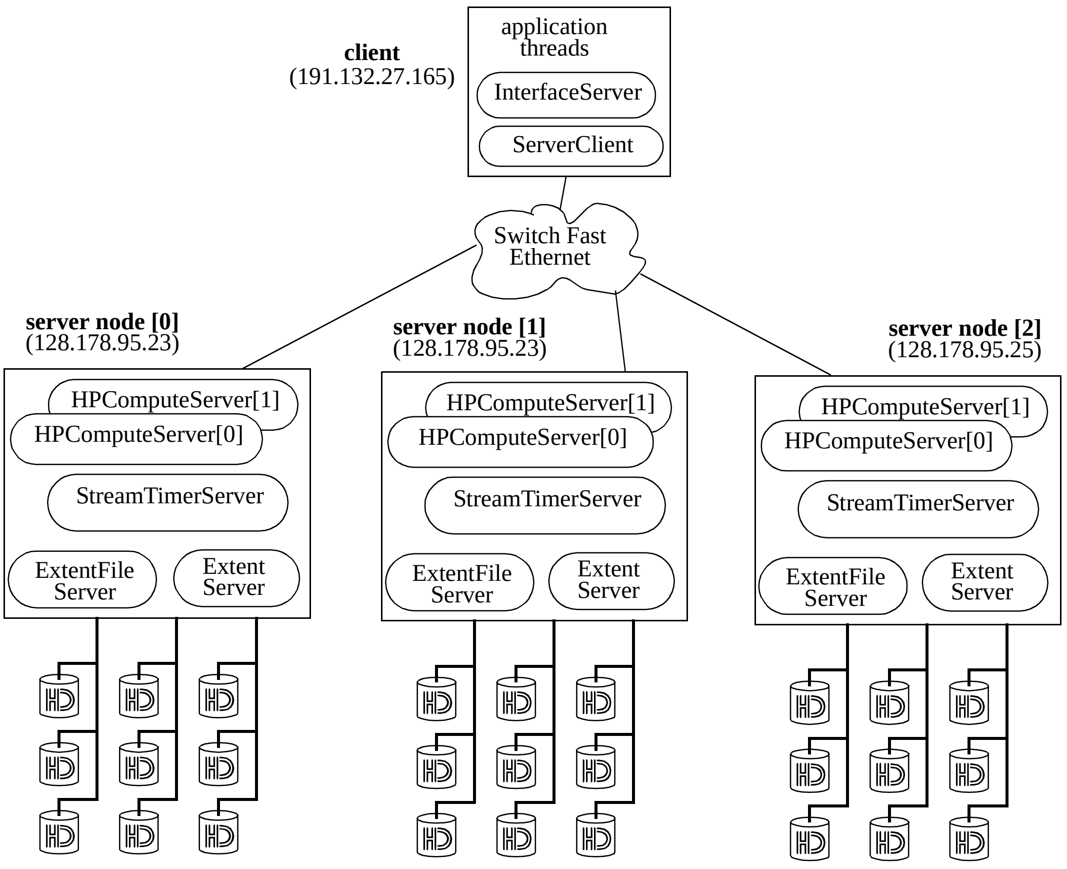


Figure 3-2: Server threads mapped onto a multi-PC multi-disk server architecture

- The *HPComputeServer* (HP stands for High Priority) threads are outside the PS^2 library. They have the same function as the *ComputeServer* threads offered by the PS^2 library, but the *HPComputeServer* thread have a higher priority. They are responsible for handling server side high-priority application-specific processing operations[1]. They do not offer any predefined operations. In order to stream data from the server disks to the client, continuous media server programmers add processing operations to *HPComputeServer* threads and combine these added functionalities with the predefined operations offered by the server's threads.
- The *StreamTimerServer* threads are outside the PS^2 library. They are in charge of the isochronicity behavior of each server node. By using the system timers, *StreamTimerServer* threads generate at regular time intervals the successive media access request to generate the stream. Examples of media requests are slice extraction requests in a slice stream or audio sequence requests in an audio stream.

1. Continuous media applications may take advantage of the *HPComputeServer* threads to ensure that they have a higher priority compared to non-continuous applications.

- The *ServerClient* threads are outside the PS^2 library. They are responsible for handling application-specific processing operations, e.g. for resequencing and merging data from the server nodes into a coherent stream. Similar to the *HPComputeServer* threads, they do not offer any predefined operations.

Fig. 3-2 shows an example on how the server threads are mapped onto a multi-PC multi-disk server comprising 3 server nodes. On the client node, the *application threads* and one *Server-Client* thread and the *InterfaceServer* thread are running. On each server node, an *ExtentFileServer* thread, an *ExtentServer* thread, a *StreamTimerServer* thread, and several *HPComputeServer* threads are running. The *application threads* communicate with the *Inter-faceServer* thread for opening and closing parallel streams and for reserving and releasing the appropriate resources. *Application threads* communicate directly with the *StreamTimerServer* threads to start streaming data from the server disks. The *StreamTimerServer*, *ExtentServer*, *HPComputeServer* and *ServerClient* threads collaborate to stream data from the server disks to the client at a controlled rate.

Although the thread-to-process mapping is under the control of application programmers, some restrictions are imposed to the server threads:

- There must be always a single *InterfaceServer* thread.
- There must be at least a single *ExtentFileServer* thread and a single *ExtentServer* thread per server node. Thanks to the asynchronous extent access operations, a same *ExtentServer* thread can serve in parallel several disks.
- There must be always a single *StreamTimerServer* thread per server node.
- For load-balancing purposes there must be the same number of *HPComputeServer* threads on each server node.

Fig. 3-3 shows a configuration file for the CAP runtime system specifying the mapping of threads to Windows NT processes running on the server architecture shown in Fig. 3-2. Process

```
configuration {
processes:
A (“user”);
B ( “128.178.95.22”, “server.exe”);
C ( “128.178.95.23”, “server.exe”);
D ( “128.178.95.24”, “server.exe”);
threads:
“StreamServer.ServerClient” (A);
“Ps2Server.InterfaceServer” (A);
“StreamServer.ServerNode[0].HPComputeServer[0]” (B);
“StreamServer.ServerNode[0].HPComputeServer[1]” (B);
“StreamServer.ServerNode[0].StreamTimerServer” (B);
“Ps2Server.ServerNode[0].ExtentFileServer” (B);
“Ps2Server.ServerNode[0].ExtentServer” (B);
“StreamServer.ServerNode[1].HPComputeServer[0]” (C);
“StreamServer.ServerNode[1].HPComputeServer[1]” (C);
“StreamServer.ServerNode[1].StreamTimerServer” (C);
“Ps2Server.ServerNode[1].ExtentFileServer” (C);
“Ps2Server.ServerNode[1].ExtentServer” (C);
“StreamServer.ServerNode[2].HPComputeServer[0]” (D);
“StreamServer.ServerNode[2].HPComputeServer[1]” (D);
“StreamServer.ServerNode[2].StreamTimerServer” (D);
“Ps2Server.ServerNode[2].ExtentFileServer” (D);
“Ps2Server.ServerNode[2].ExtentServer” (D);
};
```

Figure 3-3: CAP configuration file mapping the server threads onto different server nodes

A runs the application's front end on the user machine (IP address: 191.132.27.165). Processes B, C and D execute the server application (server.exe) on the server nodes whose IP addresses are 128.178.95.23, 128.178.95.24 and 128.178.95.25 respectively. The parallel stream server comprises 17 threads. The *InterfaceServer* thread and one *ServerClient* thread are mapped on the user's node. One *ServerNode* CAP high-level process is mapped per server node. Each *ServerNode* high-level CAP process comprises an *ExtentFileServer* thread, an *ExtentServer* thread, a *StreamTimerServer* thread and two *HPComputeServer* threads. Several *HPComputeServer* threads are mapped per server node so as to benefit from dual processor PC's (e.g. Bi-Pentium).

3.5 Stream internal representation

In the parallel stream server library, a stream is represented by two types of objects: the *striped stream object* and the *stream object* (Fig. 3-4). When opening a striped stream, a *striped stream object* is created by the *InterfaceServer* thread, and a *stream object* is created on each involved server node. The *stream objects* are created by the *ExtentFileServer* threads. When closing the striped stream, all the created objects are destroyed.

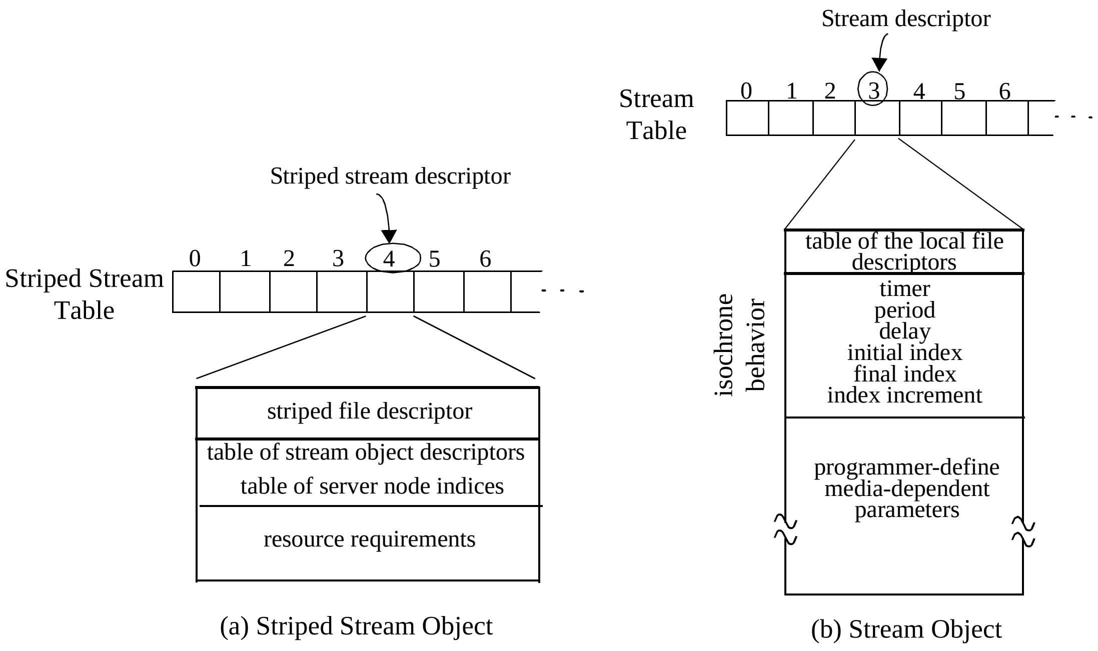


Figure 3-4: Stream internal representation

On each server node, the *ExtentFileServer* thread maintains a table of *stream objects* (Fig. 3-4b). The descriptor of a *stream object* is simply the index in the table where that *stream object* is stored. The *stream object* comprises the necessary parameters to implement the media streaming operation. These parameters are

- the descriptors of the local files where the stream data resides;

- a set of parameters describing the isochrone behavior of the server node, i.e. the regular time intervals at which media access requests are made in order to create the stream;
- a set of media-dependent parameters. These parameters describe how the stream data is striped across the server disks. These parameters are used to compute the extent indices corresponding to a given media access request. When creating a stream, for each computed extent index, an extent request is launched.

In order to keep track of the open striped streams and their corresponding resources consummation in the parallel server, the *InterfaceServer* thread maintains a striped stream object table. Each *striped stream object* is referred by its descriptor, i.e. its index in the table. The *striped stream object* (Fig. 3-4a) contains:

- the name of the striped file and the file open flags used when opening the striped stream;
- the striped file descriptor in order to close the striped file when closing the striped stream;
- the table with the indices of the concerned server nodes, and the table of the corresponding stream object descriptors;
- the resource requirements.

3.6 Predefined parallel operations

In addition to the previous threads and their corresponding operations, the parallel stream server library offers two predefined parallel operations. A parallel operation for opening a stream striped across the server disks (*OpenStripedStream* operation) and a parallel operation to close an open striped stream (*CloseStripedStream* operation).

Fig. 3-5 shows the operation schedule of the *OpenStripedStream* parallel operation[1]. For the sake of simplicity, error handling is not shown. The input of the parallel *OpenStripedStream* operations is the path name of the striped file where the stream data resides and the opening mode flags. First, the target striped file is open by calling the *OpenStripedFile* parallel operation predefined by the PS^2 library. The output of the *OpenStripedFile* operation is directed to the *InterfaceServer* thread who identifies the server nodes involved in the stream, i.e. the server nodes whose disks contain stream data. A request for creating a stream object is sent to the *ExtentFileServer* threads located on each server node contributing to the stream. Each *ExtentFileServer* thread creates a stream object and sends its descriptor to the *InterfaceServer* thread who creates a striped stream object and initializes it with the received stream descriptors. The output of the *OpenStripedStream* operation comprises the striped stream descriptor, the indices

1. In the graphical specification of the operations, single line graphical objects specify sequential operations and double line graphical objects specifies parallel operations, i.e. a high-level operation comprising several operations carried out in parallel.

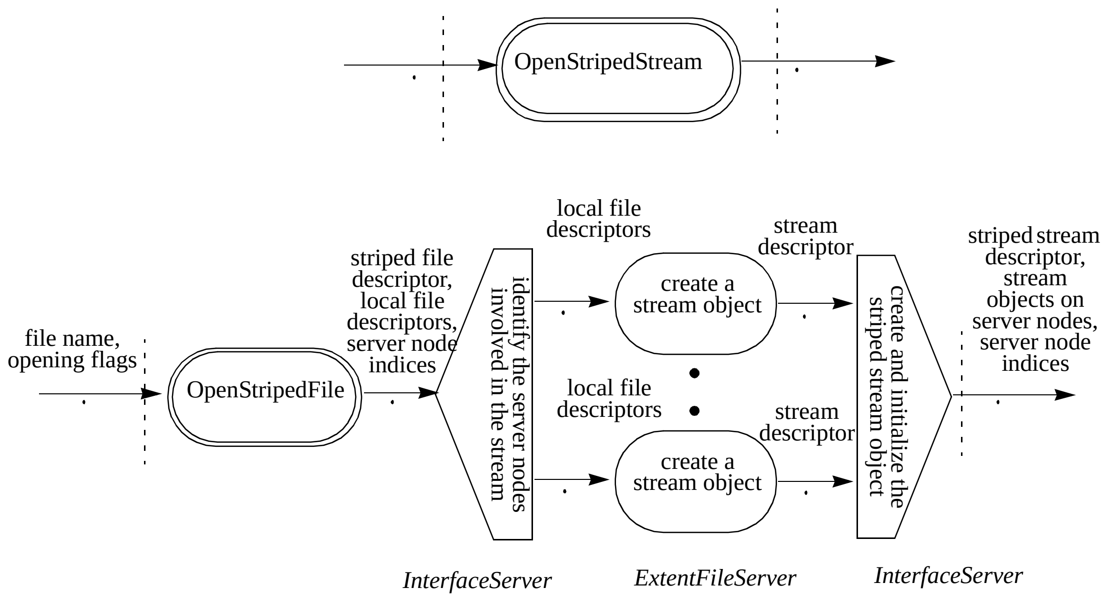

OpenStripedStream
file name, opening flags
OpenStripedFile
striped file descriptor, local file descriptors, server node indices
identify the server nodes involved in the stream
local file descriptors
create a stream object
local file descriptors
create a stream object
stream descriptor
stream descriptor
create and initialize the striped stream object
striped stream descriptor, stream objects on server nodes, server node indices
InterfaceServer
ExtentFileServer
InterfaceServer

Figure 3-5: Graphical specification of the *OpenStripedStream* parallel operation

of the server nodes involved in the stream and the descriptors of the stream objects created on each server node.

The *CloseStripedStream* parallel operation (Fig. 3-6) is designed according to a similar pattern. The input of the parallel *CloseStripedStream* operation is the striped stream descriptor of a previous *OpenStripedStream* operation. The striped stream descriptor is directed to the *InterfaceServer* thread who accesses to the corresponding striped stream object in order to identify the concerned server nodes. A request containing the adequate stream descriptor is sent to each concerned *ExtentFileServer* thread to destroy the stream objects associated to these stream descriptors. After destruction of the stream objects, the *InterfaceServer* thread destroys the cor-

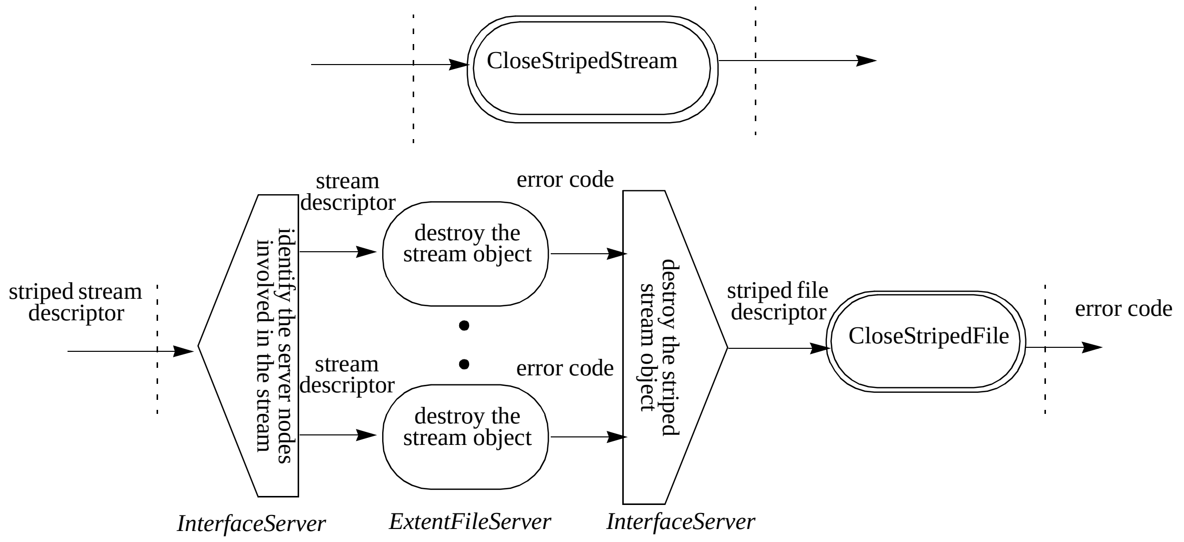


Figure 3-6: Graphical specification of the *CloseStripedStream* parallel operation

responding striped stream object. Finally, the *CloseStripedFile* parallel operation predefined by the PS^2 library is called to close the striped file. The output of the parallel *CloseStripedStream* operation is merely an error code. Once a stream is closed, the corresponding descriptors become invalid and can be reallocated to another stream.

3.7 Admission control algorithm

The admission control ensures that a new stream request does not cause the violation of the real-time requirements of streams already being serviced [4]. For each accepted stream, the admission control algorithm reserves the resources to serve them. The server comprises the following resources (1) parallel disk I/O bandwidth, (2) parallel server processing power, (3) network bandwidth and (4) processing power at the client node. The current implemented admission control algorithm reserves the bandwidth of each server disk[1]. The disk bandwidth is reserved up to a given bound (e.g. 2 MB/s per disk). The estimation of a pessimistic bound ensures that deadlines are met, but implies a poor utilization of the resource. An optimist bound implies a high resource utilization but may overload the resource. The parallel stream server applies a *deterministic* admission control algorithm [70], i.e. it estimates the reservation bound from worst-case experimental results. As opposed to the *deterministic* admission control algorithms, there are *statistical*[2] and *measurement-based*[3] admission control algorithms.

In the parallel stream server, the *InterfaceServer* thread implements the admission control algorithm and the resource reservation mechanism. The *InterfaceServer* thread maintains a table of all open parallel streams, the reserved server load (the reserved disk bandwidth for each disk in the parallel server) and a reservation table whose entries specify the reserved resources for each of the streams being served. When reserving resources, the *InterfaceServer* thread evaluates the server resources in order to accept or refuse the new stream. If the stream is accepted, the reserved server load is updated and an entry in the reservation table is created for the new stream. When the stream retrieval is terminated, the *InterfaceServer* thread deletes the reservation table entry specified by the parallel stream descriptor and updates the reserved server load.

3.8 Isochrone behavior

In order to delivery the stream data respecting both the stream real-time contraints and the resources allocated to the stream, the parallel stream server relies on the isochrone behavior of

1. We consider continuous media applications which are I/O bound. Since the required processing power is proportional to the required I/O bandwidth, it is sufficient to allocate and reserve I/O bandwidth.
2. Statistical admission control algorithms use probability distribution of the rate and disk access times of media streams to guarantee that deadlines will be met with a certain probability. Such algorithms achieve a much higher utilization than *deterministic* algorithms, and are used when clients can tolerate infrequent losses [85].
3. Measurements-based admission control algorithms use past measurements of rate and disk access times of media streams as an indicator of future resource requirements. They achieve the highest disk utilization at the expense of providing the weakest guarantees [86].

each server node, i.e. on the ability of each server node to deliver at constant time intervals the corresponding amounts of stream data. In the parallel stream server, the isochronicity of each server node is performed by the high-priority *StreamTimerServer* thread which by calling the *capDoNotCallSuccessor* CAP function [35] during the execution of the *parallel-while-suspend* CAP construct [35] allows the suspension of the token flow during a specified period of time. Later in the section, we describe how to use these CAP constructs in order to implement the isochrone behavior of the *StreamTimerServer* threads.

The isochrone behavior of the *StreamTimerServer* thread is defined by a set of parameters contained in the stream object (Fig. 3-4b). The *timer* object (i.e. a system timer) is responsible for creating timer events at regular time intervals. The *period* parameter specifies the interval of time between two timer events. The *delay* parameter indicates the time to wait before starting the timer. Finally, the stream duration is defined by the *initial_index*, *final_index* and *index_inc* parameters. Application programers determine the appropriate values of the isochronicity parameters for each *StreamTimerServer* thread according to their own strategy for striping the stream data across the server nodes and according to the stream rate. Application programmers should implement an operation to initialize these parameters (and possibly other specific-media parameters) on each concerned server node. This “initialization stream” operation is to be called after reserving the required resources for the stream and before starting to retrieve the stream.

The isochrone behavior of the *StreamTimerServer* threads is described as follows. Once the stream is initialized (by calling the initialization stream operation) and the *StreamTimerServer* thread receives from the client the request to start the stream retrieval, the *StreamTimerServer* thread waits for a certain *delay* time before starting the corresponding *timer*. When the *delay* time is elapsed, the first timer event is signaled. Successive timer events are periodically signaled by the *timer* according to the *period* time. At each new timer event the *initial_index* value is incremented by *index_inc* units. When the *initial_index* value reaches the *final_index* value, the *timer* is reset and stopped. Application programmers may at each timer event generate a media access request or a set of media access requests when implementing their own stream retrieval operations.

In addition to the stream rate, the value of the *delay* and *period* parameters describing the isochronicity behavior of each node may significantly vary depending on how the stream data is segmented into extents and distributed across the server node disks. *Delay* gives the time offset between request deadlines on different server nodes. The *period* gives the request intervals on a given server node. Fig. 3-7 shows the value of the *delay* and *period* isochronicity parameters for each server node when requesting a 3 frames/s stream striped across 3 server nodes according to the time striping policy[1][55]. In Fig. 3-7a, a stripe unit or extent contains one single entire

1. The time striping policy consists of partitioning a stream in units of time (or frames) across multiple servers. A stripe unit contains F frames, if $F < 1$, then it is called subframe striping, otherwise it is called frame striping. In contrast to the time striping policy, the space striping policy divides a stream into stripe units having a fixed size in bytes.

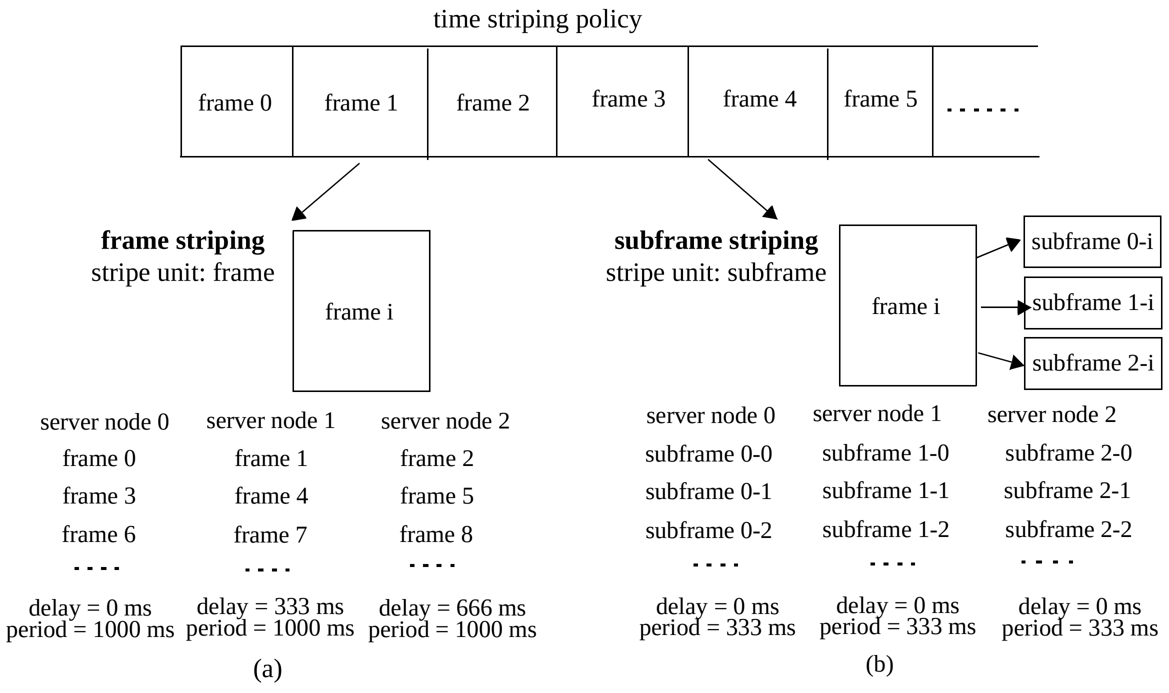


Figure 3-7: Value of the *delay* and *period* isochronicity parameters when requesting a 3 frames/s stream striped across 3 server nodes according (a) to the frame striping and (b) to the subframe striping policies

frame (frame striping). The stripe units are distributed across the server nodes in a round-robin fashion. In Fig. 3-7b, a stripe unit contains a frame part, i.e. 1/3 of a frame (subframe striping).

Fig. 3-8 shows the CAP source code to implement the isochrone behavior. By using the *parallel-while-suspend* CAP construct (line 28), the *StreamTimerServer* thread calls the *GenerateFrameRequests* routine (line 11) to generate each time it is called a *FrameRequestT* request (line 15). The created *FrameRequestT* request is passed as parameter to the *Suspend-Function* routine (line 1) of the *parallel-while-suspend* CAP construct. There, after setting the *timer* of the stream object to be signaled in *period* time units, i.e. in the time interval between two successive frames (line 4), we call the *capDoNotCallSuccessor* CAP function (line 5) which suspends the current *FrameRequestT* request. Since this request is needed by the *GenerateFrameRequests* routine of the *parallel-while-suspend* CAP construct, the generation of the next frame request is also suspended. When the *period* time ends, the timer becomes signaled and its *TimerCallbackRoutine* callback routine is called (line 7). In the timer callback routine, the previously suspended *FrameRequestT* request is resumed by calling the *capCallSuccessor* CAP function (line 9). Resuming the current *FrameRequestT* request allows the *Generate-FrameRequests* routine to generate the next frame request. The resumed *FrameRequestT* request is directed to the corresponding server threads in order to extract the frame (line 31). Thanks to the suspension of the frame request flow during the *period* time, consecutive frames are extracted at appropriate time intervals. The number of generated frame requests depends on the *Initial_Index, Final_Index* and *Index_Inc* parameters of the stream object (lines 16 and 17).

```
void SuspendFunction (FrameRequestT* NextP)
{
   StreamObjectT * streamObjectP = StreamObjectTable[NextP->StreamDescriptor];
   streamObjectP->Timer.Set (streamObjectP->Period, TimerCallbackRoutine (NextP);
   capDoNotCallSuccessor (NextP);
}
void TimerCallbackRoutine (FrameRequestT* NextP)
{
  capCallSuccessor (NextP);
}
bool GenerateFrameRequests (StreamRequestT* InputP, FrameRequestT* PreviousP,
                            FrameRequestT* &NextP)
{
  StreamObjectT * streamObjectP = StreamObjectTable[InputP->StreamDescriptor];
  NextP = new FrameRequestT (InputP, PreviousP);
  streamObjectP->Initial_Index += streamObjectP->Index_Inc;
  return ( streamObjectP->Initial_Index == streamObjectP->Final_Index );
}

void MergeFunction (ErrorT* IntoP, FrameT* FromP)
{
  // .... C++ sequential code
}
operation StreamTimerServerT::ExtractStream
in StreamRequestT InputP
out ErrorT OutputP
{
   parallel while (GenerateFrameRequests suspend SuspendFunction (),
                  MergeFrame, ServerClient, ErrorT Result)
   (
      // ... CAP code for extracting a frame
   )
}
```

Figure 3-8: Implementation of the isochrone behavior using CAP

3.9 Disk scheduling algorithm

In order to reduce seek times, to achieve a high disk throughput and to guarantee the real-time requirements of the media streams being serviced, the *ExtentServer* thread schedules the extent requests as follows. An extent request consists of the stream descriptor, the server node index, the local file descriptor, the local extent index and the request deadline. A zero deadline value specifies an access to a non-continuous media extent. The *ExtentServer* thread maintains two extent request lists per disk hooked on its server node, a list containing extent requests for non-continuous media and a list containing extent requests for continuous media. The continuous media request list has always a higher priority than the non-continuous media request list, i.e. continuous media extent requests are always served first. Non-continuous media extent requests are served according to a first come first served policy. Continuous media extent requests are served according to the earliest deadline first (EDF) strategy [72]. If several extent requests have the same deadline, they are served as follows:

- for extent requests belonging to different streams (e.g. streams of different media, or different streams of the same media), the extent request containing more time slices is served first,

- for extent requests belonging to the same stream, extent requests are served in the order of their extent indices. This enables to move the disk head in one direction, similar to the SCAN disk scheduling algorithm, since extents with consecutive indices are generally stored in consecutive blocks.

3.10 Synthesizing a video on demand server

Although the parallel stream library has been developed to support parallel I/O and compute intensive continuous media server applications (e.g. the *Beating Heart Slice Server* described in the next chapter), it is able to support any continuous media application, e.g. a video-on-demand server (VoD). This section describes the main steps when developing a VoD server using the parallel stream server library and the CAP tool[1]. The presented example demonstrates how thanks to the CAP tool application programmers develop easily and efficiently continuous media applications combining application-specific operations with the reusable operations offered by the parallel stream library.

MPEG video declustering strategy

First, we define the strategy for declustering an MPEG video file into a set of subfiles, each one stored in a different disk. A simple declustering strategy consists of segmenting the MPEG video into GOPs (group of pictures). GOPs are stored into individual extents that are distributed across the server disks in a round-robin fashion (Fig. 3-8). In order to serve a video file at 30 frame/s, assuming that each GOP contains 10 video frames[2] and that each video file is striped over 3 disks (i.e. a striped video file consists of 3 extent files stored in different disks), the server reads in parallel 3 GOPs per second.

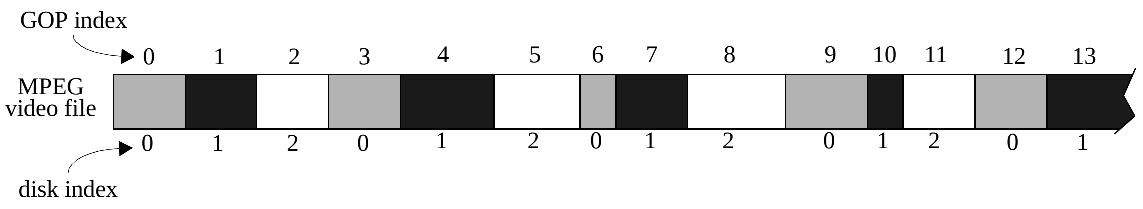


Figure 3-8: Strategy for declustering an MPEG video across 3 disks

Fig. 3-9 shows the graphical representation of the operation for distributing the GOPs (extents) across the server disks[3]. The input of this operation contains the file name of the video to be

1. Since the conversion of graphical CAP schedules into CAP source code is straightforward (section 2.2), we only show the graphical representation of the CAP schedules.
2. Although the number of frames contained into a GOP is fixed (i.e. in our VoD server, 10 frames/GOP), the GOP size in bytes can vary. Remember that the striped file component library enable to store extents of different sizes in the same extent file.
3. We assume that the video striped file has been created and is opened, i.e. an empty striped file exists.

striped. We assume that the video file is initially stored in the server node where the *ServerClient* thread resides[1]. The *ServerClient* thread reads the GOPs comprising the video file. Each GOP is packed into a write extent request that is sent to the appropriate *ExtentServer* to write it in the extent file.

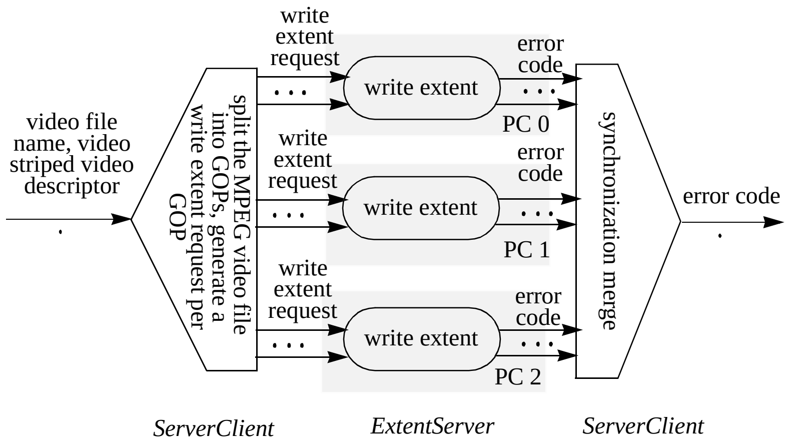

video file name, video striped video descriptor
split the MPEG video file into GOPs, generate a write extent request per GOP
write extent request
write extent
error code
PC 0
PC 1
PC 2
synchronization merge
error code
ServerClient
ExtentServer
ServerClient

Figure 3-9: Graphical representation of the GOP distribution operation

In addition, we store meta-information in extent 0 of each extent file. In our VoD server, the meta-information contains the number of GOPs stored in the extent file, the number of frames per GOP and the average GOP size in bytes. Our VoD server uses the meta-information to reserve the I/O bandwidth required of each extent file.

Resource reservation for reading an MPEG video stream

Before starting to stream video data, the *ServerClient* thread needs to open the striped video stream by calling the *OpenStripedStream* parallel operation described in section 3.6. Once the video stream is opened, the *ServerClient* thread reads the meta-information by asking the concerned *ExtentServer* threads to perform the *ReadExtent* operation. From the video meta-information, the *ServerClient* can derive the resource requirements needed for extracting the striped video stream. For instance, knowing that the video is striped across 3 disks (e.g. disk0, disk1 and disk2), that the mean GOP size is 89 KB, 105 KB and 98 KB for disk0, disk1 and disk2 respectively, that each GOP contains 10 frames and that the video rate is of 30 frame/s, the required disk throughput from disk0, disk1 and disk2 are respectively 89 KB/s, 105 KB/s and 98 KB/s. The disk bandwidth required is packed into a resource reservation request and sent to the *InterfaceServer* thread which applies the control admission algorithm described in section 3.7. If resources are available, the *InterfaceServer* thread accepts the video stream and reserves

1. The *ServerClient* thread is the proxy of the client on the server; it serves as an interface between the real client located on the Web and the server.

the resources. Otherwise, an error message indicating the lack of disk bandwidth resources is sent back (Fig. 3-10).

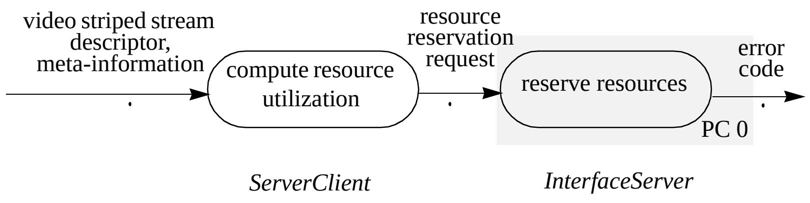

video striped stream descriptor, meta-information
compute resource utilization
resource reservation request
reserve resources
error code
PC 0
ServerClient
InterfaceServer

Figure 3-10: Resource reservation for a MPEG video

MPEG video stream initialization

Once a video stream is accepted, the video stream has to be initialized before starting to stream video data into the client. In the stream initialization operation, the stream object's isochrone parameters (sections 3.5 and 3.8) in the concerned server nodes are initialized according to the video rate and to the striping strategy. For the video on demand server, all server nodes are initialized in the same way[1]. *Delay* are *period* parameters are set to 0 and 1000 ms respectively. *Initial_index* and *final_index* parameters are set to the extent index of the first GOP and to the extent index of the last GOP respectively. *Index_inc* parameter is set to 1. According to these values, the *StreamTimerServer* threads generate (*final_index* - *initial_index* + 1) periodic requests. Fig. 3-11 shows the graphical representation of the video stream initialization operation. The operation input is an array of initialization values. An element of this array contains the initialization values for a given server PC. The *ServerClient* thread splits the array into individual initialization requests that are sent to the concerned server PCs. A *HPComputeServer* thread located in each server PC initializes the stream parameters with the initialization request's values.

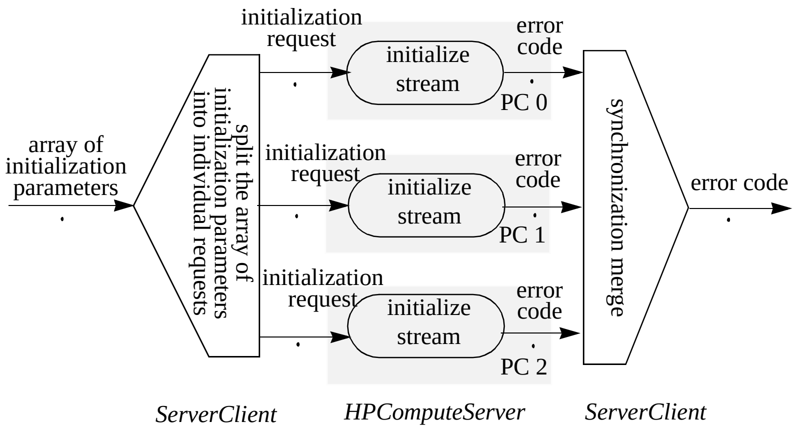


Figure 3-11: Graphical representation of the MPEG video stream initialization operation

1. In our VoD server, we only initialize the stream object's isochrone parameters. More complex servers can require to initialize additional media-specific parameters (i.e. the stream object's media-dependent parameters, section 3.5). For instance, in the case where a periodic request involves several extent accesses, a server may have to compute which extents need to be read in each periodic request

MPEG video stream extraction

Once accepted and initialized, the video stream can be continuously read. Fig. 3-12 shows the graphical representation of the video stream extraction operation[1].

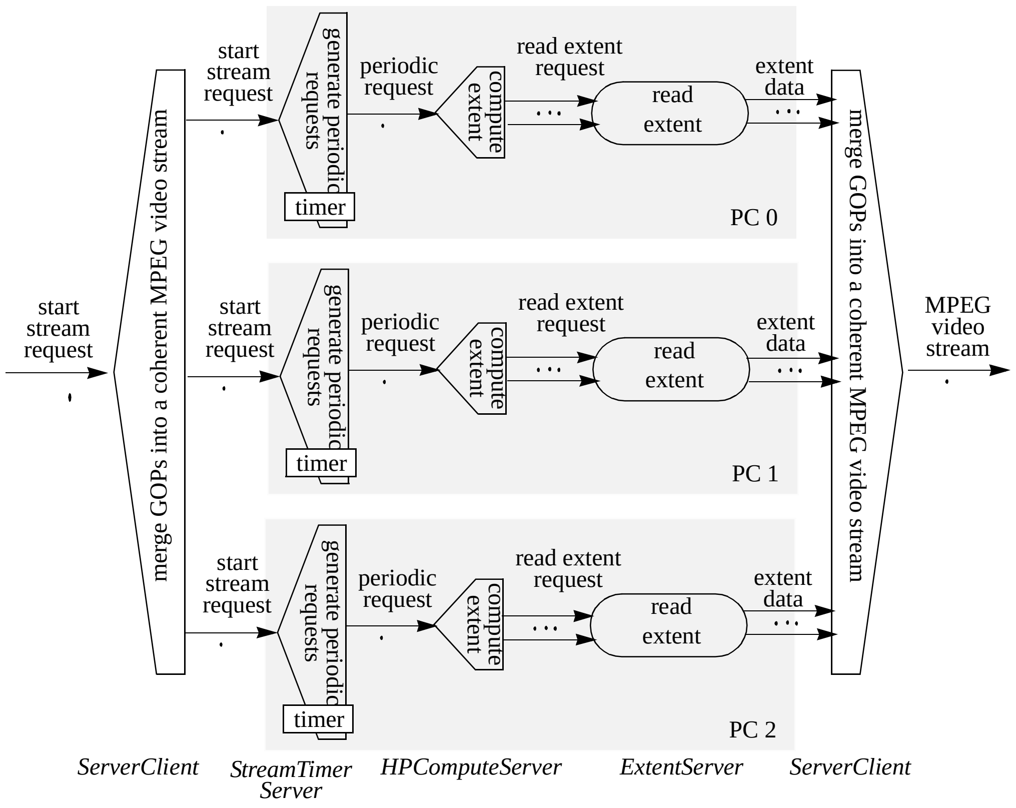


Figure 3-12: Graphical representation of the video stream extraction operation

First, the *ServerClient* thread sends a start stream request to all the *StreamTimerServer* threads located on the server PCs contributing to the stream. The start stream request contains a stream object descriptor that will be used by the different threads to access to the stream object's parameters. Using the system timers and the previously initialized isochrone parameters, each *StreamTimerServer* thread generates a periodic request every 1000 ms. Each periodic request contains the stream descriptor, a deadline and an index indicating the periodic request position within the stream. Each periodic request is directed to a *HPComputeServer* thread running in the same server PC. For each periodic request, the *HPComputeServer* thread creates a timed extent reading request. The extent index to be read is equal to the periodic request's index + 1[2]. The reading requests are sent to the *ExtentServer* threads who apply the disk access request scheduling algorithm (section 3.9) before launching extent reading requests to the disk. Once

1. Our VoD server sends the striped video stream to the client according to the server-push delivery model [55]. In Chapter 4, data delivery models are discussed more in detail.
2. Recall that the extent 0 of each extent file contains the meta-information.

the extents are read, they are sent back to the *ServerClient* thread which resequences and merges the GOPs[1] into a coherent MPEG video stream. After buffering some GOPs, an MPEG player can start to play the video stream while the next GOPs are being read and transferred.

3.11 Summary

This chapter describes the basic continuous media services that we developed in order to deliver stream data from multiple server disks to the client while guaranteeing the stream's real-time delivery contraints. The continuous media services comprise (1) a deterministic admission control algorithm, (2) the isochrone behavior of the server nodes supporting both the frame striping and the subframe striping policies and (3) a disk scheduling algorithm based on the SCAN-EDF policy. The presented example of a video-on-demand server demonstrates how thanks to the CAP tool, application programmers develop easily and efficiently parallel continuous media applications combining application-specific operations with the reusable operations offered by the parallel stream library. As a further example of an application built on top of the parallel stream server library, we present the *Beating Heart Slice Stream Server* in Chapter 4.

1. An extent contains a single GOP.

4 The 4D Beating Heart Slice Stream Server

4.1 Introduction

This chapter describes the design and implementation of a parallel continuous media server application requiring intensive I/O access and processing operations: the *4D beating heart slice stream server* [81]. Such a server is based on the parallel stream server library described in Chapter 3. A 4D tomographic beating heart incorporates as many 3D tomographic volumic images as time slices. In our experimental server, we store a beating heart made of 320 volumic images, corresponding to time slices of 1/16 second over a time interval of 20 seconds.

This server receives from clients slice stream access requests. Each slice stream request specifies the orientation of the slices within the 3D time-varying tomographic image and the access rate. After admission control (to allocate the required resources), the server continuously extracts slices according to the specified orientation and rate and transmits them to the client for visualization purposes.

To serve several continuous slice viewing requests simultaneously, the server needs both significant processing power and high disk throughput. To benefit from low-cost commodity components, the server architecture we consider consists of a cluster of PCs interconnected by a Fast Ethernet switch. Each PC is connected to several SCSI-2 disks (up to 12 disks). Scalability issues and performances for a server configuration of 3 PCs and 24 disks are discussed.

4.2 The Parallel 3D Tomographic Slice Server

Before explaining how the 4D beating heart is stored and accessed, let us briefly present the principles underlying the storage and access of slices within a 3D tomographic volume declustered over several disks and PCs.

For enabling parallel storage and access, the volumic data set is segmented into 3D volumic extents of small size for example 32x32x17 voxels, i.e. 51 KBytes, distributed over a number of disks. Concerning the size of the volumic extents, a trade off between data locality when applying imaging operations and the effective sustained I/O throughput needs to be found. As far as data locality is concerned, a volumic extent should be as small as possible so that only the most useful data is read from disk. On the other hand, in order to maximize the effective sustained I/O throughput, an extent should be as large as possible so as to reduce the time lost in head displacement, i.e. ensure that the disk access latency is small compared to the disk data transfer time. Previous experience [33] shows that a good extent size for a wide variety of parallel imaging applications can be achieved when the disk access latency time is similar to the disk transfer time.

The distribution of volumic extents across the server disks is made so as to ensure that direct volumic extent neighbors reside on different disks hooked on different server PCs. We achieve such a distribution by storing successive extents on successive disks located in different PCs and by introducing an offset between two successive rows and between two successive planes of volumic extents [58]. This ensures that for nearly all extracted slices, disk and server PC accesses are close to uniformly distributed.

Visualization of 3D medical images by slicing, i.e. by intersecting a 3D tomographic image with a plane having any desired position and orientation is a tool of choice in diagnosis and treatment. In order to extract a slice from the 3D image, the volumic extents intersecting the slice are read from the disks and the slice parts contained in these volumic extents are extracted and resampled (Fig.4-1).

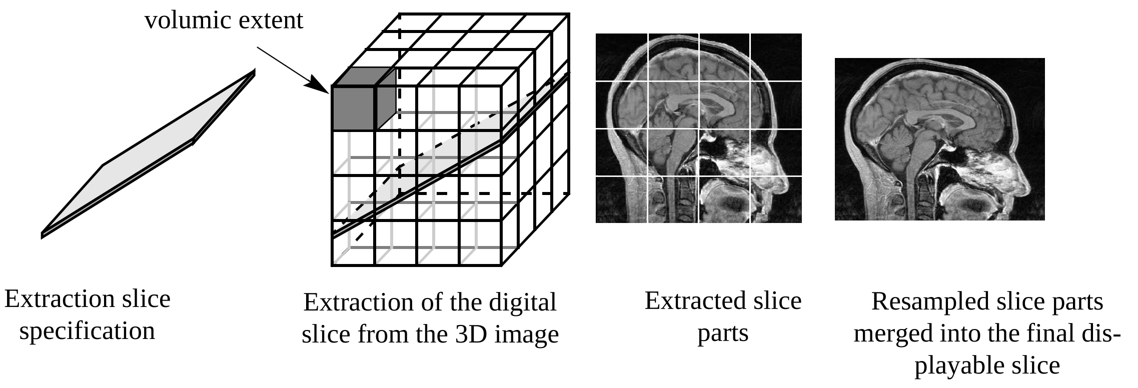

volumic extent
Extraction slice specification
Extraction of the digital slice from the 3D image
Extracted slice parts
Resampled slice parts merged into the final displayable slice

Figure 4-1: Extraction of slice parts from volumic file extent

Based on these principles, a parallel PC-based tomographic slice server was created using the CAP parallelization tool [35] and the library of parallel file striping components [59, 60], both of them described in Chapter 2. This server works in a pipelined-parallel manner and combines high-performance computing and I/O intensive operations. It offers to any client the capability of interactively specifying the exact position and orientation of a desired slice and of requesting and obtaining that slice from the 3D tomographic volume. Performances scale close to linearly from one PC with 3 disks to 5 PCs with 60 disk [58]. A scaled down version of the server (1 Bi-Pentium II with 16 disks) offers its slicing services on the Web (http://visiblehuman.epfl.ch) for accessing the Visible Human data set [1].

4.3 The 4D Beating Heart Slice Stream Server Application

The beating heart dataset consists of a sequence of 8-bits 3D volumic images, each one of size 512 x 512 x 512 (i.e. 128 MBytes). With 320 time instants, the 4D beating heart sequence reaches a size of 320 x 128 MBytes = 40 GBytes.

As shown in Fig. 4-2, each 3D volume of the beating heart is segmented into sub-volumes of size 16x16x16 voxels. A sequence of 16 successive sub-volumes is packed into a sequence of bytes making up a stripe unit, called 4D extent. The size of a 4D extent is 16x16x16x16 = 64 KBytes.

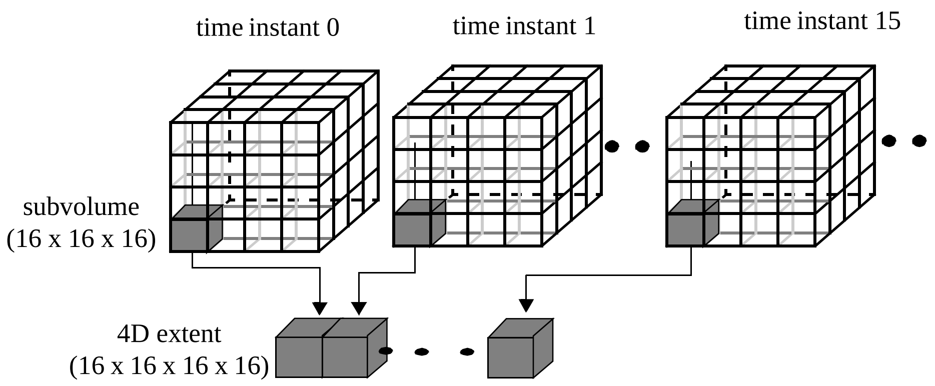
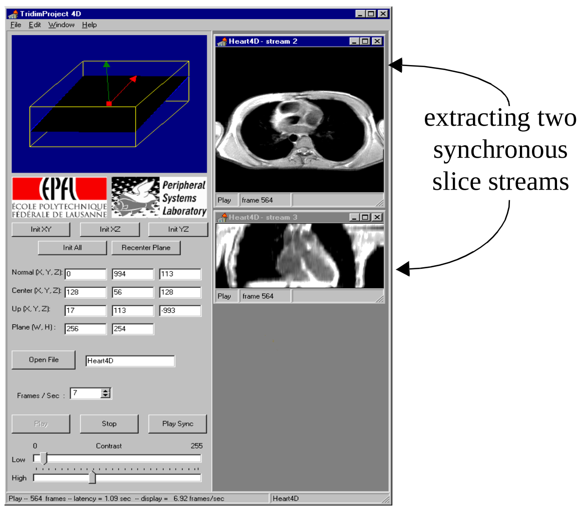


Figure 4-2: 4D extent created from a sequence of 3D subvolumes

Creating extents which incorporate sub-volumes belonging to several consecutive time instants reduces the number of disk accesses, since the visualization of consecutive slices in time requires sub-volumes associated to consecutive time instants. For a same extent size, the incorporation of several time instants reduces the extent's spatial volume. The smaller the spatial subvolumes, the less information is to be read from disks in order to extract a full slice.



Normal (X, Y, Z): 0 994 113
Center (X, Y, Z): 128 56 128
Up (X, Y, Z): 17 113 -993
Plane (W, H) : 256 254
Heart4D
Frames / Sec : 7
Contrast 0 255 Low High
frame 564
frame 564
Play -- 564 frames -- latency = 1.09 sec -- display = 6.92 frames/sec Heart4D

Figure 4-3: Client's graphic user interface

The client interface of the beating heart server (Fig. 4-3) enables users to specify the position and orientation of a 2D slice within the 3D volume. Then, the beating heart server executes extent accesses and slice extractions in order to create and send the desired slice stream (sequence of slices over time) at a user-specified rate to the client. A slice stream requires the extraction of the "same" slice from consecutive 3D image volumes. A slice is extracted from a

3D image by extracting sub-slices from subvolumes and merging them into a full slice as described in section 4.2 (Fig. 4-1).

The server application consists of a proxy residing on the client's site and of server processes running on the server node's parallel processors. Once a slice stream access request is accepted, the proxy registers the slice stream parameters (slice orientation, slice position, display rate, stream duration) in the server nodes whose disks contain extents of the required stream. The proxy sends a streaming request to each implicated server node who generates at regular time intervals the requests for the slices making up the slice stream. From the slice stream parameters, each server node determines the extents which reside on its disks and accesses them. They extract the slice parts from the extents and send them to the client proxy, which assembles them into displayable slices. After having constructed a set of 16 slices, the proxy can start to display the slices at the specified rate while the next slice set is being prepared (Fig. 4-4).

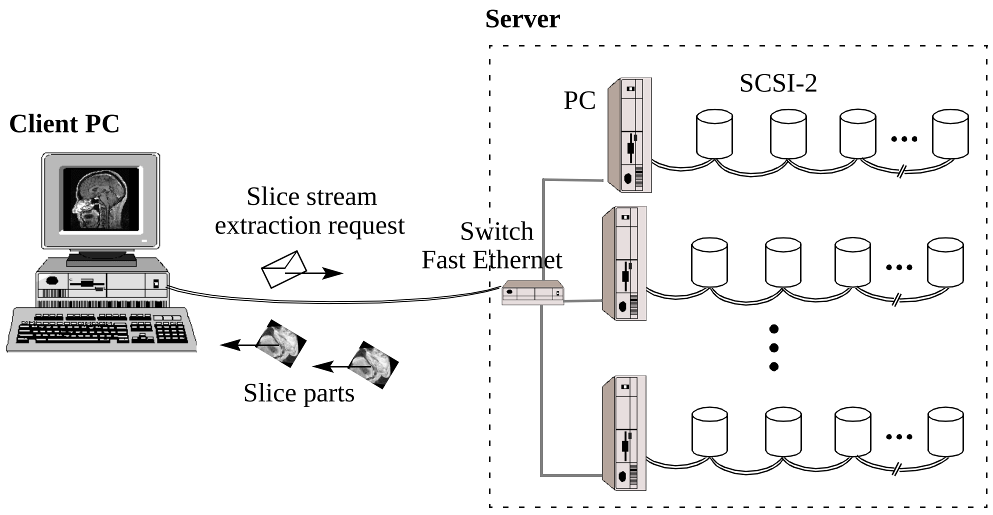


Figure 4-4: Sending a slice stream extraction request and receiving the corresponding slice parts

Users may also specify (Fig. 4-3) different slice streams (i.e. streams with different slice positions and/or different slice orientations) and visualize them synchronously.

4.4 The parallel 4D image slice stream server

The beating heart server consists of the threads defined by the parallel stream server library described in Chapter 3. The beating heart server introduces specific CAP operations mainly for initializating a 4D image slice stream and for extracting a slice stream from a 4D continuous media image file striped over a number of disks.

From the slice stream parameters (slice orientation and position, stream rate and duration) and from the strategy for partitioning and distributing the beating heart data set across the server disks, the *ServerClient* thread (section 3.4) determines the bandwidth of each server disk required for serving that slice stream. The resource requirements are sent to the *InterfaceServer* thread who applies the admission control algorithm (section 3.7) in order to accept or refuse the stream. If the required resources are available, the stream is accepted and the resources for serving it are reserved.

Once the required resources are reserved, the slice stream has to be initializated before starting to extract the stream slices, i.e. the stream objects created on each involved server node by an open striped stream operation (section 3.6) are initializated according to the slice stream parameters (slice orientation and position, stream rate and duration). In addition to the slice stream parameters (e.g. the parameters that describe the server node's isochrone behavior), the stream objects contain the 4D image description (image size, extent size, the segmentation and distribution strategy). As we will see later, the stream objects will be used by the server threads (1) for generating the successive slice requests at regular time intervals, (2) for identifying the extents intersecting a slice, and (3) for extracting the slice parts from extents and resampling them.

4.5 Parallel Stream Slice Retrieval Operation

Once the resources for serving the stream have been reserved and the slice stream is initializated, the parallel slice stream retrieval operation can start. Let us identify the basic sub-tasks that compose the parallel retrieval and presentation of a slice stream:

- compute the value of the parameters describing the isochronicity behavior of each server node contributing to the stream,
- generate the successive slice requests that form the slice stream at regular time intervals,
- compute the extents intersecting a slice,
- read an extent from a single disk,
- extract a slice part from an extent and resample it onto the display grid,
- merge an extracted and resampled slice part into a full slice,
- visualize a full slice on the client computer.

Fig. 4-5 shows the general schedule of these basic operations. The input comprises the indices of the concerned server nodes and the descriptors of the slice stream objects created on these server nodes. Remember that stream objects contain the parameters describing the server node's isochrone behavior in addition to the slice orientation and position, the display rate and the slice range forming the stream. From the input parameters, a *ServerClient* thread identifies the server nodes contributing to the stream and for these server nodes. The stream descriptors are sent to all the *StreamTimerServer* threads located on the identified server nodes. Using the system timers and the previously initializated isochronicity parameters, each *StreamTimer*Server thread generates at regular time intervals the timed slice extraction requests to generate the slice

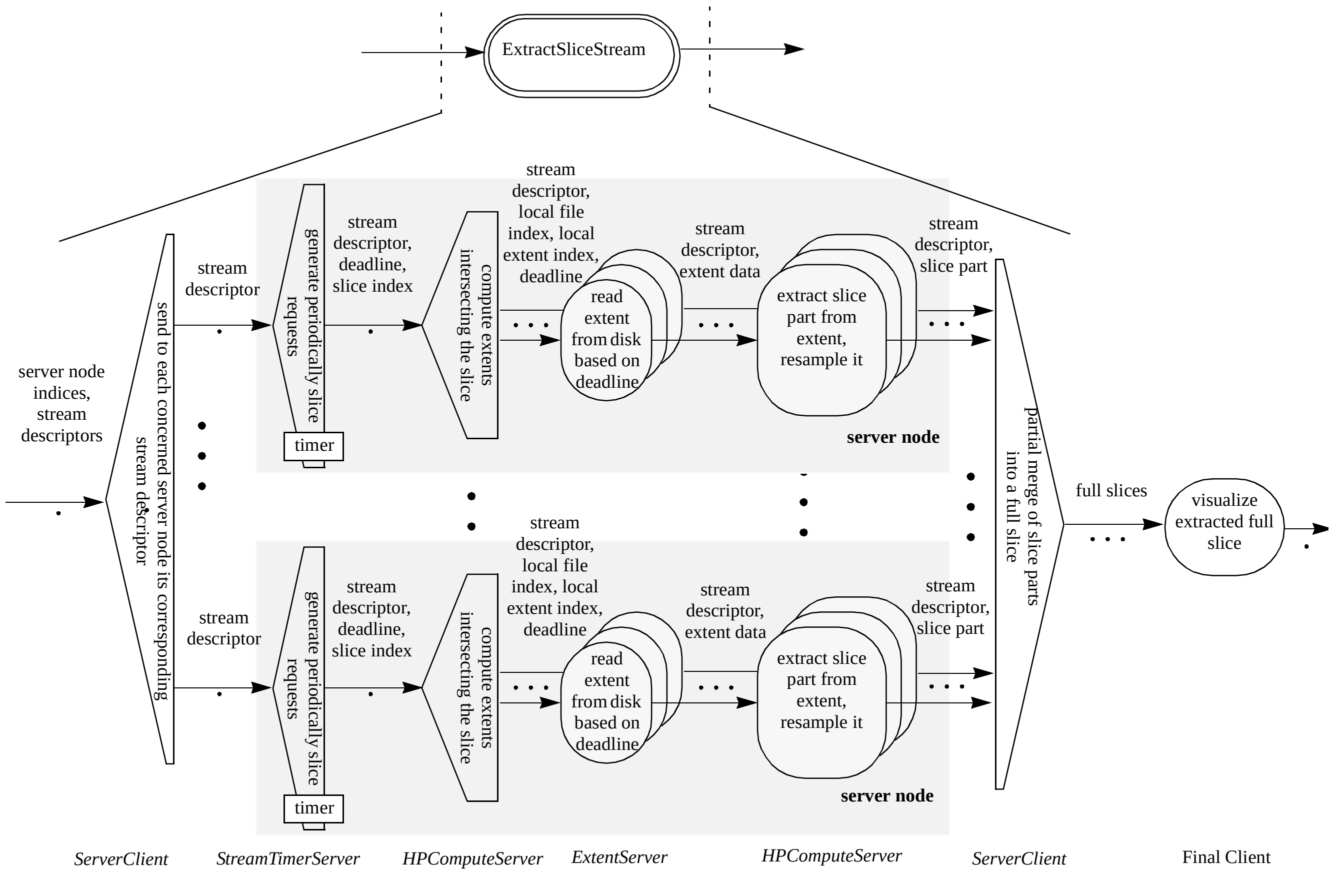

ExtractSliceStream
server node indices, stream descriptors
send to each concerned server node its corresponding stream descriptor
stream descriptor
generate periodically slice requests
timer
stream descriptor, deadline, slice index
compute extents intersecting the slice
stream descriptor, local file index, local extent index, deadline
read extent from disk based on deadline
stream descriptor, extent data
extract slice part from extent, resample it
stream descriptor, slice part
server node
partial merge of slice parts into a full slice
full slices
visualize extracted full slice
ServerClient
StreamTimerServer
HPComputeServer
ExtentServer
HPComputeServer
ServerClient
Final Client

Figure 4-5: Graphical representation of the pipelined-parallel slice stream extraction and visualization operation

stream. Each slice extraction request contains the stream descriptor, a deadline and a slice index specifying the slice position in time within the stream. The time interval between slice extraction requests is derived from the display rate and is part of the isochronicity parameters. Each slice extraction request is directed to a *HPComputeServer* thread running in the same server node. Timed extent reading requests are generated with a deadline derived from the slice extraction request deadline. There is one reading request per local extent intersecting the slice. The reading requests are sent to the *ExtentServer* thread which schedules the disk access requests (section 3.9) and then launches them to the disk (according to their deadline). Once an extent is read, it is processed by the *HPComputeServer* thread in order to extract and resample the corresponding slice part. The resulting extracted and resampled slice part is sent back to a *ServerClient* thread and merged into a full slice. Once constructed and buffered, full slices are ready to be sent to the final client and visualized at the desired display rate.

4.6 Modifying the data delivery model

The data delivery model described by the previous slice stream extraction operation corresponds to the server-push model [55] (as opposed to the client-pull model[1]). In the server-push model, once the stream extraction starts, the server sends stream data to the client at a controlled rate until the stream terminates or the client specifically sends a request to stop the stream delivery. In a parallel server, the server-push model may cause a synchronization problem due to the parallel transmissions from multiple independently running server PCs each having a different clock. To improve the synchronization between parallel server PCs, the stream request is divided into requests for stream pieces (e.g. a 10 minutes stream is divided into 10 pieces of 1 minute substreams). The client makes timed requests for the stream pieces and therefore resynchronizes the server PCs at each stream piece request. This pattern combines the client-pull and server-push delivery models. Fig. 4-6 shows the graphical representation of the combined data delivery model for extracting a slice stream in the beating heart slice stream server. A *Server-Client* thread divides a slice stream request into slice sub-stream requests by setting the slice range of the sub-stream request to the adequate value (e.g. a 5 slice/s stream request ranging from slice 0 to slice 59 can be divided into 3 slice sub-stream requests, the first sub-stream ranging from slice 0 to slice 19, the second ranging from slice 20 to slice 39 and the third ranging from slice 40 to slice 59). Using the system timers, the *ServerClient* thread requests these sub-streams at the appropriate time instants (for the previous 5 slice/s stream example, the 3 slice sub-streams are requested successively at time 0 s, 4 s and 8 s). For extracting a slice sub-stream, the *ServerClient* thread calls the slice stream extraction operation described in section 4.5.

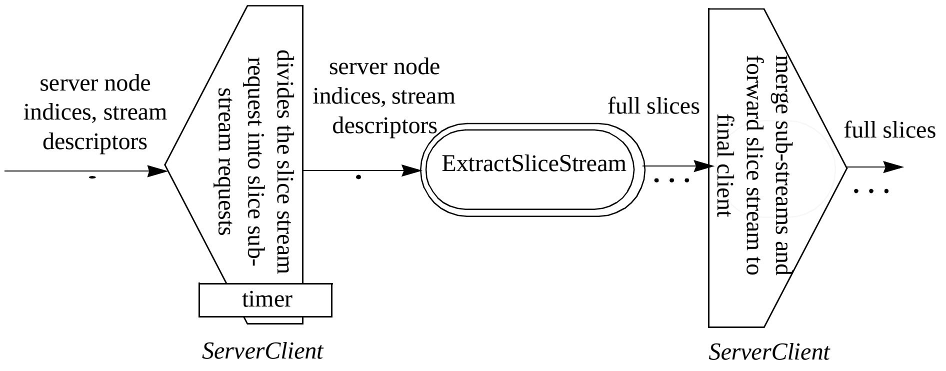

server node indices, stream descriptors
divides the slice stream request into slice sub-stream requests
timer
ServerClient
server node indices, stream descriptors
ExtractSliceStream
full slices
merge sub-streams and forward slice stream to final client
ServerClient
full slices

Figure 4-6: Graphical representation of the pipelined-parallel slice stream extraction and visualization operation according to the data delivery model combining the server-push and client-pull models.

4.7 Extracting multiple streams

The beating heart slice stream server allows users to extract multiple synchronous slice streams. At the same time, several asynchronous calls to the slice stream extraction operation (section

1. The client-pull data delivery model corresponds to the traditional request-response model where the client sends a request to the server for a particular piece of stream data.

4.5) may be launched. Fig. 4-7 shows the graphical schedule for extracting multiple synchronous slice streams. A table specifies the parameters of each of the requested slice streams. For each table entry, the *ServerClient* thread makes a slice stream request and calls the slice stream extraction operation.

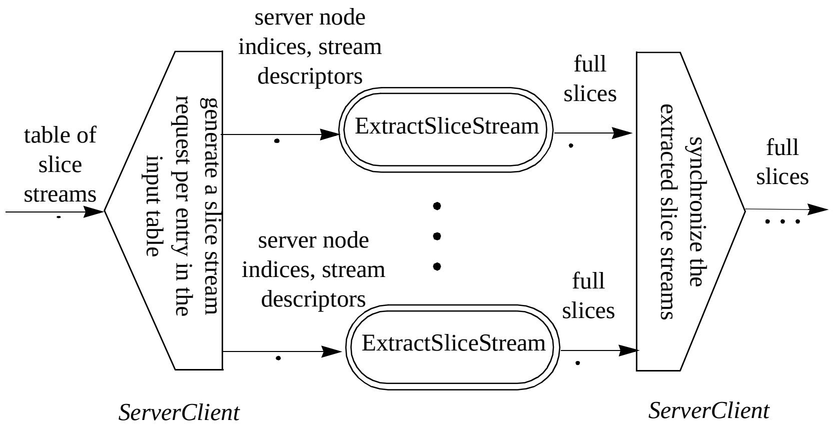

server node indices, stream descriptors
full slices
table of slice streams
generate a slice stream request per entry in the input table
ExtractSliceStream
synchronize the extracted slice streams
full slices
server node indices, stream descriptors
full slices
ExtractSliceStream
ServerClient
ServerClient

Figure 4-7: Graphical representation of the operation for extracting multiple slice streams.

4.8 Configuration file of the *4D beating heart slice stream server*

A configuration file maps the parallel and continuous media server threads onto the available processors of the hardware architecture (Fig. 4-8). Changing the configuration file enables the same program to run without recompilation on different hardware configurations. In this example, process A runs the user application on the user machine. Processes B and C execute the server application on machines whose IP addresses are 128.178.75.65 and 128.178.75.66 respectively. In the CAP program, there are 9 threads. The *InterafaceServer* thread and a *ServerClient* thread run in process A. Threads *ServerNode[0].StreamTimerServer*, *ServerNode[0].HPComputeServer*, *ServerNode[0].ExtentFileServer* and *ServerNode[0]. ExtentServer* run in process B. Threads *ServerNode[1].StreamTimerServer*, *ServerNode[1]. HPComputeServer*, *ServerNode[1].ExtentFileServer* and *ServerNode[1].ExtentServer* run in process C.

```
configuration {
processes:
  A ("User") ;
  B ("128.178.75.65", "\\Shared\Server.exe");
  C ("128.178.75.66", "\\Shared\Server.exe");
threads:
  "StreamServer.ServerClient" (A) ;
  "Ps2Server.InterfaceServer" (A);
  "StreamServer.ServerNode[0].StreamTimerServer" (B);
  "StreamServer.ServerNode[0].HPComputeServer" (B);
  "Ps2Server.ServerNode[0].ExtentFileServer" (B);
  "Ps2Server.ServerNode[0].ExtentServer" (B);
  "StreamServer.ServerNode[1].StreamTimer" (C);
  "StreamServer.ServerNode[1].HPComputeServer" (C);
  "Ps2Server.ServerNode[1].ExtentFileServer" (C);
  "Ps2Server.ServerNode[1].ExtentServer" (C);
};
```

Figure 4-8: Configuration file mapping the parallel and continuous media server threads onto different server nodes.

Each *ExtentServer* thread and its companion *HPComputeServer* thread work in pipeline; multiple pairs of *ExtentServer* and *HPComputeServer* threads may work in parallel if the configuration map specifies that different *ExtentServer*/*HPComputeServer* threads are mapped onto different processes running on different computers (Fig. 3-2). In addition, by being able to direct at execution time the *ReadExtent* and *ExtractAndResampleSlicePart* operations to the storage server node where the extents resides, operations are performed only on local data and superfluous data communications over the network are completely avoided. Load-balancing is ensured by appropriate distribution of extents onto the disks (section 4.2).

4.9 Performance and Scalability Analysis

The server architecture we consider comprises 4 200MHz Bi-PentiumPro PC's interconnected by a 100 Mbits/s Fast Ethernet crossbar switch (Fig. 4-4). Each server PC runs the Windows NT Workstation 4.0 operating system, and incorporates 12 SCSI-2 disks divided into 4 groups of 3 disks, each hooked onto a separate SCSI-2 string. We use 5400 rpm disks which have a measured mean physical data transfer throughput of 3.5 MB/s and a mean latency time, i.e. seek time + rotational latency time, of 12.2 ms. Thus, when accessing 64 KB blocks, i.e. 16x16x16x16 8-bit extents located at random disk locations, an effective throughput of 2.05 MB/s per disk is reached.

In addition to the server PCs, one client 200MHz Bi-PentiumPro PC located on the network runs the 3D beating heart visualization task which enables the user to specify interactively (Fig. 4-3) the desired slice stream access parameters (position, orientation and stream rate) and interacts with the server proxy to extract the desired slice stream. The server proxy running on the client sends the slice stream request to the server PCs, receives the slice parts and merges them into a set of displayable slices. The set of displayable slices is then passed to the 3D beating heart visualization task at the specified rate.

A TCP/IP socket-based communication library [60] called MPS implements the asynchronous SendMessage and ReceiveMessage primitives enabling CAP generated messages, i.e. tokens, to be sent from the application program memory space of one PC to the application program memory space of a second PC, with at most one intermediate memory to memory copy at the receiving site.

The present application comprises several potential bottlenecks: insufficient parallel disk I/O bandwidth, insufficient parallel server processing power for slice part extraction and resampling, insufficient network bandwidth for transferring the slice parts from server PC's to the client PC and insufficient processing power at the client PC for receiving many network packets, for assembling slice parts into the final image slice and for displaying the final image slice on the user's window.

To measure the slice stream extraction and visualization application performances, the experiment consists of requesting and displaying a 512x512 8-bit/pixel slice stream comprising 320 slices according to the server-push delivery model. In order to test the worst case behavior, the selected slice orientation is orthogonal to one of the diagonals traversing the Beating Heart's rectilinear volume.

In the experiment, 1 stream extraction request of 120 bytes is sent to each server PC. For each set of 16 consecutive slices, 1504 4D extents of size 64 KB (i.e. 16x16x16x16) are read in average from the disks (94 MB). The 16 consecutive slice parts (each one of size 390 bytes approx.) contained in a 4D extent are extracted, packed and sent to the client, i.e 1504 messages of size 6.24 KB (i.e. 16 x 390 bytes) are sent back to the client for each set of 16 512x512 image slices.

Fig. 4-9 (obtained from the Windows NT performance monitor) shows the load behavior over time of one server PC belonging to a 3 PC 24 disks server configuration. In Fig. 4-9a, one 512x512 8-bit/pixel slice stream is extracted at 6 slices/s. and in Fig. 4-9b two synchronous 512x512 8-bit/pixel slice streams are extracted at 3 slices/s. Each cycle corresponds to the extraction of (a) one set of 16 slices and (b) two sets of 16 slices. It shows that while disk accesses are made, i.e. extents become available, the processor is 100% busy extracting slice parts.

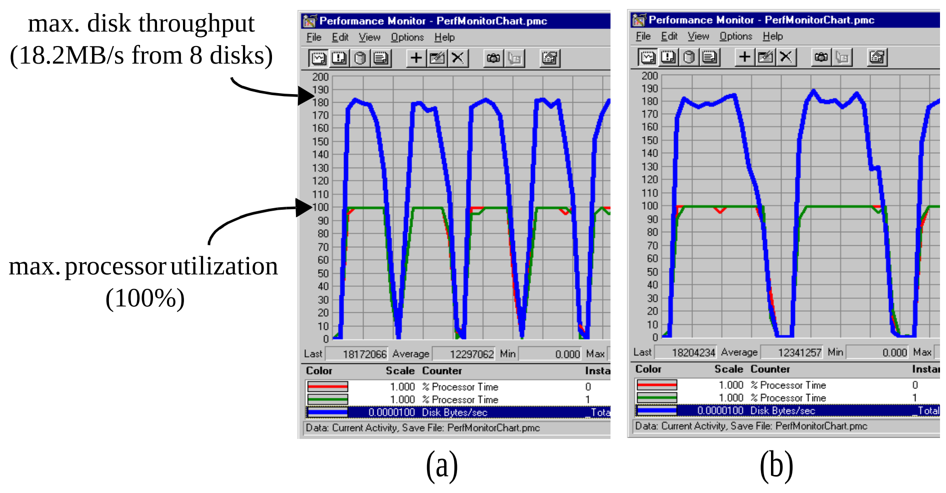


Figure 4-9: Processor utilization and disk throughput of one server PC with 8 disks belonging to a 3 PC 24 disks server configuration, (a) when extracting one 512x512 8-bits slice stream at 6 slices/s and (b) when extracting two synchronous 512x512 8-bits slice streams at 3 slices/s

Fig. 4-10 shows the performances obtained, in number of image slices per second, as a function of the number of contributing server PCs and a function of the number of disks per contributing server PC. With up to 7 disks per server PC, disk I/O bandwidth is always the bottleneck. Therefore increasing either the number of disks per server PC or the number of server PCs (assuming each PC incorporates an equal number of disks) increases the number of disks and offers a higher extracted image slice throughput.

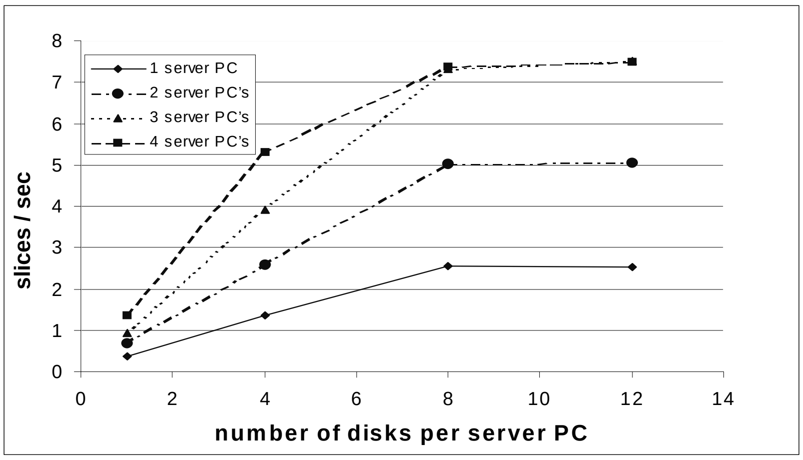


Figure 4-10: Performances extracting a 512x512 8-bits slice stream

From 8 disks per server PC, the bottleneck shifts from the disks to the limited processing power available on the server PCs (Fig. 4-11). At 99% server processor utilization, 82% are dedicated for slice part extraction and resampling, 3% for extent reading and 14% for the network interface and system activities.

From 4 server PCs, each with 8 disks (Fig. 4-11), the client PC is the bottleneck. 22% processor utilization is required for merging slice parts into a full slice and visualizing the full slices and 67% processor utilization is dedicated for the network interface and system activities. One of the two processors of the Bi-PentiumPro client PC is used at 95% for the network interface and system activities and is therefore the bottleneck.

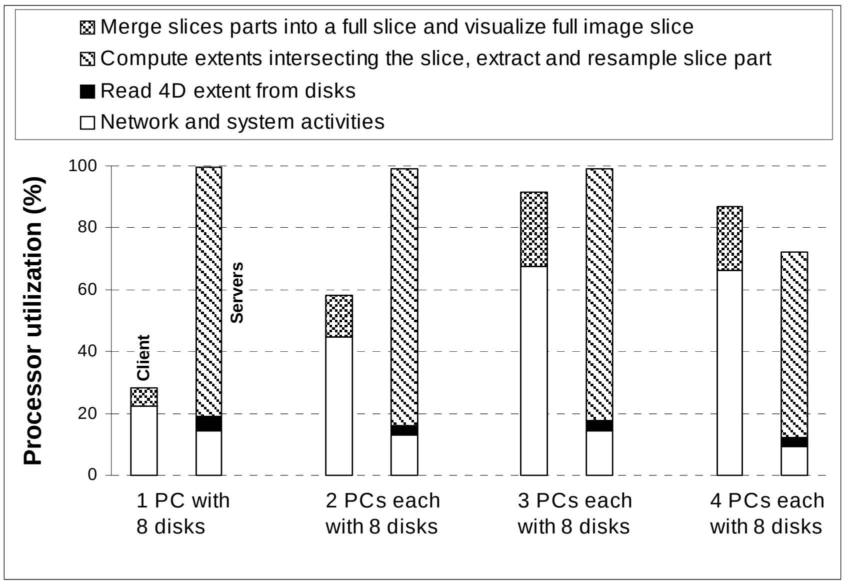


Figure 4-11: Client and server processor utilization for different server configurations

For a single client, the optimal configuration consists of 3 server PCs and 24 disks. With such a configuration, up to 7.3 full slices/s can be generated. This corresponds to an aggregate disk throughput of roughly 43.24 MB/s (7.36/16 * 94 MB), i.e. 1.80 MB/s per disk. This aggregate disk throughput is slightly below the 2.05 MB/s measured mean throughput of individual disks due to the fact that with 8 disks per PC, the processor is the bottleneck.

To analyze the delay jitter of slice parts delivered by the server as a function of the server processor utilization, we consider a single slice stream request at a given nominal rate and two synchronous slice stream requests at half the nominal rate. Fig. 4-12 shows the delay distribution and its cumulative probability distributions (cpd) for server processor utilizations of 52% (corresponding to the extraction of one slice stream at the rate of 4 slices/s) and 92% (7 slices/s). For a 92% processor utilization, the jitter delay (1.4 s) is slightly more than double the maximum jitter delay (0.6 s) at a 52% processor utilization.

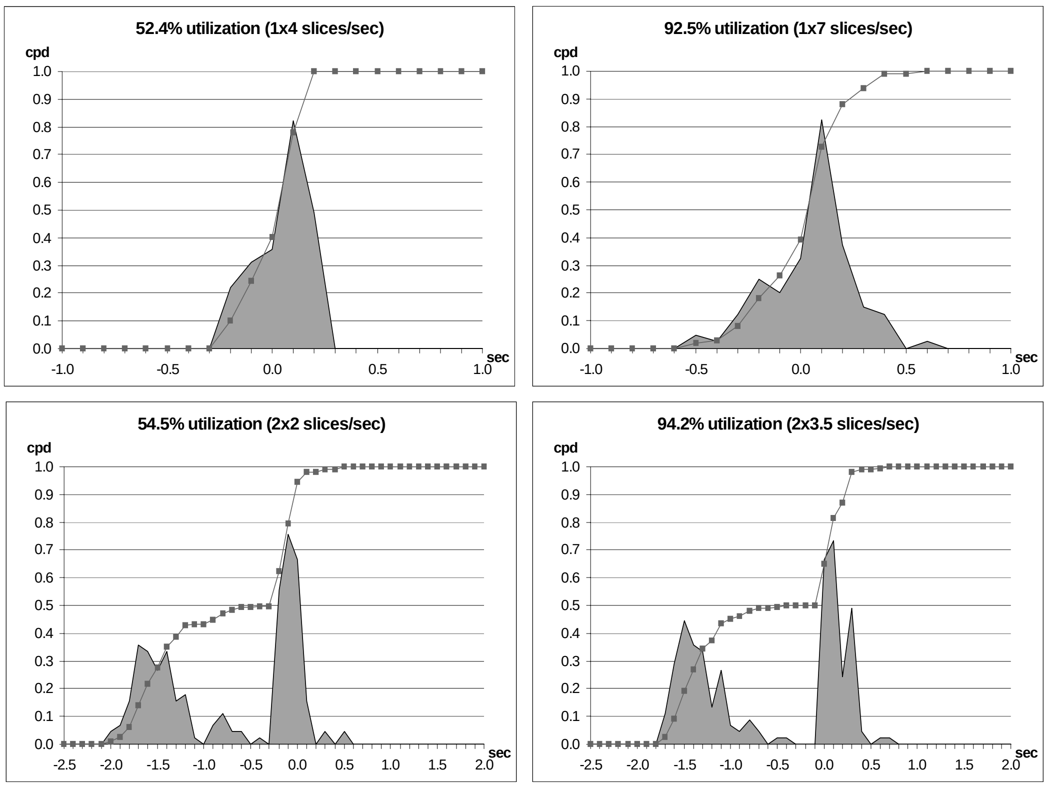


Figure 4-12: Delay jitter for one single stream and for two synchronous streams

Fig. 4-12 shows the jitter delay when extracting two synchronous slice streams at 2 slices/s (54.4% processor utilization) and 3.5 slices/s (94.2% utilization). Due to the organization of 4D extents which incorporate data for 16 consecutive slices parts, accessing two synchronous streams requires reading from disks simultaneously the extents needed for two times 16 consecutive slices, i.e. 32 consecutive slices. At the same total display rate, in the case of two synchronous streams, double the number of disk accesses are made at each deadline. However, as Fig. 4-9b shows, the interval between deadlines is twice as large as in the equivalent single stream access application. This also explains why the delay jitter for two streams is approximately double the delay jitter for a single stream (Fig. 4-12). Slice parts belonging to 16 consecutive time instants are sent to clients. In double buffering mode, clients should therefore have the memory to store 32 slices per slice stream (8 MB per 512x512 8-bit/pixel slice stream). Since the delay jitter is always less than the time to display 16 slices, delay jitter does not require additional buffer space.

4.10 Summary

In this chapter, we presented the 4D beating heart server. This server requires both a high I/O throughput for accessing from disks 4D extents intersecting the desired slices and a large amount of processing power to extract slices from 4D extents and resample them into the display grid. In order to obtain a slice set of 16 consecutive full slices, the server needs to read from the disks 1504 extents of size 64KB, i.e. 94 MB. From these extents, 1504 slice parts, each of 6.24 KB are extracted, resampled and merged at the client site (9.3 MB). With a server configuration of 3 Bi-Pentium Pro PCs and 24 disks (physical throughput: 3.5 MB/s, latency 12.2 ms), up to 7.3 slices can be delivered per second, i.e. 43 MB/s are continuously read from disks and 4.1 MB/s of slice parts are extracted, transferred to the client and merged. This performance is close to the maximal performance deliverable by the underlying hardware.

In the case of a single stream, and at a display rate of 52% of the maximal display rate, the delay jitter is 0.6 s. At 92% of the maximal display rate, the jitter grows to 1.4 s. For the same resource utilization, the jitter is proportional to the number of streams that are accessed synchronously. As long as the worst case delay jitter is smaller than the time to display a slice set, it does not require more memory than the memory for double buffering a full slice set (presently 16 slices).

The presented 4D beating heart application shows that thanks to the parallel stream server library described in Chapter 3, computation- and I/O-intensive continuous media server applications can be built on top of a set of simple PCs connected to SCSI disks. The creation of other continuous media applications may rely on the parallel stream server library. Data structures, individual operations and indices specifying thread locations would need to be appropriately modified.

The 4D beating heart application also suggests that tomographic equipment manufacturers may offer continuous 3D volume acquisition equipment by interfacing the acquisition device with a cluster of PCs, each PC being connected to several disks.

5 Scalable Terabyte Optical Jukebox Server

5.1 Introduction

This chapter describes a scalable server based on optical disk jukeboxes. Such an architecture offers highly accessible terabyte storage capacities with significant cost advantages. Examples of applications requiring access to large volumes of data are NASA's earth observing system (EOS) which stores more than a petabyte of data [23], telecommunication service providers which store terabytes of phone call data for billing, data mining, and fraud detection, digital libraries, image repositories and video-on-demand servers which also store extremely large data volumes. For large data storage systems, optical jukebox storage systems are cheaper than storage systems based entirely on magnetic disks. The scalable server architecture we propose comprises several PCs, magnetic disks and optical jukeboxes.

Server scalability can be achieved by increasing the number of optical disks, drive units, robotic devices and server PCs. An analytical performance model has been created in order to analyze the potential bottlenecks of a jukebox server and to compute its overall throughput, taking into account the robotic device, the optical disk drives as well as the client request arrival rate. The analytical model is extended by a simulation model and verified by specific performance measurements. Using optical jukeboxes as the backbone of a scalable server requires means to scale the I/O throughput. This can be achieved by striping files [68, 13, 30] across multiple optical disks. These disks may be loaded into different drives located either on a single or on several server PCs. A global file can be read at a high throughput by reading its subfiles simultaneously from the different optical disks.

Magnetic disks are used as caches [49, 50] in order to improve the response time when frequently accessed data is read from robotic storage libraries. We present the server's strategy for delivering files either directly from the optical disks or from the magnetic disk cache.

Finally, the software architecture for the scalable terabyte server and the files required to configure the terabyte server are described. The software architecture supports continuous media access as well as parallel access to multiple optical disks.

5.2 Related work

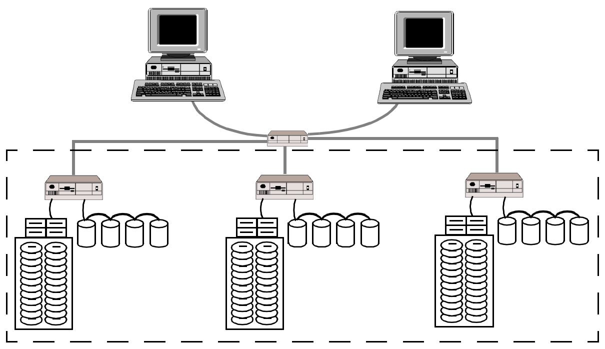
Previous work related to the use of optical jukeboxes for information servers focussed mainly on special purpose applications such as video-on-demand servers [16, 6, 24, 29]. An early analysis [16] showed that in the case of video servers, simple jukeboxes with a limited number of drive units cannot compete with magnetic disk storage due to their limited throughput capabilities. However for video storage, a storage hierarchy comprising CD-ROM jukeboxes and magnetic disks provides a cost-effective solution [39, 53, 11, 28, 43]. Unfrequently accessed videos are stored on removable media and popular videos are stored on magnetic disks.

Several articles describe striping techniques to improve tape robot system performances [26, 27, 40, 14, 17]. Drapeau and Katz show how to extend RAID technology [68, 13] to tape robot systems [26, 27]. In [14, 17], tape striping is applied to improve tape data transfer rates and response times for large requests. Techniques for overcoming striping-related problems such as synchronization and increased number of tape exchanges are discussed. Berson et al. show how to apply a striping technique in multimedia servers in order to decrease the response time of a robotic tape storage system [6].

In most existing optical jukeboxes, the single robotic device responsible for up- and unloading the available drives is the limiting factor [10, 16].

5.3 Scalable jukebox server architecture

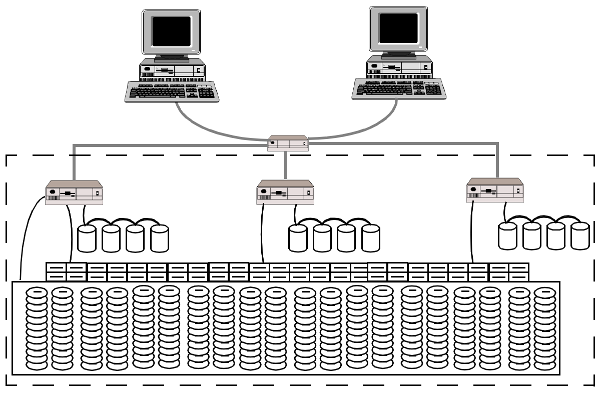
A scalable server architecture based on optical jukeboxes may comprise several server nodes, each one consisting of a number of processors and jukeboxes. Optical jukeboxes, both for CD-ROMs and DVDs have the potential of offering terabyte storage capabilities and scalable I/O throughput. Magnetic disks, connected to the server nodes, can be used as a cache for optical disk files. They allow to significantly increase the number of files that can be served during a given time period. Fig. 5-1 shows a jukebox server made of several small jukeboxes (e.g. NSM jukeboxes [65]), whose drives are connected through SCSI channels to one or several server PCs. Server PCs and client stations may be connected by a Fast or Gigabit Ethernet switch. Fig. 5-2 shows a jukebox server made of one very large jukebox, the GiantROM from ALP Electronics [2].



Single NSM Satellite jukebox (CD-ROM):
disk exchanging time: 8.7 s.
disk spin up time: 8.6 s.
seek time: 70 ms. to 140 ms. according to position
disk throughput: 2MB/s to 4MB/s, according to position
number of robotic devices (RS-232 interface): 1
number of read drives: 4
number of optical disks: 75
total storage space: 44 GB, in the future 300 GB (DVD)

Figure 5-1: Terabyte server made of several small jukeboxes

A scalable jukebox server architecture can combine heterogeneous jukeboxes, i.e. different jukebox models from the same jukebox implementor and different jukebox models from different jukebox implementors. The scalable jukebox server provides the protocols specified by the jukebox implementors in order to control the robotic arm of the different jukebox models.



ALP Giant-ROM jukebox (CD-ROM):
disk exchange time: 12.2 s
disk spin up time: 8.6 s
seek time: 70 ms to 140 ms, according to position
disk throughput: 2MB/s to 4MB/s, according to position
number of robotic devices (RS-232 interface): 2
max. number of read drives: 120
max. number of optical disks: 9695
max. total storage space: 6 TB, in the future 38 TB (DVD

Figure 5-2: Terabyte server made of one large jukebox

5.4 Scalable jukebox architecture model

Central issues for offering scalable throughput to users located on a fast network, e.g. Fast Ethernet or Gigabit Ethernet, are the time required by the robotic device to move an optical disk from a magazine slot to an optical drive unit or vice-versa, the physical bandwidth of a single drive unit as well as its disk spin up, head displacement and mean rotation times, the throughput from the jukebox SCSI strings to server PCs and the throughput from a server PC to the network.

Let us establish an analytical model describing the behavior of a jukebox architecture. The model is expressed in function of the time $T_{exchange}$ required by the robotic device to unload and load successive drives, the number N_{Drives} of drive units, the disk throughput X_{drive} of a single optical drive unit, its disk spin up T_{spinup}, seek T_{seek} and rotation T_{rot} times and the file size F_{size} to be accessed.

First, we examine the behavior of the previous parameters of a NSM Satellite jukebox by performing experimental measurements. The considered NSM Satellite jukebox comprises 1 robotic arm, 4 Plextor PX32CS CD-ROM drives and 5 CD-ROM magazines each one containing 15 CD-ROMs, i.e. 75 CD-ROMs.

Fig. 5-3a shows the histogram of the optical disk exchange time ($T_{exchange}$) when random exchanges are performed on a NSM Satellite jukebox. It may be approximated by a normal distribution with mean equals to 8.7 s and standard deviation equals to 0.26 s, i.e. $N(8.7, 0.26)$.

According to our measurements, the spin-up time histogram of a Plextor PX32CS CD-ROM drive within a NSM Satellite jukebox presents two significative peaks: the first one located about 9 s and the second at 15.7 s. They depend on the type of optical disk. The spin-up times of pressed optical disks (i.e. optical disks produced from a master optical disk) fall around the first peak. However for most optical disks recorded with a CD-Writer (Plextor 12/4/32 SCSI), we measured the highest spin-up times, i.e. those around the second peek. In the present analysis, we will only take into account the spinup times the first peak, i.e. these for pressed disks. We therefore eliminate spinup times generated by recorded optical disks. The considered spinup times (Fig. 5-3b) follow a normal distribution with mean equals to 8.6 s and standard deviation equals to 0.37 s, i.e. *N*(8.6, 0.37).

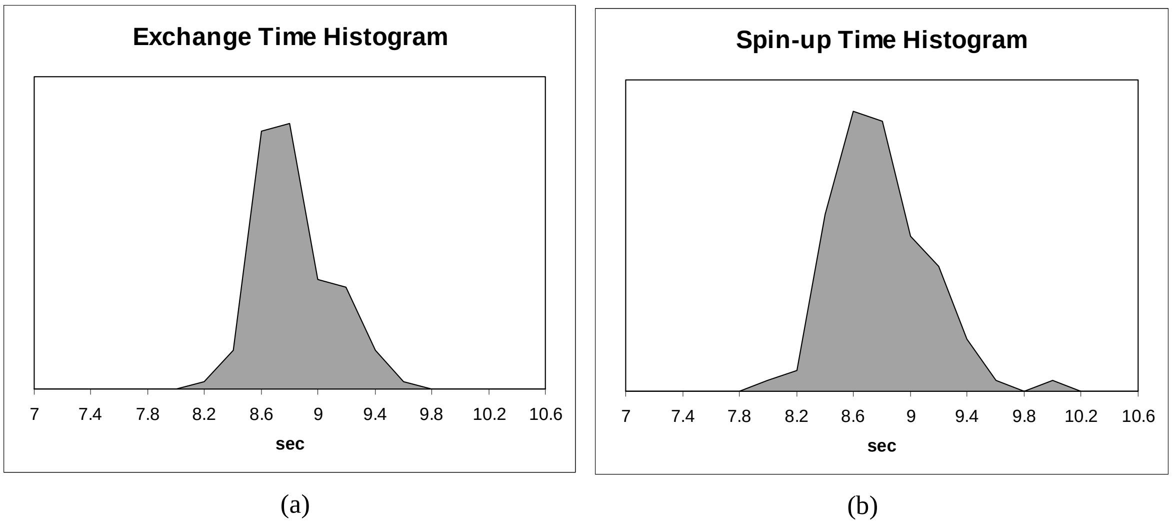


Figure 5-3: Histograms of the optical disk exchange and spin-up times for a NSM Satellite jukebox

According to our measurements, the disk throughput X_{drive} (MB/s) varies as a function of the position (*pos*) of the file on the disk according to Fig. 5-4 and is approximated by

$$X(pos)_{drive} = 2 + \frac{2.5}{600} \cdot pos$$

where *pos* is the position on disk given in MB.

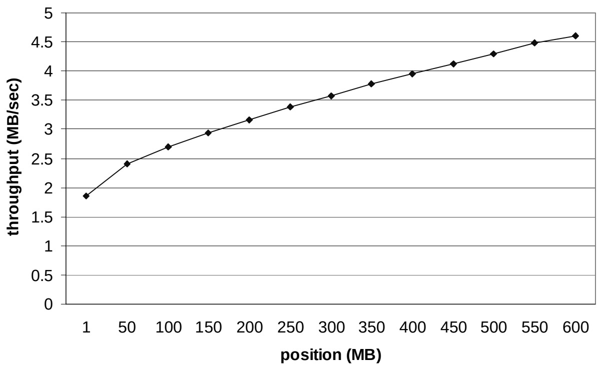


Figure 5-4: Throughput as a function of position on a Plextor PX32CS CD-ROM drive

The mean rotation time is half the time of a single rotation, i.e. 4.3 ms at a rotation speed of 6890 rpm (Plextor PX32CS CD-ROM drive). According to our measurements, the seek time T_{seek} incorporates a latency of 70 ms plus a time varying close to linearly as a function of the amount of head displacement (Fig. 5-5).

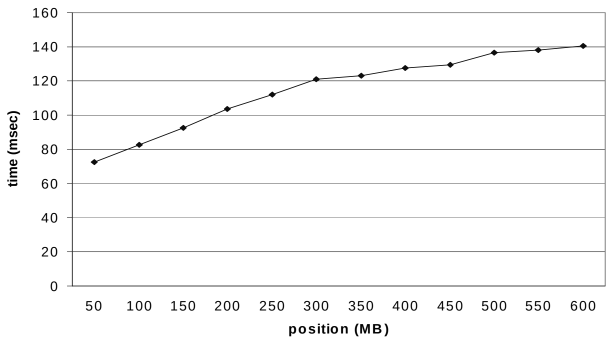


Figure 5-5: Seek time from position 0 to a given position on a Plextor PX32CS CD-ROM drive

When accessing a file from a new optical disk, rotation and seek times are very small compared to exchange time and spin-up time and can therefore be neglected.

5.4.1 Bottleneck analysis

Let us to analyze the potential bottlenecks of a single jukebox assuming that all requested files have the same size. In a jukebox, the critical resources are the robotic device and the drive units. In order to get the highest resource utilization (i.e. drive unit and robotic device utilization), the first jukebox drive (D1 in Fig. 5-6) has to complete its data transfer at the moment when the robotic device finishes loading a disk onto the last drive (D4 in Fig. 5-6). Otherwise, either the robotic device (Fig. 5-7) or the drive units (Fig. 5-8) are idle for certain time periods assuming that the file request arrival rate is high enough. Note that the spinup time can be overlapped, i.e. the robot arm does not have to wait for a completion of disk mounting operation before it can start another disk exchange. However, all the disk exchanges have to be performed sequentially for jukeboxes comprising one single robot arm.

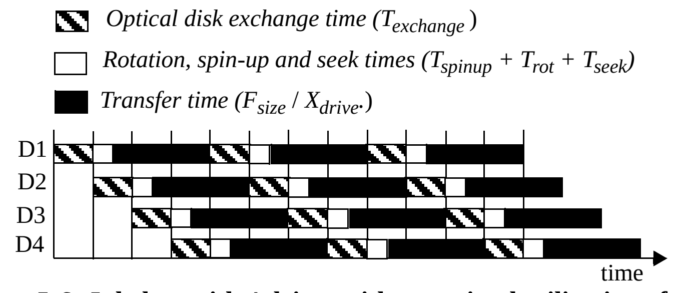

▨ Optical disk exchange time ($T_{exchange}$)
□ Rotation, spin-up and seek times ($T_{spinup} + T_{rot} + T_{seek}$)
■ Transfer time (F_{size} / X_{drive}.)
D1
D2
D3
D4
time

Figure 5-6: Jukebox with 4 drives with a maximal utilization of both the robotic device and the drive units

In the case of a maximal utilization of both the robotic device and the drive units, we have the following equation stating that the robot serves (N_{Drives}-1) drives while exactly one file of size F_{size} is read from the disk.

$$(N_{Drives} - 1) \cdot T_{exchange} = T_{accesstranfer} \qquad (5\text{-}1)$$

where

$$T_{accesstranfer} = T_{spinup} + T_{rot} + T_{seek} + \frac{F_{size}}{X_{drive}}$$

The robot arm's utilization (U_{robot}) is

$$U_{robot} = \frac{N_{drives} \cdot T_{exchange}}{T_{exchange} + T_{accesstranfer}}$$

Since a drive is idle during the exchange of its optical disk, maximal utilization of drive units (U_{Drives})[1] is

$$U_{Drives} = \frac{T_{exchange}}{T_{exchange} + T_{accesstransfer}}$$

In the case of the resource maximal utilization (equation 5-1), the robot arms' utilization is

$$U_{robot} = \frac{N_{drives} \cdot T_{exchange}}{T_{exchange} + T_{accesstranfer}} = \frac{N_{drives} \cdot T_{exchange}}{T_{exchange} + (N_{drives} - 1) \cdot T_{exchange}} = 1$$

and the drive utilization is

$$U_{Drives} = \frac{T_{exchange}}{T_{exchange} + T_{accesstransfer}} = \frac{(N_{Drives} - 1) \cdot T_{exchange}}{T_{exchange} + (N_{Drives} - 1) \cdot T_{exchange}} = \frac{N_{Drives} - 1}{N_{Drives}}$$

From equation 5-1, we can determine the optimal file size ($F_{optimalsize}$) to reach the highest resource utilization for a given jukebox.

$$F_{optimalsize} = ((N_{Drives} - 1) \cdot T_{exchange} - (T_{spinup} + T_{rot} + T_{seek})) \cdot X_{drive} \qquad (5\text{-}2)$$

When file sizes are larger than the optimum, the robotic device waits for the file transfer completion (Fig. 5-7). In order to increase the system performance we can either increase the number of drive units or improve the drive units' performances (i.e. their spin-up time and throughput).

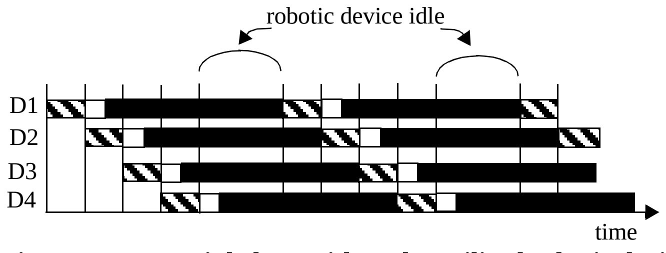


Figure 5-7: Same jukebox, with under utilized robotic device

1. The drive utilization comprises the spinup, the rotational, the seek and the transfer times.

In this case, the robot resource utilization is

$$U_{robot} = \frac{N_{Drives} \cdot T_{exchange}}{T_{exchange} + T_{accesstransfer}} < 1 \tag{5-3}$$

and the drive utilization is maximal

$$U_{Drives} = \frac{T_{accesstransfer}}{T_{exchange} + T_{accesstransfer}} > \frac{N_{Drives} - 1}{N_{Drives}} \tag{5-4}$$

On the other hand, when the file sizes are smaller than the optimum, the drive units wait for the robotic device (Fig. 5-8), which is too slow to unload and load successive drives. In this case, assuming accesses to files of the same size, increasing the number of drive units does not improve the system's performances. Under these circumstances, a cheaper three drive jukebox may offer the same performances as a four drives jukebox.

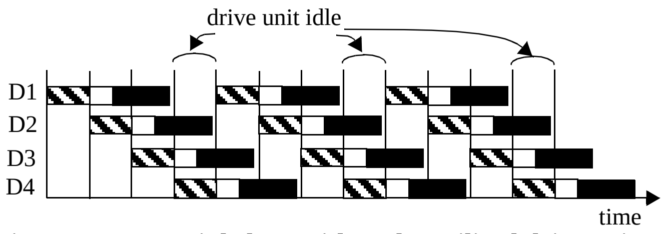


Figure 5-8: Same jukebox, with under utilized drive units

In this case, the robot utilization is

$$U_{robot} = 1 \tag{5-5}$$

and the drive utilization is

$$U_{Drives} = \frac{T_{accesstransfer}}{N_{Drives} \cdot T_{exchange}} < \frac{N_{Drives} - 1}{N_{Drives}}$$

According to the previous equations, Fig. 5-9 plots the maximal robot and drive unit utilization of a NSM Satellite jukebox as a function of the file size. The highest resource utilization is reached when accessing 56 MB files (i.e. files with the optimal size according to equation 5-2). The robot arm is used at 100% and the drive units are used at 75%. At a drive unit utilization higher than 75%, the number of drive units becomes the bottleneck resulting in an exponential decrease of the robotic arm's utilization.

From equation 5-1, we can determine the maximum number of drive units $N_{MaxDrives}$ allowing to reach the highest resource utilization for a given file size.

$$N_{MaxDrives} = \left\lceil \frac{T_{spinup}}{T_{exchange}} + \frac{T_{rot}}{T_{exchange}} + \frac{T_{seek}}{T_{exchange}} + \frac{F_{size}/X_{drive}}{T_{exchange}} + 1 \right\rceil \tag{5-6}$$

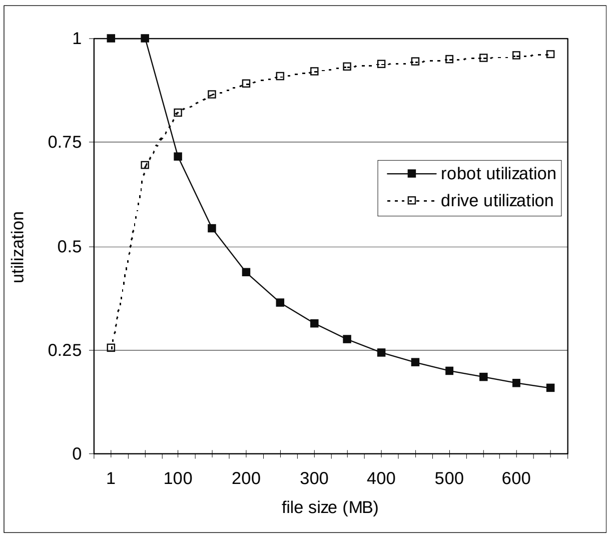


Figure 5-9: Robot and drive unit utilization of a NSM Satellite jukebox as a function of the file size

5.4.2 Throughput analysis

The time T_{tot} to access a file of size F_{size} from a single optical disk located in a magazine slot is

$$T(pos, disp)_{tot} = T_{exchange} + T_{spinup} + T_{rot} + T(disp)_{seek} + \frac{F_{size}}{X(pos)_{drive}}$$

where *pos* and *disp* express respectively the file position and the head displacement to reach that position. The corresponding effective single drive throughput X_{eff} is

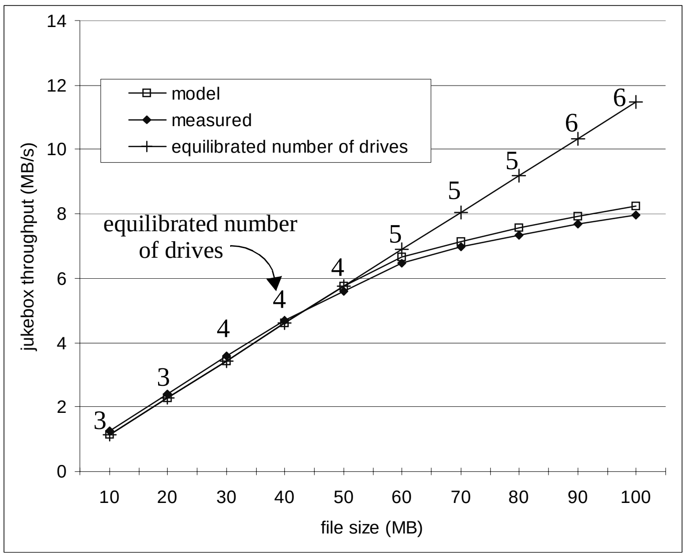
$$X(pos, disp)_{eff} = \frac{F_{size}}{T(pos,disp)_{tot}} = \frac{F_{size}}{T_{exchange} + T_{spinup} + T_{rot} + T(disp)_{seek} + F_{size}/X(pos)_{drive}}$$

If the drive units are fully utilized and the robot arm is partially idle (case of Fig. 5-7), the global mean throughput of a jukebox architecture ($X_{jukebox}$) is the sum of the throughputs of the individual drives.

$$X_{jukebox} = \sum_{i=1}^{N_{Drives}} X(pos, disp)_{eff} \qquad (5\text{-}7)$$

when accessing individual files of size $F_{optimalsize}$ or smaller. However, in most cases, the robotic device is the limiting factor (Fig. 5-8). In these cases, the effective throughput is

$$X_{jukebox} = \frac{\sum_{i=1}^{N_{Drives}} F_{sizei}}{N_{Drives} \cdot T_{exchange}} \qquad (5\text{-}8)$$



number of drive units	optimal file size (MB)
3	28.16
4	56
5	83.84
6	111.68

Figure 5-10: NSM Satellite jukebox's throughput as a function of file size according to the model described above and to the measured results. In addition, the throughput of a jukebox with drives and a robotic device of the performances of a NSM Satellite jukebox but with an equilibrated number of drive units is plotted.

Fig. 5-10 plots the NSM Satellite jukebox throughput as a function of file size according to the model (equations 5-7 and 5-8) and to the experimental measurements. In addition, the throughput of a jukebox with the performances of a NSM Satellite jukebox but comprising an equilibrated number of drive units ($N_{MaxDrives,}$ equation 5-6) is plotted as a function of the file size. We assume that accesses are at uniformly distributed optical disk locations and that there are always requests waiting for service, i.e. a heavy load is present in the system. This graph shows that the model is close to the experimental measurements. Until 6.4 MB/s (obtained when accessing files with the optimal size of 56 MB, equation 5-2), the NSM Satellite jukebox's throughput increases linearly. From 6.4 MB/s, the robotic device is under utilized and a NSM Satellite jukebox needs additional drive units to increase significantly its throughput. The jukebox's throughput increases linearly with an equilibrated number of drive units since the size of files to be accessed is always smaller than the optimal file size and therefore drive units are underutilized. In these cases, the throughput is proportional to the file size. The jukebox serves the same number of files per time unit until the optimal file size is reached. The maximal throughput supported by a jukebox is equal to $N_{Drives}*X_{drive}$, e.g. 4 * 3.2 MB/s = 12.8 MB/s for a NSM Satellite jukebox. To reach 90% of this maximal throughput, we have to access 500 MB files (i.e. 8.92 times larger than the optimal size of 56 MB). In this case, the drive units are used at 95% (equation 5-4, Fig. 5-9), whereas the robotic device is only used at 20% (equation 5-3, Fig. 5-9). This graph also shows that to serve large files, jukeboxes comprising a large number of drive units are needed at full load.

5.4.3 Service rate analysis

Let us to analyze the service rate of a jukebox ($\mu_{jukebox}$) and the effect of the file request rate (λ) when accessing individual files. When accessing files whose sizes are larger than the optimal file size (Fig. 5-7), the number of the drive units is the limiting factor when the file access request rate increases. In this case, the jukebox serves file requests at drive unit service rate μ_{Drives}.

$$\mu_{Drives} = \frac{N_{Drives}}{T_{exchange} + T_{spinup} + T_{rot} + T_{seek} + \frac{F_{size}}{X_{drive}}} \tag{5-9}$$

On the other hand, when the robotic device is the limiting factor (Fig. 5-8), the jukebox serves file requests at the robotic device's service rate (μ_{robot}).

$$\mu_{robot} = \frac{1}{T_{exchange}} \tag{5-10}$$

From the service rate of the robotic arm (μ_{robot}) and of the drive units (μ_{Drives}), we can determine the jukebox's service rate ($\mu_{jukebox}$) as follows

$$\mu_{jukebox} = min(\mu_{robot}, \mu_{Drives}) = min\left(\frac{1}{T_{exchange}}, \frac{N_{Drives}}{T_{exchange} + T_{spinup} + T_{rot} + T_{seek} + \frac{F_{size}}{X_{drive}}}\right)$$

5.4.4 Simulation model

In order to analyze the single-jukebox server's behavior when increasing the request arrival rate (λ), a simulation model of the jukebox server is proposed. The simulation model has the following parameters[1]:

- the arrival rate of requests,
- the position of the file within the optical disk,
- the size of the file to access,
- the service time of the robot arm for loading, unloading or exchanging optical disks,
- the spin up time of a drive to mount a optical disk into a drive,
- the throughput of a drive (according to the file position and size),
- the number of drives

1. The mean rotational time is about 4.3 ms and the seek time ranges from 70 to 140 ms. The simulation model neglects these times since they are very small compared with the service times of the robot arm and the spin-up times (both of them over 8 s).

The model represents the different phases of the file access request service time. We assume that file access requests are independent and follow an exponential distribution. Each request requires one single drive. The arriving request joins an input queue of requests waiting for drives. As a drive is unloaded and becomes free, it is allocated to the request at the front of the queue. Once a drive is allocated, the robot arm loads the appropriate optical disk in it. The drive then preforms the disk mount and carries out a head displacement to the start of the file. At this point the drive is ready to transmit the file data. As soon as a drive terminates its transmission, the request is completed. Then the optical disk needs to unloaded by the robot. In this model we assume that I/O requests concern files located in any of the jukebox optical disks. Since requests arrive from several parts of the network, and the number of optical disks is very large compared to the number of drives, we assume that the probability of accessing an optical disk already being loaded on a drive is close to zero. It is therefore desirable to unload the optical disk as soon as a transmission is terminated. To reduce waiting times, the robot arm execute all disk loading requests before starting disk unloading requests, i.e. loading requests have a higher priority than unloading requests.

In order to validate the model, we simulate the NSM Satellite jukebox and compare its results with the experimental measurements. This is illustrated in Fig. 5-11 where the response time obtained from the simulation model and from the experimental measurements is plotted as a function of the file request arrival rate when accessing to 50 MB files. The simulation results are close to the experimental measurements. By increasing the arrival rate, the response time increases exponentially.

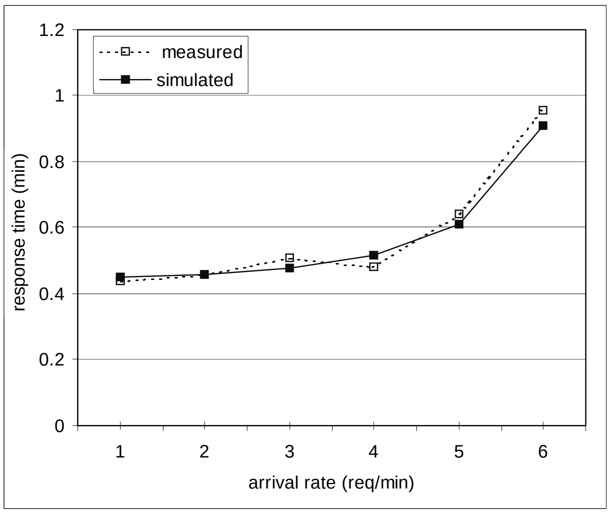


Figure 5-11: Response times obtained from the simulation model and from experimental measurements as a function of the file request arrival rate when accessing to 50 MB files

The single-jukebox server is stable [47], when the mean arrival rate is less than the jukebox's mean service rate, i.e. the stability condition is given by

$$\lambda < \mu_{jukebox}$$

Otherwise the system is unstable since the number of requests in the input queue waiting for service grows continuously and tend to be infinite. In this situation, the jukebox's response time tends to infinity. Applying the stability condition when the jukebox serves 50 MB files (Fig. 5-11), we deduce

$$\lambda < \mu_{jukebox}$$

$$\lambda < min(\mu_{robot}, \mu_{Drives})$$

$$\lambda < min\left(\frac{1}{T_{exchange}}, \frac{N_{Drives}}{T_{exchange} + T_{spinup} + T_{rot} + T_{seek} + \frac{F_{size}}{X_{drive}}}\right)$$

$$\lambda < min(0.114, 0.121) \Rightarrow \lambda < 0.114\frac{request}{s} \Rightarrow \lambda < 6.84\frac{request}{min}$$

This means that when serving 50 MB files, a NSM Satellite jukebox supports a maximal arrival rate of 6.84 req/min before becoming unstable. This rate obtained by the model is close to the experimentally measured maximal sustainable rate (Fig. 5-11). From 6.84 req/min, the robotic arm is the system's bottleneck as shown in Fig. 5-12.

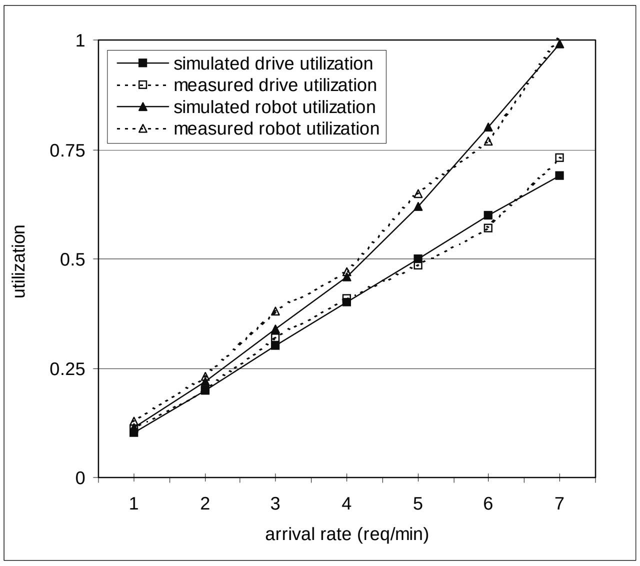


Figure 5-12: Robotic arm and drive units utilization as a function of the file request arrival rate when accessing 50 MB files

Using the previously described model, we simulate the model for two different jukeboxes whose specifications are presented in Table 1.

	unit	4-drive jukebox (NSM Satellite jukebox)	32-drive jukebox
number of drive units (N_{drives})	-	constant, 4	constant, 32
disk exchange time ($T_{exchange}$)	sec	normal distribution, *N*(8.7, 0.26)	normal distribution, *N*(12.2, 0.64)
drive spin up time (T_{spinup})	sec	normal distribution, *N*(8.5, 0.37)	normal distribution, *N*(8.5, 0.37)
file size (F_{size})	MB	constant, from 10 to 500 MB	constant, from 10 to 500 MB
file position (*pos*)	MB	normal distribution, *N*(300,250)	normal distribution, *N*(300,250)
drive throughput (X_{drive})	MB/s	from 1.8 to 4.6 MB/s, according to file position and size	from 1.8 to 4.6 MB/s, according to file position and size

Table 1: Specifications of the simulated jukeboxes

Fig. 5-13 illustrates response times as a function of the arrival rate when accessing small files (10, 50 and 100 MB files). Each curve corresponds to a given jukebox (4-drive or 32-drive) and a fixed file size. The response time comprises: (1) the waiting time in the input queue, (2) the service time of the robot arm, (3) the spin-up time for mounting the optical disk and (4) the file transfer time. Fig. 5-14 illustrates response times as a function of the arrival rate when accessing large files (300, 400 and 500 MB files).

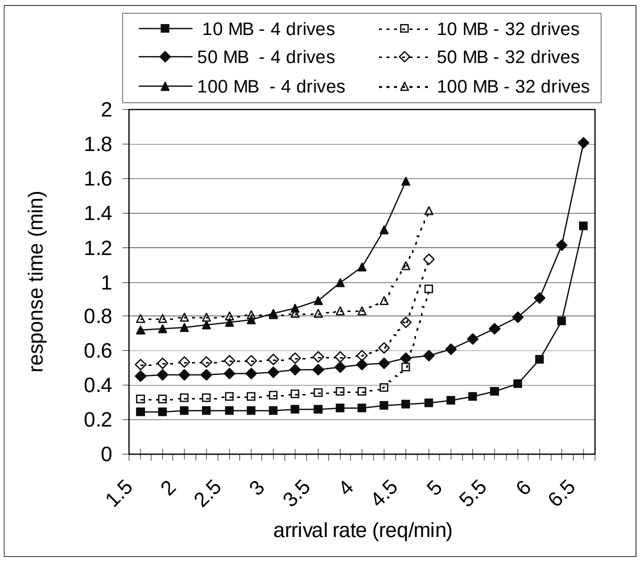


Figure 5-13: Response times as function of the arrival rate for 10, 50 and 100 MB files

When accessing 10 and 50 MB files from the 4-drive jukebox, a maximal arrival rate of about 6.75 req/min is supported. At higher arrival rates, the robot arm is used at 100% and is therefore the system's bottleneck. This can easily be seen in Fig. 5-15a, which plots the robot arm utilization as a function of the arrival rate for different file sizes. When accessing 100, 300, 400 and 500 MB files, the 4-drive jukebox serves at about 4.5, 2, 1.5 and 1.25 req/min respectively. In these cases, the limiting factor is the number of the drive units. Their utilization rate is 82%, 93%, 94% and 95% respectively, close to the drive units' maximal utilization for the considered file sizes (Fig. 5-9). Fig. 5-15b shows the drive unit utilization as a function of the arrival rate. Additional drive units are needed to serve at higher arrival rates. These simulation results match the previous theoretical analysis. Applying equation 5-2 to the 4-drive jukebox yields an optimal file size of 56 MB. When accessing files smaller than the optimum (e.g. accessing 10 or 50 MB files) the supported maximal arrival rate corresponds to the robot arm's service rate (for the 4-drive jukebox, μ_{robot} = 6.89 req/min. from equation 5-10). Conversely, when accessing files larger than the optimum (e.g. accessing 100, 300, 400 or 500 MB files), the maximal arrival rate is limited by the number of drive units to μ_{Drives} requests per time unit (i.e. from equation 5-9, 4/(8.7+8.6+100/3.2) = 4/48.55 req/s = 4.94 req/min, 2.16 req/min, 1.68 req/min and 1.38 req/min for 100, 300, 400 and 500 MB files respectively).

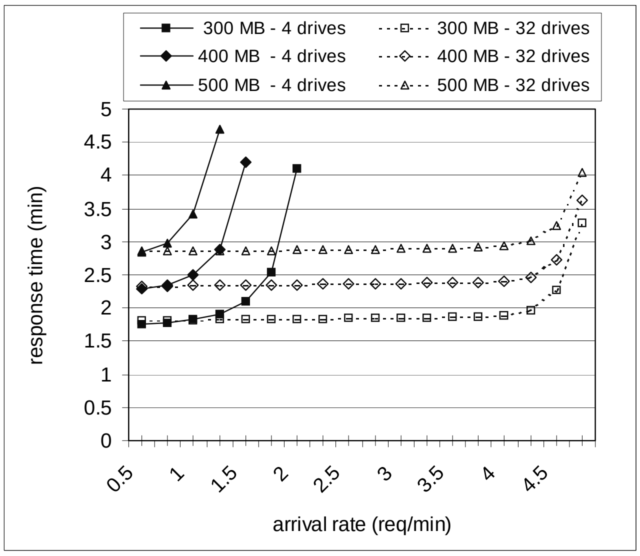
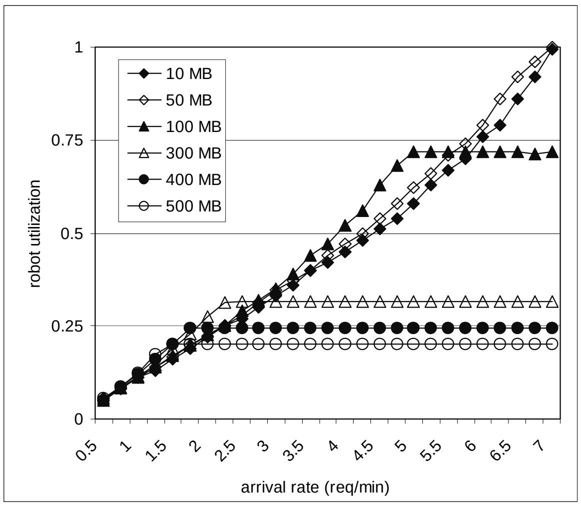
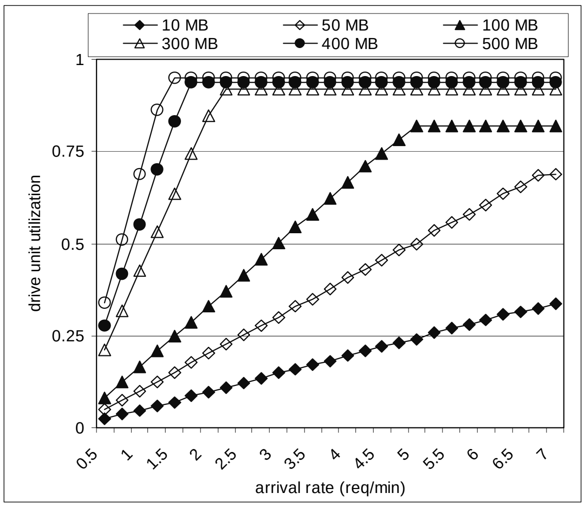


Figure 5-14: Response times as function of the arrival rate for 300, 400 and 500 MB files

When accessing 50 MB files, the 32-drive jukebox serves about 4.75 req/min as maximum (Fig. 5-13), in contrast to the 6.75 req/min. supported by the 4-drive jukebox (Fig. 5-13). The 32-drive jukebox robot arm is significantly slower than the 4-drive jukebox robot arm (i.e. $1/T_{exchange}$= 4.91 req/min of the 32-drive jukebox versus $1/T_{exchange}$= 6.89 req/min of the 4-drive jukebox). On the other hand, thanks to its larger number of drive units, the 32-drive jukebox supports much better accesses to large files, e.g. when accessing 400 MB files a maximal arrival rate of 4.75 req/min is supported (Fig. 5-14).



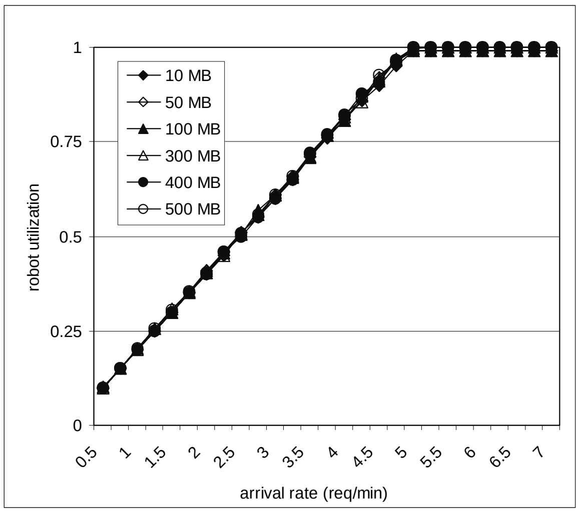
(a)



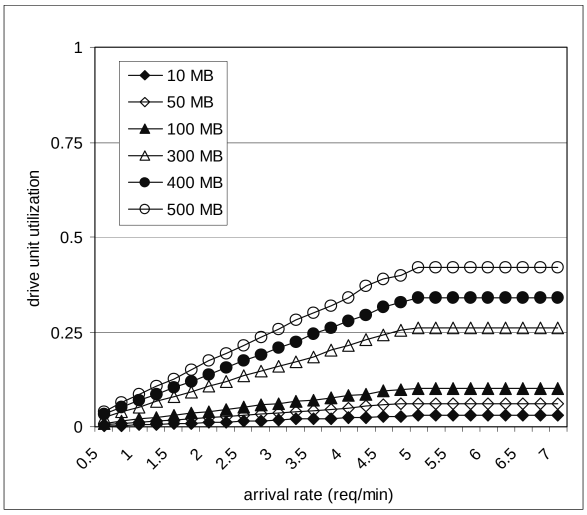
(b)

Figure 5-15: Utilization of (a) the robot arm and of (b) the 4 drive units of a NSM Satellite jukebox as function of the arrival rate (the number of drive units is the limiting factor when drive utilization is higher than 3/4 = 75%)

For all the considered file sizes, the robotic arm device is always the bottleneck in a 32-drive jukebox (Fig. 5-16a). To shift the bottleneck to the number of drive units, we should access files larger than the optimum, i.e. 1.15 GB according to equation 5-2.



(a)



(b)

Figure 5-16: Utilization of (a) the robot arm and of (b) the drive units of a 32-drive jukebox as a function of the arrival rate (the number of drive units is the limiting factor when drive utilization is higher than 31/32 = 96.8%)

5.4.5 File striping analysis

Due to the small number of robot arms and of drives units in optical jukeboxes, it is highly probable that a new request will not find the target optical disk already loaded and will generate a optical disk exchange. The optical disk exchanges are expected to occur frequently and are largely responsible for the high latency experienced by users of optical jukebox servers, e.g. a NSM Satellite jukebox takes over 17 s. to perform a disk exchange. Depending of the file size, the transfer time could also be a significant fraction of the total response time. Fig. 5-17 illustrates an optical disk exchange and a data transfer of a 500 MB file on a NSM Satellite jukebox. The entire procedure requires over 183 s., where 8.7 s. are spent by the robot arm for performing a disk exchange, 8.6 s. for mounting the disk, and 166.6 s. for file transferring considering a drive's throughput of 3 MB/s.

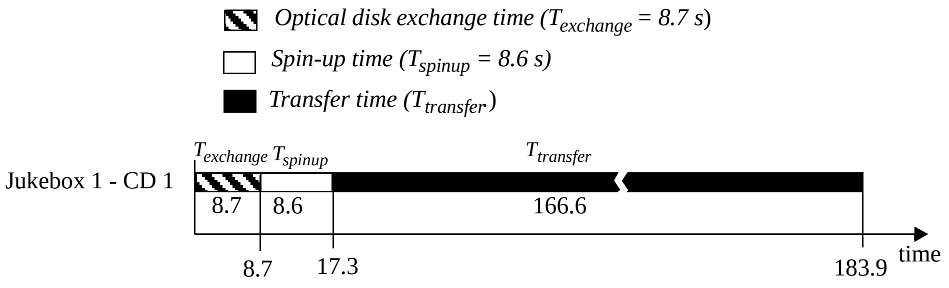

Optical disk exchange time ($T_{exchange}$ = 8.7 s)
Spin-up time (T_{spinup} = 8.6 s)
Transfer time ($T_{transfer}$)
$T_{exchange}$ T_{spinup}
$T_{transfer}$
Jukebox 1 - CD 1
8.7
8.6
166.6
8.7
17.3
183.9
time

Figure 5-17: Timing diagram for retrieving a 500 MB file from a NSM Satellite jukebox

One way to improve the bandwidth of a system is to involve several drives in a single transfer. In other words, if a file is striped across several optical disks, then fractions of it are transferred simultaneously by several drives, thus increasing the effective bandwidth of the transfer. The stripe factor (*SF*) gives the number of optical disks storing a striped file. Fig. 5-18 shows the access time for the same 500 MB file striped across 4 optical disks[1] (i.e. *SF* = 4), 125 MB per disk, on a NSM Satellite jukebox. In this case, the transfer time is reduced to approximately 42 s. Hence, striping effectively increases the bandwidth of the system and reduces the file transfer time.

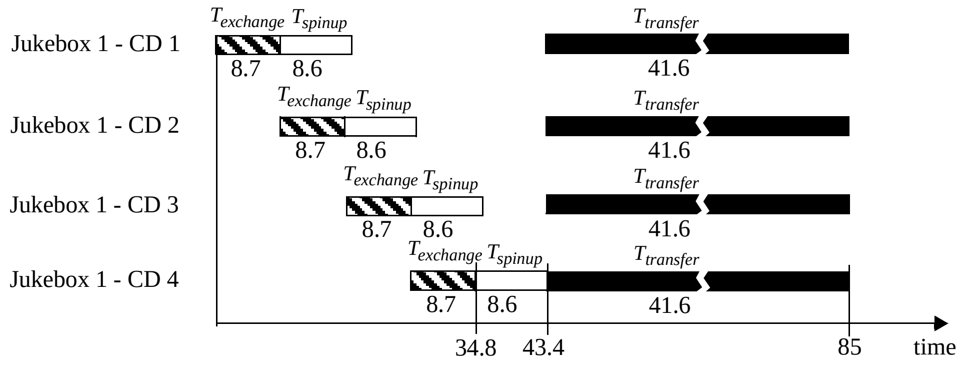


Figure 5-18: Timing diagram for retrieving a 500 MB file striped across 4 CD-ROMs of one NSM Satellite jukebox.

1. All the drives participating in servicing the striped file request read the striped file simultaneously allowing to verify that no errors occurred during the load and transfer operations.

However, striping increases the number of disk exchanges performed per request, which further contributes to the already high disk exchange latency. The latency penalty for striping a file across *SF* optical disks on a single-arm jukebox is of *SF-1* additional disk exchanges. This is illustrated in Fig. 5-18 for an NSM Satellite jukebox retrieving a 500 MB file striped across 4 disks, where exchanging 3 additional disks adds about 26 seconds to the total response time. By transferring the 500 MB file from 4 disks simultaneously, we gain about 125 s. Hence, in this example, the net effect of striping is an improvement of about 99 s. in the total response time. However, for smaller files, or a system with faster transfer rates, this might not be the case. The following equation expresses the response time as a function of the stripe factor (*SF*) for a given file size.

$$T(SF)_{tot} = (SF \cdot T_{exchange}) + T_{spinup} + T_{rot} + T_{seek} + \left(\frac{F_{size}/X_{drive}}{SF}\right) \qquad (5\text{-}11)$$

Fig. 5-19 plots the total response time for different file sizes (10, 100, 200, 300, 400, 500 and 600 MB) as a function of the stripe factor. Increasing the stripe factor, the response time decreases until reaching the minimum response time that corresponds to the optimal stripe factor ($SF_{optimal}$)[1]. Increasing the stripe factor from the optimum, the response time increases linearly with slope of $T_{exchange}$, since the ($SF{*}T_{exchange}$) term becomes in the most significant fraction of the total response time (equation 5-11), whereas the T_{spinup} term remains constant and the (F_{size}/X_{drive})/*SF* term tends to zero.

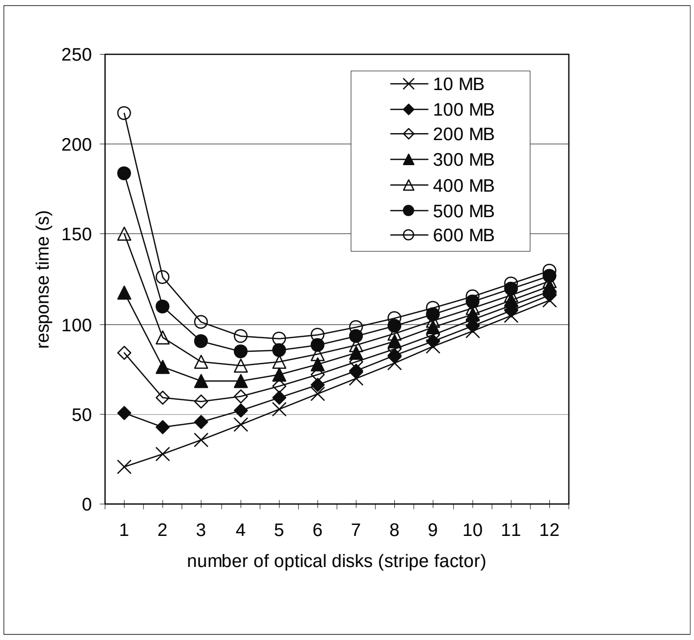


Figure 5-19: Response time as function of the number of optical disks for file sizes varying from 10 to 600 MB.

1. For small files, e.g. 10 MB file, optical disk striping does not reduce response time. In these cases, the optimal stripe factor is equal to 1.

We can determine the optimal stripe factor ($SF_{optimal}$) by minimizing equation 5-11,

$$\frac{dT}{dSF}(SF)_{tot} = 0$$

$$T_{exchange} - \left(\frac{F_{size}/X_{drive}}{SF_{optimal}^{2}}\right) = 0$$

$$SF_{optimal} = round\left(\sqrt{\frac{F_{size}/X_{drive}}{T_{exchange}}}\right)$$

For example, for the 500 MB file size, the optimum stripe factor is 4, which corresponds to a speedup of $T(SF=1)_{tot}$ / $T(SF=4)_{tot}$ = 183.9 / 85 = 2.16 far from the ideal speed up of 4. This difference is due to the latency of exchanging *SF* optical disks sequentially. In order to reduce this latency, jukebox manufacturers try to offer the possibility of exchanging several optical disks per robotic arm movement. For instance, optical jukeboxes produced by ALP Electronics [2] incorporate two robot arms each one comprising four grips. The jukebox may therefore carry four optical disks per robotic arm. Another way to reduce the latency is to consider a storage system comprising several jukeboxes each one with one robot arm. The optical disks involved in the stripe file are distributed across the jukeboxes in order to perform the disk exchanges in parallel. Fig. 5-20 shows the retrieving of a 500 MB file striped across 4 disks each one residing on a different jukebox of a storage system consisting of 4 NSM Satellite jukeboxes. Hence, the net effect of striping across different jukeboxes allows to reach a speed-up of $T(SF=1)_{tot}$ / $T(SF=4)_{tot}$ = 183.9 / 58.9 = 3.12. We do not reach the ideal speed-up of 4 since exchange and mounting times are a significant fraction of the total response time (i.e. a 29.37% of the total response time).

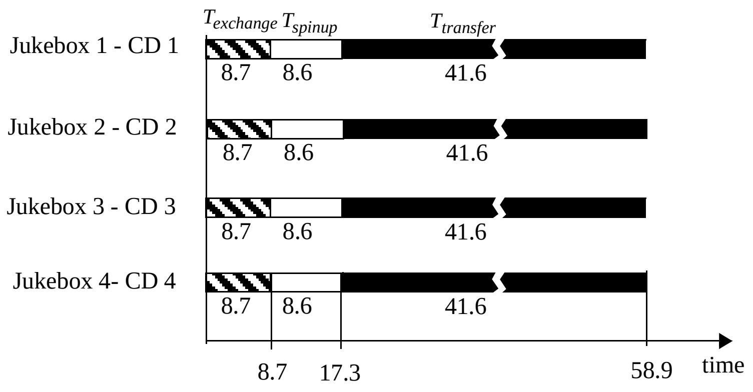


Figure 5-20: Timing diagram for retrieving a 500 MB file striped across 4 NSM Satellite jukeboxes.

Another disadvantage of striping is that it increases the contention for system resources (i.e. for drives as well as for the robot arm). The larger the striping factor, the more contention there is for robot arm and drive resources. Since the system response time increases with contention, under heavy loads it might not be advantageous to stripe even very large files. Under light loads, the effect of an increase in resource utilization is not significant, but under heavy loads it becomes the dominant factor (Figs. 5-15 and 5-16).

For scaling the server's capacity and throughput beyond that of a simple jukebox, one can connect to the server PCs more jukeboxes, up to the limitation of the SCSI interface bus (today 80 or 160 MB/s for Ultra or Ultra-2 SCSI interfaces).

5.5 File caching on magnetic disks

Magnetic disk caching can greatly improve the performances of a terabyte jukebox server. Server nodes connected to the jukebox drives incorporate dedicated disks for magnetic disk caching. Performance is enhanced due (1) to prefetching of files and faster liberation of optical disk drives and (2) due to the locality of file accesses.

The magnetic disk caching strategy implemented by the terabyte server consists in transferring to magnetic disk cache each file opened by a client if:
- the file size is of reasonable size, i.e if it does not occupy more than a few percents of the magnetic disk cache space,
- the file size is large, but fits within the available cache space and current statistics show that it is frequently accessed (multiple requests to the same file).

In the context of removable storage, complete files need to be cached on magnetic disks in order to remove the optical disk and free the corresponding optical drive unit.

If a data file to be accessed should be cached, but does not yet reside in magnetic disk cache, we immediately serve the client request and at the same time write the corresponding file part into the magnetic disk cache. In a second step, the remaining parts of the file are also transferred to the magnetic disk cache.

If we assume that the data throughput (X_{user}) required by the client is lower than the optical drive throughput (X_{drive}), after a short time, further read requests to the same data file may be served by simply reading from magnetic disk cache. For read requests to consecutive locations of the same data file, the predominant access pattern consists in simultaneously reading from the optical disk, writing onto the magnetic disk cache and reading from the magnetic disk cache to serve the client request. If the magnetic disk throughput is not at least twice as high as the optical disk throughput[1], only a certain fraction C_{factor} of the optical drive throughput is available to serve client requests.

For example, when reading sequentially a large file (e.g. 10 MB read in chunks of 128 KB) directly from a PX32CS CD-ROM drive, i.e. without magnetic disk caching, we reach a disk

1. We assume that files are stored at contiguous disk locations and that head displacements to reach different locations within a single file are negligible.

throughput[1] of 3 MB/sec. When reading the same data through the magnetic disk[2] cache while feeding it with new data from the optical disk drive, we obtain a disk throughput of 2.6 MB/sec., i.e a fraction C_{factor} = 85.3% of the maximum optical drive throughput.

In order to quickly liberate an optical disk drive, it makes sense to cache an optical disk file even if it is accessed only once under the condition that the required client throughput is significantly lower than the optical drive throughput.

The drive occupancy $U_{occupancy}$ can be expressed as a function of the caching throughput factor C_{factor}, the optical drive throughput X_{drive} and the user access throughput X_{user}.

$$U_{occupancy} = \frac{X_{user}}{X_{drive} \cdot C_{factor}}$$

For example with a drive occupancy $U_{occupancy}$ = 50%, magnetic disk caching allows to serve up to twice the amount of data that would be served without disk caching[3].

5.6 The shadow directory tree

The terabyte server maintains on a magnetic disk a *shadow directory tree* containing the subdirectories and file names of each of the optical disks located on the magazine slots of the server jukeboxes. The shadow directory tree stores the location of each optical disk within the server. The optical disk location is determined by the jukebox index and the magazine slot index within the jukebox. The shadow directory tree is created in the server initialization phase, and when a new optical disk is inserted into the server or dynamically created by a writable unit, the content of the disk is read and the shadow directory tree is updated. Similarly, subdirectories are removed from the shadow directory tree when an optical disk is removed from the server. Thanks to the shadow directory tree, the terabyte server can serve directory listing requests without having to load the corresponding optical disk into the drive unit.

When opening a file, the client sends a *open_file* request to the server. The request concerns a shadowed file (e.g. c:/shadowtree/images_1/logos/logo32.bmp). The file path is formed by the shadow tree root path (e.g. c:/shadowtree/), followed by the optical disk label[4] (e.g. images_1)

1. The disk throughput is the continuous data throughput captured by the disk head (i.e. without taking into account disk spin up, head displacement and disk rotation time to reach the desired data sector).
2. The magnetic disk we used has a measured latency (seek+rotation time) of 12.1 ms and a measured disk throughput of 5.5 MB/s.
3. This is an asymptotic figure for very large data files. For smaller file sizes, disk exchange, spin-up and seek times need to be taken into account
4. With optical disk labels, there may be optical disks without label or several optical disks with the same label. Instead of an optical disk label, a unique optical disk identifier can be used.

and the file path within the optical disk (/logos/logo32.bmp). From the shadow file path the server reads the optical disk label. The terabyte server maintains a table of labels of all optical disks. Each entry in the table indicates basically (1) the corresponding jukebox index, (2) the slot index within the jukebox magazine and (3) the disk status (i.e. magazine slot, loaded in drive, loading into drive or unloading from drive).

5.7 Software architecture

Since small jukeboxes (Fig. 5-1) have at least 4 optical disk reading units and large jukeboxes (Fig. 5-2) may have several dozens of reading units, applications requiring high data throughputs may have to access simultaneously data striped over multiple optical disks. In order to provide services for high-throughput parallel applications and for continuous media support, the terabyte server is based on the library of striped file components described in Chapter 2 and on the parallel stream server library described in Chapter 3. Thanks to these libraries and by using the CAP Computer-Aided Parallelization tool (Chapter 2), the terabyte server can execute pipelined parallel data access and processing operations, where processing operations are executed on the same server nodes as the corresponding data access operations.

By incorporating the library of striped file components and the parallel stream server library, the terabyte server offers a set of threads with appropriate operations for allowing to access large files striped over multiple optical disks loaded into different optical disk drives and for providing continuous media support for streaming data from the server's jukebox to the client. Application programmers are free to incorporate specific operations to the existing threads and to combine them with the predefined operations to implement their parallel continuous media server running on top of the scalable jukebox server architecture.

The scalable terabyte server is composed by the following threads:

- The *InterfaceServer* thread (striped file library) maintains on a magnetic disk a *shadow directory tree* containing the directory trees of all present optical disk files in the server jukeboxes.
- One or several *ExtentFileServer* threads (striped file library) run on the server nodes and are responsible for opening and closing optical disk files.
- One or several *ExtentServer* threads (striped file library) run on the server nodes and access the optical disk for reading data files.
- One *StreamTimerServer* thread (parallel stream server library) runs on each server node and is responsible for the isochrone behavior of the server node.
- Several *ComputeServer* (striped file library) and *HPComputerServer* (parallel stream server library) threads run in each server node. Application developers are free to add specific processing operations running on the *ComputeServer* and *HPComputeServer* threads.

- One or several *ServerClient* threads (parallel stream server library).
- A *RobotServer* thread runs on the server nodes for controlling the jukebox robotic arm.

Fig. 5-21 shows how the terabyte server threads are mapped onto a scalable jukebox server architecture comprising 3 server nodes. When accessing a file stored in an optical disk, an application thread sends a *open_file* request to the *InterfaceServer* thread. By looking up within its internal structures or within its *shadow directory tree*, the *InterfaceServer* thread identifies the optical disk location, i.e. the jukebox and the magazine slot within the jukebox. If the disk resides in the magazine slot and there are available drive units, the *InterfaceServer* thread sends a *load_disk* request to the corresponding *RobotServer* thread. Once loaded, the *InterfaceServer* thread sends an *open_disk_file* request to the corresponding *ExtentFileServer* thread which opens the file from the disk and sends back the disk file descriptor. The *InterfaceServer* thread returns the server node index and the disk file descriptor to the client. For subsequent reading of data, an application thread sends *read_disk_file* requests directly to the corresponding *Extent-Server* threads which read from the optical disk and send the data back. There is no further unnecessary communication with the *InterfaceServer* thread. The application threads commu-

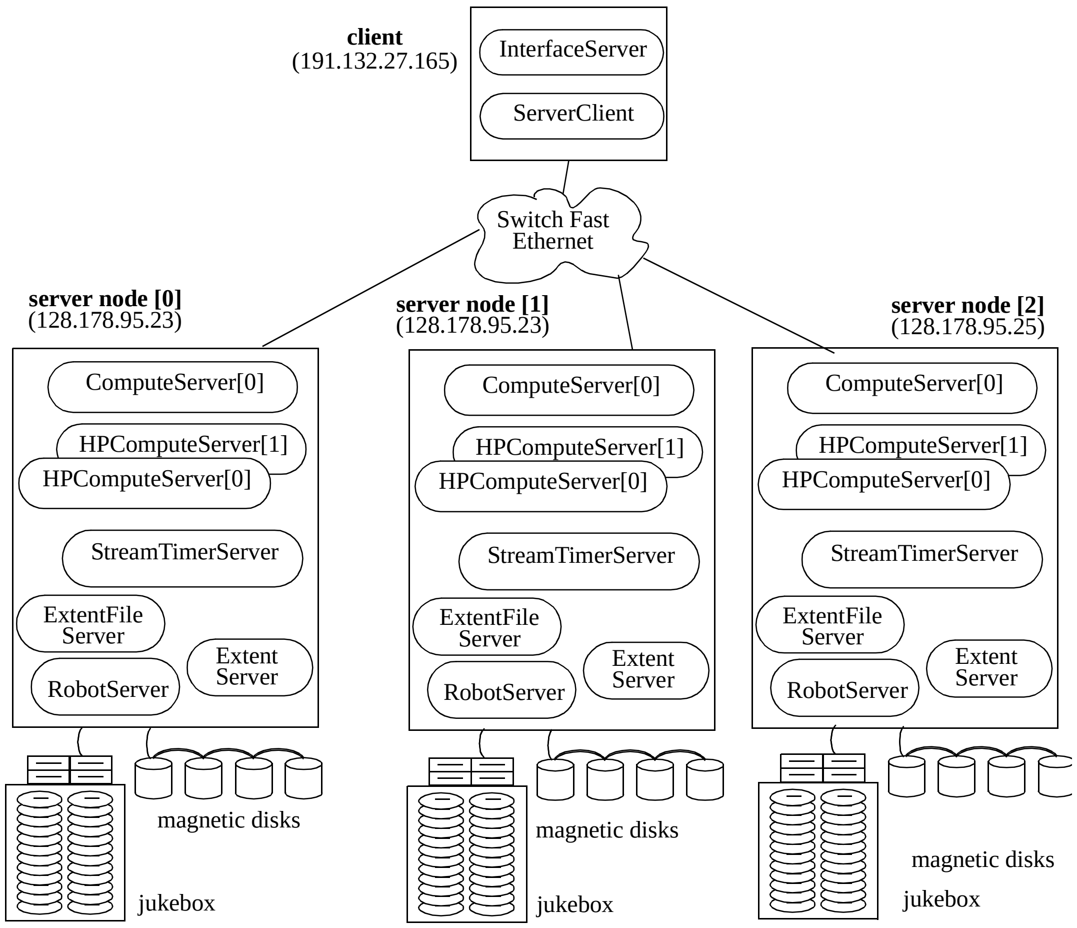


Figure 5-21: Terabyte server threads and how they are mapped onto a scalable jukebox server architecture

nicate with the *InterfaceServer* thread for the open/close operations. The *InterfaceServer* thread communicates with the *ExtentFileServer* threads to execute these open/close operations on the optical disks. In order to maintain the consistency between server nodes, an application thread never communicates directly with the *ExtentFileServer* thread, for example for opening a disk file. On the other hand, application threads communicate directly with the *ExtentServer* threads, e.g. for reading data from disk files.

Two configuration files are required to run the terabyte server. One specifies the mapping of threads to Windows NT processes running in the server and one describes the terabyte server architecture, i.e. it specifies the jukeboxes and the magnetic cache disks in the server (Annex A).

5.7.1 The *InterfaceServer* thread

The *InterfaceServer* thread serves directory listing, file opening and closing requests from users and is responsible for shadow tree management.

The *InterfaceServer* thread maintains a LRU table of cached files on magnetic disks. When requesting a file, the *InterfaceServer* thread first searches this table for the required file. If no entry associated to that file is found, the file is not cached and must be loaded into the drive units. A table contains the file access statistics for each accessed file. This table allows to determine if a file should be cached on magnetic disk. Since the terabyte server incorporates several magnetic disks for file caching, the *InterfaceServer* thread maintains a magnetic cache disk table, each entry specifying the path root of the magnetic disk (e.g. e:/), its capacity (e.g. 4096 MB) and its free space for caching files (e.g. 1208 MB). The *InterfaceServer* thread keeps track of the cached files and maintains file access statistics in order to decide which files should be swapped out of magnetic disk cache.

After a file opening request, the *InterfaceServer* thread identifies, thanks to the shadow directory tree, the target optical disk, its slot location and the jukebox on which it resides. If the requested file is not cached on magnetic disks, it checks for available drive units. If drive units are available, it selects one of them to load the optical disk. A *load_CD-ROM* request is sent to the *RobotServer* thread which issues a command to the robotic device to move the optical disk from its magazine slot to the selected drive unit. After loading the disk, the *InterfaceServer* thread sends a file open request to the corresponding *ExtentFileServer* thread who opens the file from the optical disk. The *InterfaceServer* thread also decides if the requested file has to be cached on magnetic disk. In that case, it sends a cache file request to the corresponding *ExtentServer* who starts to transfer the file from the optical disk to the magnetic disk cache.

An optical disk may be unloaded when the requested file has been completely transferred to magnetic disk cache, when a file close command has been issued, or when an uncached opened file has not been accessed during a certain period of time.

As mentioned, the *InterfaceServer* thread coordinates both the parallel file operations (Chapter 2) and the parallel stream operations (Chapter 3). In order to support a continuous media service, the *InterfaceServer* thread performs the admission control algorithm and the resource reservation thus ensuring that a new stream request does not violate the real-time requirements of streams already being serviced (Chapter 3).

5.7.2 The *RobotServer* thread

The *RobotServer* thread serves load and unload requests from the *InterfaceServer* thread. The *RobotServer* thread issues commands to the jukebox robotic arm in order to load or unload the drive units. These commands are non-blocking, i.e. several commands may be issued simultaneously for example for activating at once two robotic devices (e.g. for the jukebox of Fig. 5-2). In order to control several different jukebox models by the same *RobotServer* thread, support for different command protocols defined by the jukebox manufacturers [48] is provided.

5.7.3 The *ExtentFileServer* thread

The *ExtentFileServer* thread serves directory listing, file opening and closing requests from the *InterfaceServer* thread. To create and delete cache files on magnetic disks, the *ExtentFileServer* thread serves the cache create and cache delete file requests from the *InterfaceServer* thread.

5.7.4 The *ExtentServer* thread

ExtentServer threads perform asynchronous I/O operations in order to access files stored on optical and magnetic disks. With asynchronous I/O operations, one *ExtentServer* thread can manage multiple drive units and multiple magnetic cache disks. *ExtentServer* threads serve both the read file requests from users and the cache file requests from the *InterfaceServer* thread.

For supporting continuous media access, there are three different types of file requests (1) normal read requests from clients, (2) real time read requests from clients accessing continuous media streams and (3) file caching requests launched by the interface server thread. These different types of requests are served separately, by having one queue for each request type. The highest priority queue is the real time request queue, the mid-priority queue is the normal read request queue and the lowest priority queue is the caching request queue. The normal read requests and caching read requests are served following a first-in first-out policy. The continuous media read requests are scheduled on disks[1] according to the earliest deadline first and to the incremental disk track scanning (SCAN-EDF) method (see section 3.9).

5.8 Summary

This chapter proposes a scalable jukebox server architecture in order to offer access to large volumes of data at low cost. A analytical model has been created to analyze the potential

1. Regarding disk scheduling of continuous media read requests, in order to avoid frequent head displacements, we assume that only one data stream is served from each optical disk.

bottlenecks of a jukebox (i.e. the number of drive units and of robotic arm devices) and to calculate its overall throughput and its service rate. The model takes into account the robot arm and drive unit performances as well as the client request arrival rate. The effect of striping a file across different drive units of a single jukebox as well as across several jukeboxes is analyzed. For scaling the server's capacity and throughput beyond that of a simple jukebox, we can connect to the server PC additional jukeboxes or add additional PCs each one with one or several jukeboxes.

The terabyte server comprises one master server PC running the server interface receiving client access requests and additional server PCs connected each one to one or several jukeboxes. The master PC manages the shadow file directory containing the directory of all files residing on the jukeboxes. It is also responsible for resource allocation and for file opening operations. Slave server PCs running the *ExtentServer* threads are responsible for accessing files located on their jukeboxes as well as for caching optical disk files on magnetic disks. At low client access rates, caching of complete files on magnetic disks allows to quickly liberate optical disk drives and to serve more clients. The terabyte server allows to access large files striped across multiple optical disks. Parallel access to optical disks located each one in a different drive may offer new opportunities for high-quality continuous media editing and processing applications. Chapter 6 presents a WEB server based on the scalable jukebox server architecture and discusses scalability and performance issues.

6 A Scalable Terabyte WEB Server

6.1 Introduction

This chapter presents a WEB server based on the scalable terabyte optical jukebox architecture described in Chapter 5. The World Wide Web is characterized by multitudes of clients and by servers whose processing power, I/O bandwidth and storage capabilities must scale both with application requirements and the number of clients accessing simultaneously the server.

We are interested in the performances and the scalability of the teraserver jukebox architecture, especially for parallel server applications making use of several server resources, such as multiple server PCs and multiple optical disk drive units. We test the scalability of the terabyte WEB server by running the *Visible Human server* and the *4D beating heart slice stream server* applications on top of the jukebox terabyte server. These server applications access simultaneously their respective data sets striped over either 2, 4, 6 or 8 CD-ROMs.

In the terabyte server, the storage structure is viewed as a single file system tree. Early CD-ROM-based storage solutions present each individual optical disk as a subdirectory of the root of the logical volume. This method has the drawback of requiring applications to maintain the physical location of all data. A preferred method used by the most recent optical storage solutions [48] combines the directory structures of all optical disks into a single common directory structure, i.e. the shadow directory tree. This method has the major advantage of presenting all data in a common directory structure, whether there is one optical disk or thousand disks, thus eliminating the need for users or applications to know the physical location of data. This makes application development much simpler. File systems can have all the functionality and control that the host operating system offers. The Jukeman software [48] implements a native file system for NT clients which appears as a drive letter and can be accessed and shared easily using standard file system libraries and utilities (e.g. Windows Explorer). Since developing an NT native file system [64, 19] can be a difficult task, there is also the possibility to develop a file system filter [75, 88, 66], i.e. an intermediate driver that intercepts and processes I/O request for an underlying file system. Instead of developing a file system or a file system filter specific to a given operating system, we interface the jukebox terabyte server to a WEB server in order to offer its services to Internet clients. A WEB server has the advantage that it is independent of the operating system running at the client side. In addition, a WEB server can, thanks to ISAPI [36] or CGI [51] server interfacing technologies, offer access to advanced server applications such as the *4D Beating Heart slice stream server* described in Chapter 4.

6.2 The terabyte WEB server architecture

The jukebox server architecture we consider comprises 3 server units interconnected by a 100 Mbits/s Fast Ethernet crossbar switch. One server unit (i.e. the master PC) comprises one server PC (Bi-Pentium-III 733MHz) running the WEB server on a Windows 2000 operating system. The master PC incorporates a magnetic disk to store the shadow directory tree. Each one of the other server units (i.e. the slave PCs) comprises one PC (Bi-Pentium-II 333MHz) connected to one NSM Satellite CD-ROM jukebox incorporating 4 read-only optical disk drive units and a magnetic disk for caching purposes. The slave PCs run server processes on a Windows NT Workstation 4.0 operating system. This configuration is easily extended, e.g. by adding a second jukebox per slave PC as well as additional server units made of 1 slave PC and two jukeboxes.

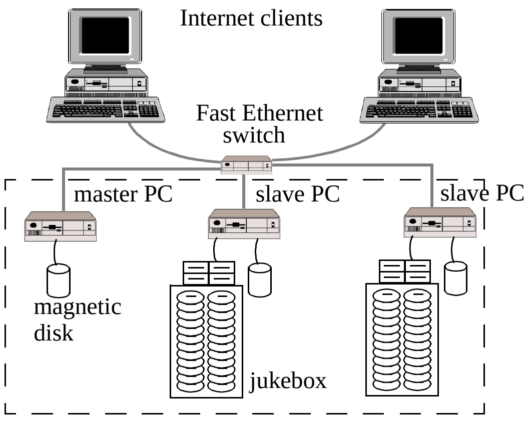


Figure 6-1: Terabyte WEB server architecture

The jukebox terabyte server is interfaced to a WEB server using the CGI technology [51] as shown in Fig. 6-2. An Internet client running an Internet browser asks for a certain terabyte server's service (e.g. visualization of an image) by sending an HTTP request to the WEB server located in the master PC. The HTTP request contains the service code (e.g. to read a file) and its corresponding parameters (e.g. the file name). The WEB server responds to the HTTP request by creating a CGI process that receives the request[1]. In Fig. 6-2, the WEB server created CGI process 1 for responding to the HTTP request coming from client 1, and CGI process 2 for the client 2. The CGI process communicates with the jukebox terabyte server through sockets. The CGI process sends the request to the terabyte server interface thread that resides in the master PC. In collaboration with the slave PCs, the master PC performs the requested service and sends back the result to the corresponding CGI process which sends it to the WEB server process. Finally the WEB server process sends back the result to the client. In the next section, we describe three advanced services offered by the jukebox terabyte server. They are extraction of slices and of MPEG-1 animations comprising a sequence of slices from the Visible Human head data set. The third service runs the 4D beating heart slice stream server application (described

1. In order to minimize memory consumption and process creation time, each created CGI process requires only 59 KB of memory. The terabyte server application requires 2704 KB of memory.

in Chapter 4). It allows users to stream slices from the 4D beating heart data set through the WEB.

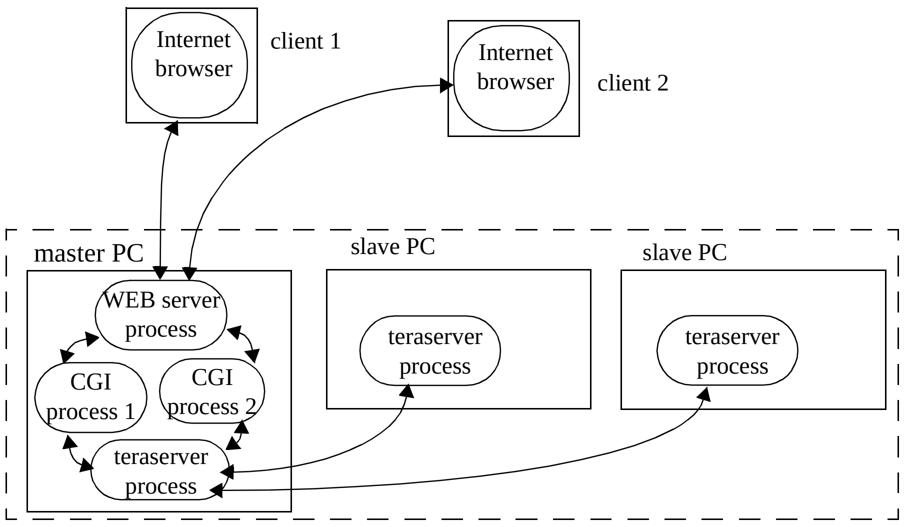


Figure 6-2: Terabyte WEB server software architecture

6.3 The Visible Human Server

Web-based slicing services for extracting slices from the Visible Human data set exist since 1995, but they authorize only to extract slices perpendicular to the main axes. Since June 1998, EPFL’s Visible Human Slice Server [45] allows to extract arbitrarily oriented and positioned slices. More than 200,000 slices are extracted per year. A surface extracting service has been added and allows to define, extract and flatten curved surfaces. But extracted slices and surfaces are only 2D images which do not reveal the full 3D anatomic structures. To give an impression of 3D, a service allowing to extract successive slices along a user-defined trajectory has been created [7]. It is a kind of video on demand service (VOD), where Web clients specify the trajectory of the desired slice sequence animation across the body of the Visible Human. The resulting animation is streamed to the client as an MPEG-1 video.

The publicly accessible version of the Visible Human Slice Web Server and the Visible Human Slice Sequence Animation Server run each one on a single Bi-Pentium-II PC connected to 16 magnetic disks and offer their interactive services at *http://visiblehuman.epfl.ch* (for slicing services) and at *http://visiblehuman.epfl.ch/animation/* (for slice sequence animation services).

Thanks to the CAP framework, we have easily integrated the Visible Human Slice Server and the Visible Human Slice Sequence Animation Server in the Terabyte WEB Server. We tested the scalability of the Terabyte WEB Server by extracting slices and slice sequence animations from the Visible Human Head data set striped over either 2, 4, 6 or 8 CD-ROMs. For enabling parallel access to the Visible Human Head data set, it is segmented into sub-volumes (extents) of size 54x54x19 RGB voxels, i.e. 162.3 KB, which are striped over a number of CD-ROMs.

In order to make fair comparisons, i.e. to have similar disk throughputs for the different experiments, each Visible Human Head subfile is written at the center of its respective CD-ROM.

6.3.1 The Visible Human slice server application

The Visible Human slice web server application consists in a server interface residing on the Web server PC and server processes running on the server's slave PCs.

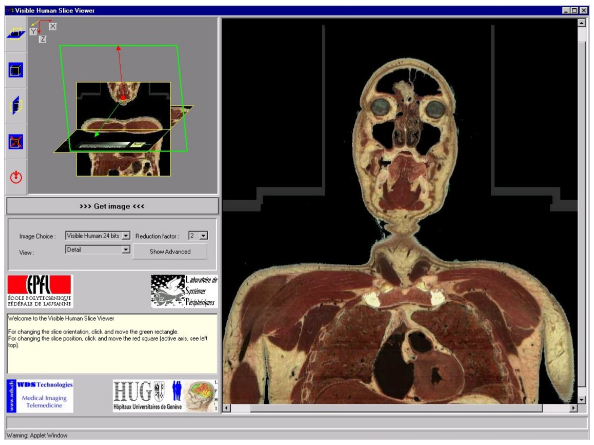

Visible Human Slice Viewer
Y
X
Z
>>> Get image <<<
Show Advanced
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Laboratoire de Systèmes Périphériques
WDS Technologies
www.wds.ch
Medical Imaging Telemedicine
HUG
Hôpitaux Universitaires de Genève

Image Choice : Visible Human 24 bits
Reduction factor : 2
View : Detail

Welcome to the Visible Human Slice Viewer

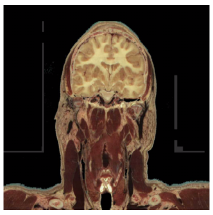
For changing the slice orientation, click and move the green rectangle.
For changing the slice position, click and move the red square (active axis, see left top).

Warning: Applet Window

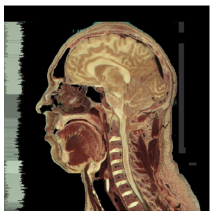
Figure 6-3: Selecting within a Java applet an image slice

When the server interface receives a slice extraction request, it verifies the drive units' availability. If drive units are available, the server interface sends to the slave PCs requests to load the optical disks into the corresponding drives assuming that the optical disks containing the Visible Human Head data set reside in the jukebox's magazines. Once the optical disks are loaded, the server interface interprets the slice location and the orientation parameters defined interactively by the user (Fig. 6-3) and determines the volumic extents (sub-volumes) which need to be accessed. It sends the extent reading and image slice part extraction requests to the concerned servers (servers whose optical disks contain the required extents). These servers execute the requests and transfer the resulting slice parts to the Web server PC which assembles them into the final displayable image slice. This final image is compressed in JPEG format and sent to the corresponding WEB client.

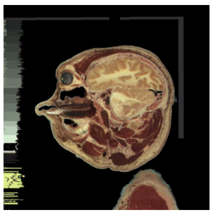
To measure the Visible Human slice server application performances, the experiment consists of extracting 1024x1024 24-bit/pixel slices. Since the slice extraction time depends mainly on the slice position and orientation, we have selected three slices with different orientations. One slice parallel to the XZ plane (refered as the coronal slice), one slice parallel to the YZ plane (the sagittal slice) and one slice whose orientation is orthogonal to one of the diagonals traversing the Visible Human Head's rectilinear volume (the diagonal slice). The parameters of the extracted slices are shown in Fig. 6-4. The *center* indicates the enter of the slice within the coordinate system of the Visible Human head. The *normal* vector gives the slice orientation. The *up* vector specifies the 3D orientation of the vertical axis of the extracted slice. Extracting the coronal, sagittal and diagonal slices requires to read respectively 252, 291 and 380 extents, i.e. 252x164 KB = 40.35, 46.60 and 60.85 MB of data from the optical disks.



coronal slice
height, width: 1024, 1024
normal (x.y.z): 0, -1000, 0
center (x,y,z): 1024, 608, 432
up (x,y,z) : 0, 0, -1000
# extents : 252 (40.35 MB)



sagittal slice
height, width: 1024, 1024
normal (x.y.z): -1000, 0, 0
center (x,y,z): 1024, 608, 432
up (x,y,z) : 0, 0, -1000
# extents : 291 (46.60 MB)



diagonal slice
height, width: 1024, 1024
normal (x.y.z): -707, -3, 707
center (x,y,z): 1024, 608, 432
up (x,y,z) : -707,-30, -707
# extents : 380 (60.85 MB)

Figure 6-4: Extracted slice parameters

We measure first the scalability of parallel slice extraction assuming that the optical disks are already loaded in their drives, i.e. without disk loading and disk spinup times. Fig. 6-5 shows the slice extraction times and the corresponding speedup as a function of the number of CD-ROMs on which the Visible Human Head data set is striped for the coronal, sagittal and diagonal slices (Fig. 6-4).

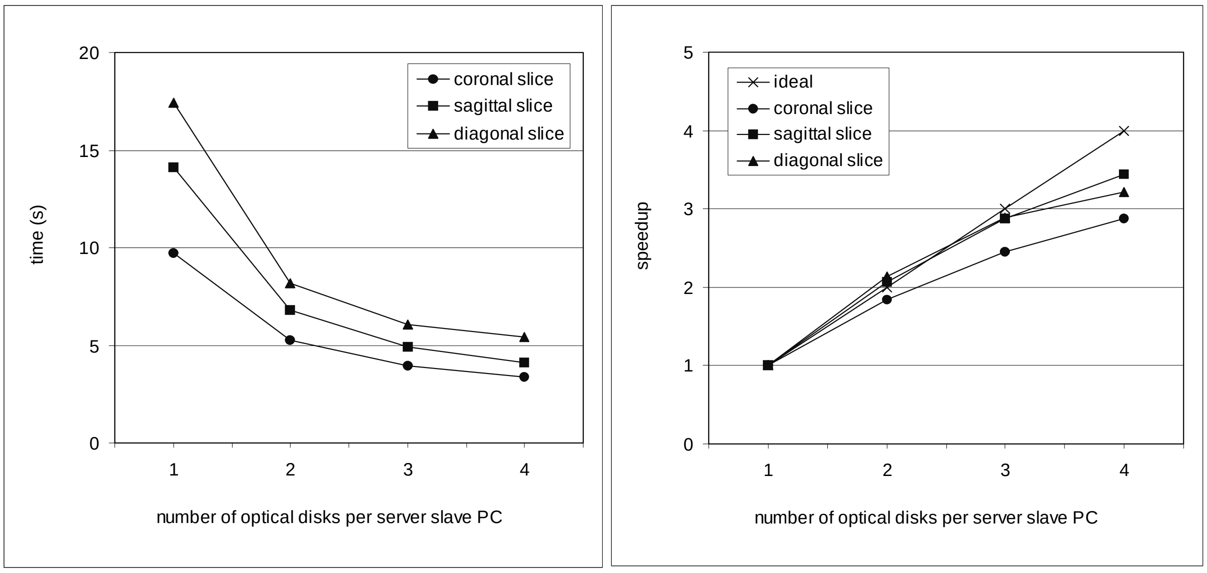


Figure 6-5: Slice extraction times (without taking into account the disk load and spinup times) and speed-up as a function of the number of simultaneously accessed CD-ROMs. The server configuration comprises one master PC and two slave PCs. Each slave PC is connected to one NSM Satellite jukebox

For the considered slices, extraction times scale quite well when increasing the stripe factor, i.e. the number of simultaneously accessed CD-ROMs. When extending the number of simultaneously accessed CD-ROMs from 2 to 8, we obtain a speedup of 2.87, 3.21 and 3.44 for coronal, sagittal and diagonal slices respectively (Fig. 6-5). However, for a smaller slice or for a different slice orientation, this might not be the case. We don't reach an ideal speedup (i.e. a speedup of 4 in the case of 8 CD-ROMs) primarily due to the fact that the extents to be accessed are located at different positions on the optical disk depending on how many CD-ROMs the Visible Human

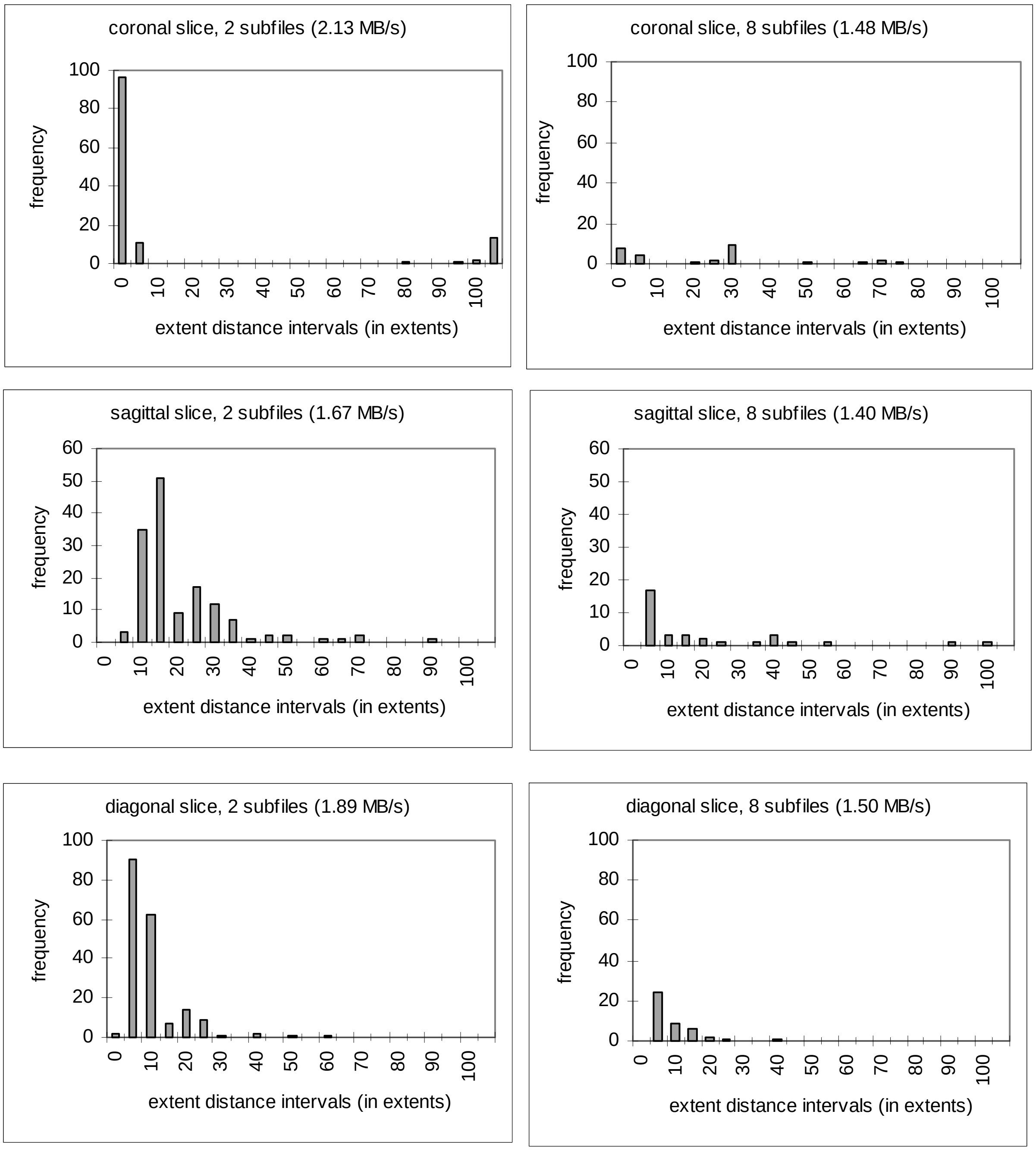


Figure 6-6: Extent distance interval histograms when striping over 2 and 8 subfiles located each one on a different CD-ROM for coronal, sagittal and diagonal slices

head is striped. The extent distribution on the striped file affects the optical disk throughput since head displacements reduce the effective optical disk throughput. Fig. 6-6 shows the extent distance interval histograms for subfiles belonging to a file striped on 2 and on 8 subfiles, when accessing a slice. Fig. 6-6 also shows the optical disk throughput obtained for each extent distribution. The maximal optical disk throughput is measured when extracting the coronal slice from 2 CD-ROMs, i.e. 2.13 MB/s. Most accessed extents are located at contiguous positions on the subfile, i.e. the first interval is predominant in the histogram. By striping across 8 CD-ROMs, the individual optical disk throughput drops by 30.51% to 1.48 MB/s. Due to this significant difference of throughput, we obtain the worst speedup when extracting the coronal slice (Fig. 6-5). For the diagonal slice, the optical disk throughput drops to 20.6%, from 1.89 MB/s (when extracting the slice from 2 CD-ROMs) to 1.50 MB/s (when extracting from 8 CD-ROMs). And for the sagittal slice, the throughput drops by 17.15% from 1.69 MB/s to 1.40 MB/s. The smaller the throughput drop, the higher the speedup. In order to reach the ideal speedup, there should be no throughput drop.

Fig. 6-7 plots the throughput for reading extents of the slowest optical disks involved in the stripe file as a function of the number of optical disks (subfiles) accessed simultaneously. For a given slice, the slowest optical disk involved in the striped file determines the slice extraction time. For example, when extracting the diagonal slice from 8 CD-ROMs, the slowest optical disk reads 49 extents at 1.501 MB/s, i.e. it takes 5.22 s, close to the diagonal slice extraction time of 5.34 s. When extracting the diagonal slice from 2 CD-ROMs, the slowest optical disk reaches a throughput of 1.877 MB/s. Considering the minimal throughputs, we have an I/O bandwidth of 2 x 1.877 = 3.754 MB/s when extracting slices from 2 CD-ROMs and 8 x 1.501 = 12.008 MB/s from 8 CD-ROMs. The resulting speedup is of 12.008 / 3.754 = 3.19, that is, close to the speedup of 3.21 for the diagonal slice extraction. When extracting the sagittal and diagonal slices from 4 CD-ROMs, the I/O bandwidth per optical disk is higher than when

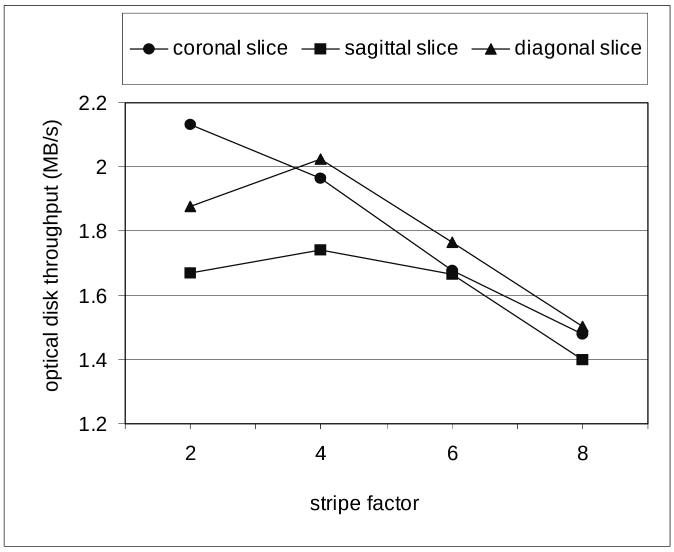


Figure 6-7: Throughput of the slowest optical disk involved in the striped file as a function of the stripe factor for the considered slices

extracting them from 2 CD-ROMs (Fig. 6-7). This explains that in these cases the measured speedup, i.e. a speedup of 2.06 for the sagittal slice and of 2.13 for the diagonal slice are higher than the ideal speedup of 2 (Fig. 6-5).

This experiment shows that I/O bandwidth (i.e. the limited number of drive units) is the bottleneck. Therefore increasing either the number of drive units per server slave PC (e.g. by incorporating a second jukebox per slave PC) or the number of server slave PCs (assuming each PC incorporates an equal number of jukebox) increasses the number of drive units and offers a higher I/O bandwidth. Server processors and network bandwidth are underutilizated. When accessing slices striped across 8 CD-ROMs, the master PC is utilizated at about 10%, the slave PCs are utilized at about 20%. A network throughput of 1.01 MB/s is reached, far of the 12.5 MB/s maximum throughput sustained by Fast Ethernet.

We measured slice extraction times, assuming that the drive units are busy and therefore optical disk exchanges need to be performed to load the corresponding optical disks. Fig. 6-8 shows the diagonal slice extraction times as a function of the number of CD-ROM accessed simultaneously, taking into account the optical disk exchange and spinup times. For the diagonal slice, the minimum extraction time is reached when extracting the slice from 4 CD-ROMs.

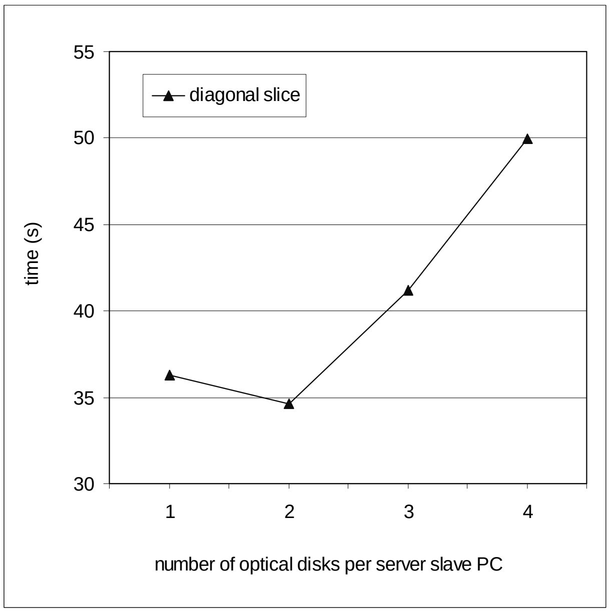


Figure 6-8: Diagonal slice extraction times as a function of the number of CD-ROMs accessed simultaneously for a server configuration comprising one master PC and two slave PCs

Fig. 6-9 shows the timing diagram for extracting the diagonal slice from 4 CD-ROMs distributed across two NSM Satellite jukeboxes. The diagonal slice extraction time is 34.1 s, where 2 x 8.7 = 17.4 s are spent by each robot arm for performing two disk exchanges, 8.6 s for the spin-up time of the last loaded disk, and 8.1 s for reading in parallel the file data.

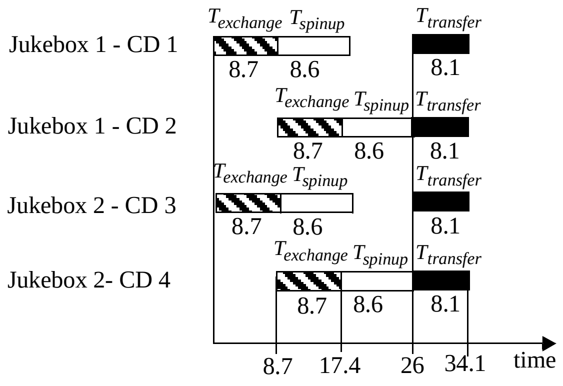


Figure 6-9: Timing diagram for extracting the diagonal slice from 4 CD-ROMs distributed across two NSM Satellite jukeboxes

Regarding the number of CD-ROMs for striping the Visible Human head data set, a trade off between the slice extraction response time, the service rate able to be sustained once the corresponding optical disks are loaded and contention for the drive units needs to be found. As shown in Fig. 6-8, we reach the minimum extraction time with 4 CD-ROMs for the diagonal slice. However, in case of heavy slice access load, we may stripe on more CD-ROMs in order to serve at a higher rate once the optical disks are loaded. With 4 CD-ROMs, the system serves 1/8.2 = 0.012 slices/s = 7.31 slice/min. With 8 CD-ROMs, a service rate of 1/5.4 = 0.185 slice/s = 11.11 slice/min can be sustained. On the other hand, using 8 CD-ROMs to serve Visible Human slice requests monopolizes all the server's drive units. Requests asking for a different service (e.g. a 4D beating heart slice stream extraction) could not be served.

6.3.2 The Visible Human slice sequence animation server application

The slice animation server [7] provides a 3D view of anatomic structures by showing animations made of many successive slices along the user-defined trajectory. To extract a slice sequence animation, an Internet client defines the animation parameters (i.e.trajectory control points, distance between slices, animation size...) using the Visible Human's java applet (Fig. 6-3). When the server interface receives the animation request, the request is converted into a list of slices orthogonal to the trajectory. This slice list is split into individual slice requests. Each slice is extracted in parallel by the slave PCs as previously described (section 6.3.1). Once a slice is extracted, the interface server writes the slice into a slice buffer. This sequence of operations is realized in pipeline for all slice requests. At the same time as the pipelined parallel slice extraction is carried out, an MPEG-1 encoder thread that runs on the master PC gets the extracted slices from the slice buffer, compresses them into MPEG-1 format and sends the resulting animation part to the Internet client. The MPEG-1 encoder works in streaming mode, i.e. it repeatedly compresses a small set of slices and sends the resulting animation part to the Internet client.

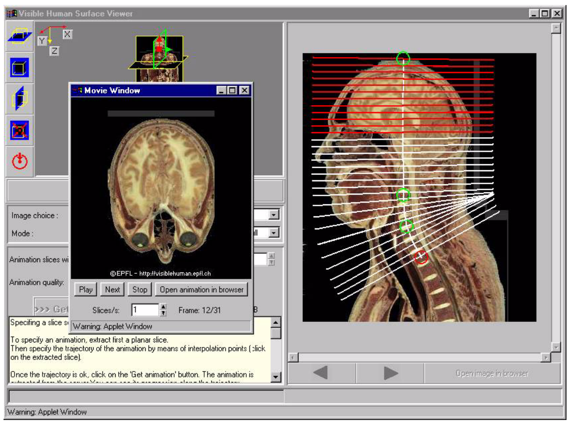


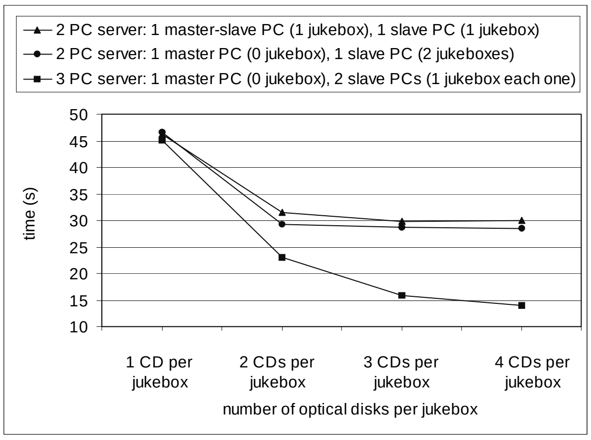
Figure 6-10: Animation display in synchronization with slice progression

Once the animation is completely downloaded, the Internet client can visualize it thanks to a Java MPEG decoder [44] integrated in the applet (Fig. 6-10). The applet allows to play, pause, stop and step-by-step display of the resulting animation, always in synchronism with the slice progress displayed on the user define-trajectory.

Accessing and extracting slice sequence animations from the Visible Human Head data set requires high processing power for extracting and resampling the slice parts and for encoding full slices into MPEG-1 format. In the experiment, an animation comprising 143 256x256 24-bit/pixel slices is extracted. Extents read from CD-ROMs are cached in the extent memory cache maintained by the library of striped file components. The size of the extent memory cache is of 100 MB. For this animation, 773 extents (123.8 MB) are read from optical disk and 8729 extents are read from the extent cache. Fig. 6-11 shows the time to extract the slice sequence animation[1] and the speed-up as function of the number of CD-ROMs on which the Visible Human data set is striped and for three server configurations:

- a 2 PC server configuration with a PC functioning as master and slave. Each server PC is connected to one NSM Satellite jukebox,
- a 2 PC server configuration where a server PC is dedicated to master operations. The second server PC is connected to two NSM Satellite jukeboxes and performs the slave operations,
- and a 3 PC server configuration (Fig. 6-1) where a server PC is dedicated to master operations and 2 server PCs perform the slave operations. Each server slave PC is connected to one NSM Satellite jukebox.

1. We assume that the optical disks already reside in their drives, i.e. disk loading and disk spin up times are not taken into account.



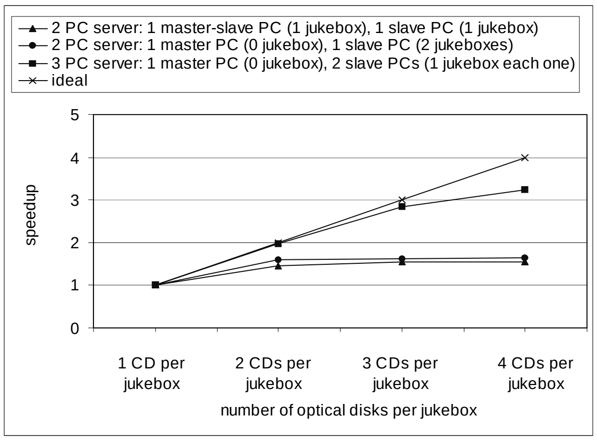


Figure 6-11: Slice sequence animation extraction times and speed-up for three server configurations as a function of the number of simultaneously accessed CD-ROMs, always with two jukeboxes

The master PC runs the WEB server process, merges slices parts into full slices and encodes full slices into MPEG-1 format. The slave PCs read extents from optical disks, extract and resample slice parts.

For the 2 PC server configuration where one PC is at the same time master and slave, when extracting the animation from 2 CD-ROMs, the I/O bandwidth is the bottleneck. From 4 CD-ROMs, the bottleneck shifts from the drive unit throughput to the limited processing power available on the server PC acting as master and slave. Fig. 6-12 shows the processor utilization of the 2 server PCs. When extracting the animation from 8 drive units, the processor utilization of the master-slave PC is 97.18% in average[1]. 33.59% are dedicated for slice part extraction and resampling, 1.26% for extent reading, 17.1% for merging slices parts into a full slice, 22.09% for encoding full slices into MPEG-1 format and 23.14% for the network interface and system

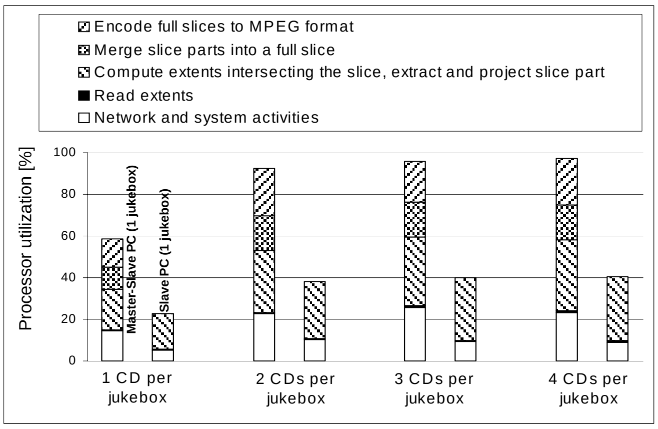


Figure 6-12: Processor utilization for the 2 PC server configuration where one PC server acts as master and slave, always with two jukeboxes active

1. The two Pentium III processors of the master-slave PC are used at 94.96% and at 99.40% respectively. The resulting mean processor utilization is of 97.18%.

activities. On the other hand, the slave bi-processor PCs are used at 40.63% in average (i.e. one processor at 37.62% and the other at 43.65%).

In order to reduce the load of the server PC acting as master and slave, we consider a 2 PC server configuration where one server PC is dedicated exclusively to the master operations (i.e. without jukebox hooked on) and the second server PC is connected to two NSM Satellite jukeboxes. For this configuration, from 4 CD-ROMs, the bottleneck shifts from the drive unit throughput to the limited processing power available on the server slave PC. Fig. 6-13 shows the processor utilization of the master and slave server PCs. When extracting the animation from 8 drive units, the slave PCs are used at 95.64%. 70.2% are dedicated for slice part extraction and resampling, 3.56% for extent reading and 21.88% for the network interface and system activities. On the other hand, the master PC processors are used at 40.63%.

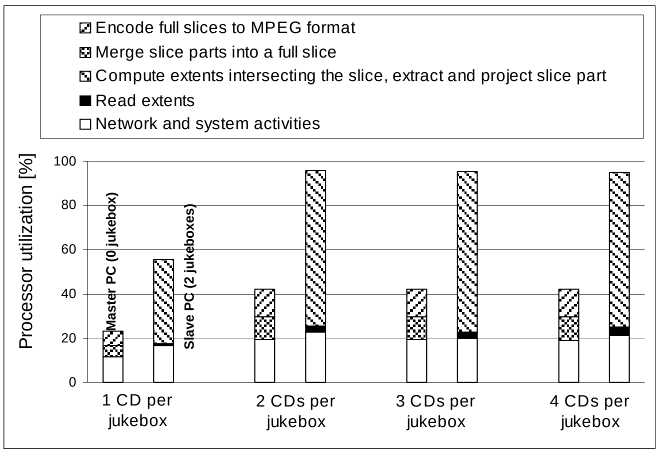


Figure 6-13: Processor utilization for the 2 PC server configuration where one PC is dedicated to master operations and the second server PC is connected to two jukeboxes

In the 3 PC server configuration (Fig. 6-1), a Bi-Pentium III PC is dedicated to the master functions and two Bi-Pentium III PCs are dedicated to the slave operations. For this configuration, with up to 3 CD-ROMs per slave PC, I/O bandwidth is always the bottleneck (Fig. 6-11). From 4 CD-ROMs per slave PC, the network bandwidth and the slave PC's become the bottleneck. Extracting the animation from 8 CD-ROMs takes about 13.9 s. In this time, 9502 slices parts are sent from slave PCs to the master PC. The slice part mean size is of 14336 bytes ranging from 90 to 25080 bytes. In addition to the slice part, the overhead of each message (i.e. control information) sent to the master PC is of 60 bytes. So, 9502 * (14336 + 60) = 136790792 bytes = 130.45 MB are sent to the master PC in 13.9 s, therefore 9.38 MB/s are transferred over the network. The Windows performance monitor measures a network throughput of about 10.4 MB/s. We consider this difference (i.e. a 9.8%) due to the TCP/IP protocol overhead. The network throughput (10.4 MB/s) is close to the maximum throughput of 12.5 MB/s sustained by the FastEthernet network. The master PC processors are used at about 45%. On the other hand, the slave PCs with an utilization of 94.155% start to saturate. These performance measurements show that a faster network is needed in order to scale the server with additional slave PCs.

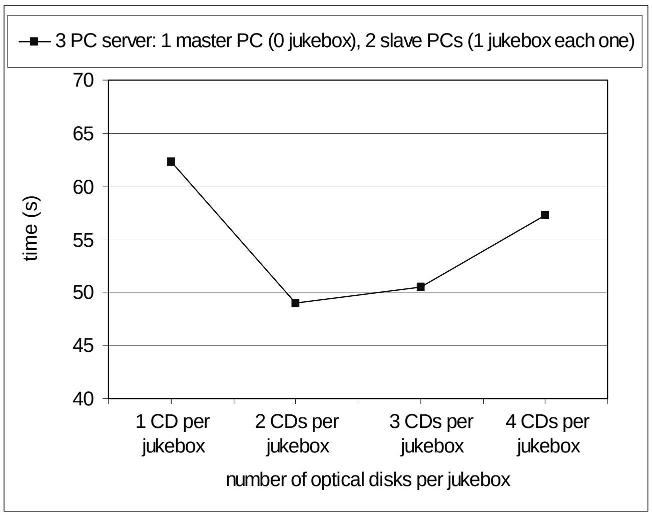


Figure 6-14: Slice animation extraction times as a function of the number of CD-ROM accessed simultaneously for a server configuration comprising a master PC and two slave PCs. Each slave PC is connected to one NSM Satellite jukebox

Fig. 6-14 plots the slice animation extraction times as a function of the number of CD-ROM accessed simultaneously for the 3 PC server configuration and taking into account the optical disk exchange and spinup times. For the considered slice animation, the minimum extraction time is reached when extracting the animation from 4 CD-ROMs distributed over two slave PCs. In this case, the slice animation takes 48 s, where 2 x 8.7 = 17.4 s are spent by each robot arm for performing two disk exchanges, 8.6 s for the spin-up time of the last loaded disk, and 22.9 s are spent for each drive unit for reading in parallel the file data. The timing diagram corresponding to the slice animation extraction is similar to the diagram shown in Fig. 6-9.

6.4 The 4D Beating Heart Slice Stream Server application

The *4D Beating Heart Slice Stream Server* application, described in Chapter 4, supports the visualization at a specified rate of freely oriented slices from a 4D beating heart volume. The considered beating heart data set consists of a sequence of 8-bits 3D volumic images, each one of size 256 x 256 x 112, i.e. 7 MB. With 384 time instants, the 4D beating heart sequence reaches a size of 384 x 7 MB = 2.62 GB. As described in Chapter 4, the 3D volume sequence of the beating heart is segmented into 4D extents of size 16 x 16 x 16 x 16 = 64 KB. To measure the slice stream extraction application performances, the experiment consists of requesting a 256x256 8-bit/pixel slice stream. In order to test the worst case, the selected slice orientation is orthogonal to one of the diagonals traversing the Beating Heart's rectilinear volume. For each set of 16 slices, 252 4D extents need to be read. In order to make fair comparisons, i.e. to have similar disk throughputs for the different experiments, when extracting the stream slices only the first 336 MB[1] of each CD-ROM are accessed. It means that 96, 192, 288 and 384 slices are extracted when accessing simultaneously to 2, 4, 6 and 8 CD-ROMs respectively.

Fig. 6-15 shows the performances obtained, in number of slices per second, as a function of the number of CD-ROMs accessed simultaneously. The number of slices extracted per second scales quite well when increasing the number of simultaneously accessed CD-ROMs (Fig. 6-15). When extending the stripe factor from 2 to 8, we obtain a speedup of 3.49. Parallel I/O bandwidth is always the bottleneck. Therefore increasing either the number of drive units per slave PC (i.e. increasing the number of jukebox per slave PC) or the number of server PCs (assuming each PC incorporates an equal number of jukeboxes) increases the number of drive units and offers a higher extracted slice throughput. Server processors and network are not fully utilized. When extracting slices from 8 CD-ROMs at full speed (i.e. at 9.9 slices/s), master PC processors are used at 47.5%, slave PCs at 26.25%, the network reaches a throughput of 2.05 MB/s and we obtain for each CD-ROM an effective throughput of 1.21 MB/s.

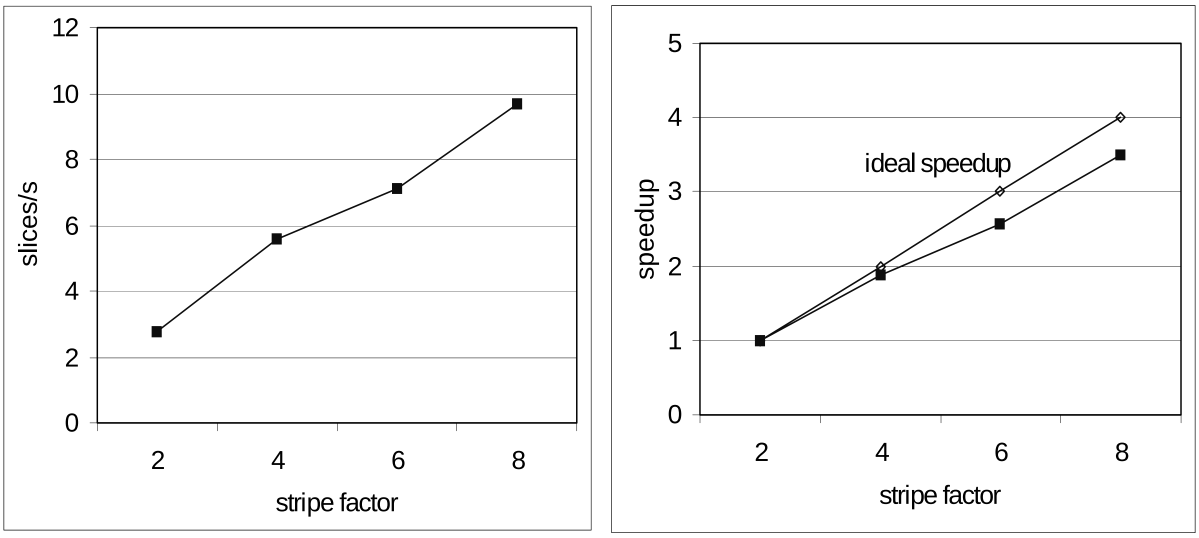


Figure 6-15: Performances in terms of slices/s and speed-up when extracting 256x256 8-bits slice streams as function of the number of CD-ROM accessed

To analyze the delay jitter of the slice parts delivered by the server PCs, the rate of the requested stream is varied in order to have a load of 50%, 70%, 90% and 95%, i.e. to read respectively 4.9 MB/s, 6.86 MB/s, 8.82 MB/s and 9.2 MB/s from 8 optical disks and extract respectively 5, 7, 9 and 9.4 slices per second.

For a 9.4 slice/s stream (95% of the maximum stream rate), the jitter delay (0.42 s) is slightly higher than the maximum jitter delay (0.33 s) at 5 slice/s (Figs. 6-17 and 6-16). Slice parts belonging to 16 consecutive time instants are sent to the master PC. In double buffering mode,

1. Frequent reading errors appear when reading a CD-ROM data from the 400 MB position. These errors may be due to the recording procedure.

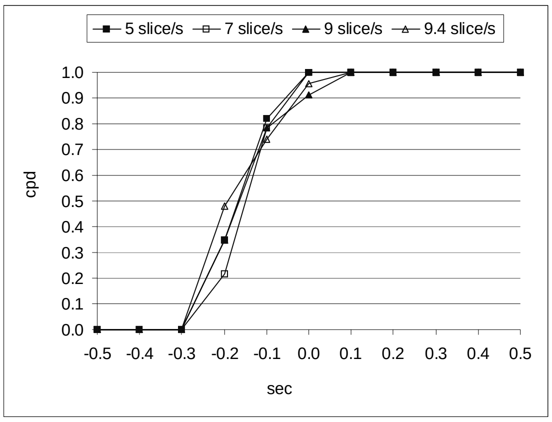


Figure 6-16: Cumulative probability distributions (cpd) of the delay jitter

the master PC should therefore have the memory to store 32 slices per slice stream (i.e. 2 MB per 256 x 256 8-bit/pixel slice stream). Since the measured delay jitter is always less than the time to display 16 slices, delay jitter does not require additional buffer space.

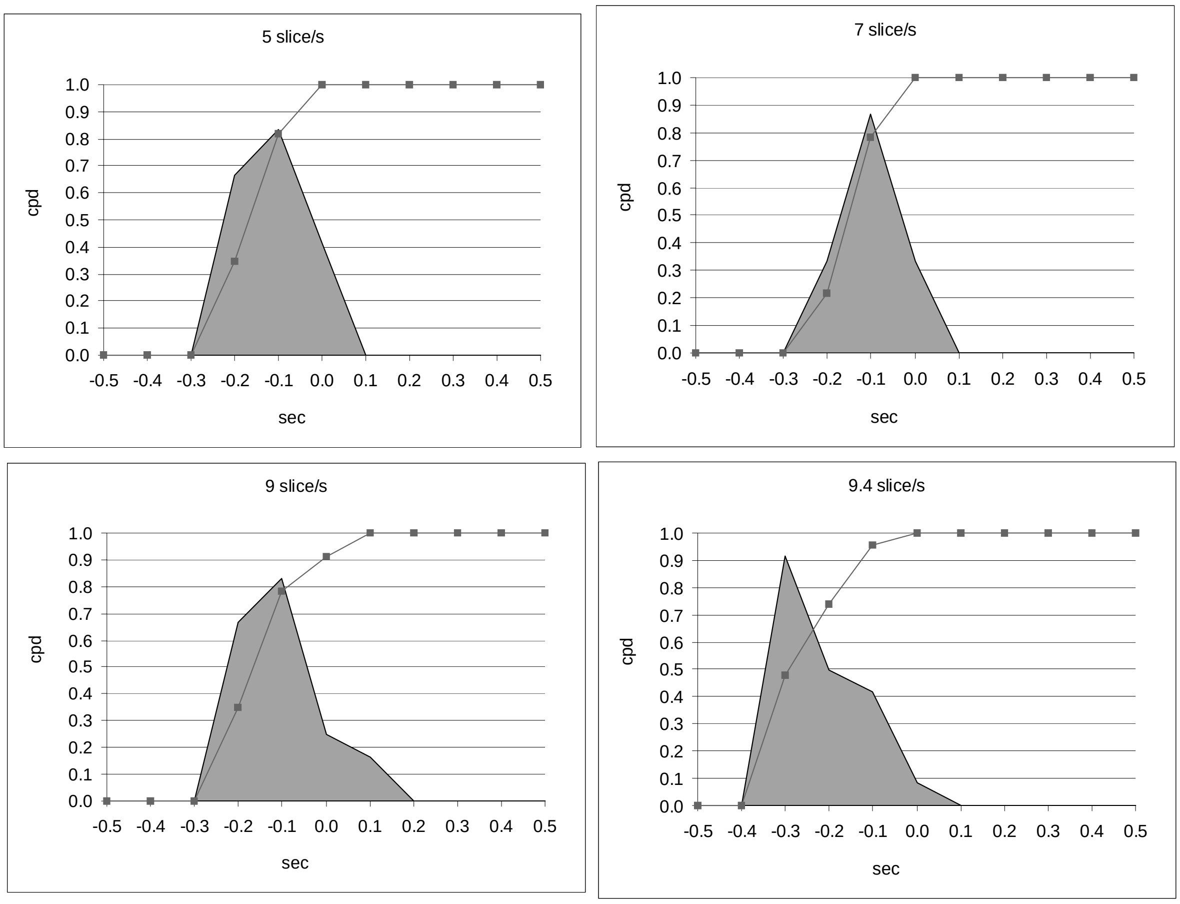


Figure 6-17: Delay jitter for a stream at 5 slice/s (50% of the maximum stream rate), at 7 slice/s (70%), at 9 slice/s (90%) and at 9.4 slice/s (95%)

6.5 Summary

This chapter describes a terabyte jukebox server comprising 3 server units interconnected by a 100 Mbits/s Fast Ethernet crossbar switch. One server unit (i.e. the master PC) incorporates a magnetic disk to store the shadow directory tree. Each one of the other server units (i.e. the slave PCs) comprises one PC connected to one NSM Satellite CD-ROM jukebox incorporating 4 read-only optical disk drive units and a magnetic disk for caching purposes. This configuration is easily extended, e.g. by adding a second jukebox per slave PC as well as additional server units made of 1 slave PC and two jukeboxes. The terabyte jukebox server is interfaced to a WEB server offering the terabyte server's services to Internet clients. We have analyzed the performances and the scalability of the teraserver jukebox architecture, especially for parallel server applications making use of several server resources, such as multiple server PCs and multiple optical disk drive units. We have tested the scalability of the terabyte WEB server by running the *Visible Human server* and the *4D beating heart slice stream server* applications on top of the jukebox terabyte server. These server applications access simultaneously their respective data sets striped over either 2, 4, 6 or 8 CD-ROMs. The parallel access experiments show that the available throughput can be considerably increased by accessing in parallel multiple optical disks located in different optical disk drives. A terabyte jukebox server architecture is appropriate for servers making use of a large number of files, e.g. the *Visible Human* data set for men and women of different ages, sizes and morphologies. Since we are able to access high-quality continuous media streams from multiple optical disks at a high throughput and since delay jitter is not significant, new perspectives for Web centric continuous media applications are offered. For example, video studio servers can be created, where clients edit and manipulate low-resolution media streams and the server executes the corresponding operations on the stream high-resolution version. In such applications, the original files are accessible from the optical disks and the new files are written to magnetic disks and later transferred to optical disks (each jukebox also incorporates a write optical disk writing unit). In such media editing applications, CD-ROMs may remain a long time in their drives and therefore justify a jukebox architecture incorporating a large number of drives, e.g. the Giant-ROM from ALP-Electronics [2] incorporating 120 drive units.

7 Conclusion

Parallel distributed memory servers for I/O and compute intensive continuous media applications are difficult to develop. A server application comprises many threads located in different address spaces as well as files striped over multiple disks located on different computers. In order to ensure performances close to those potentially offered by the underlying hardware and operating system, pipelined parallel programs need to be developed, which ensure that computations, disk accesses and network transfers occur simultaneously. This thesis proposes a new approach for developing servers for I/O and compute intensive continuous media applications.

CAP greatly simplifies the creation of pipelined parallel continuous media applications. By construction, it generates completely pipelined parallel applications: in front of each I/O, each processing and each network transfer operation, a token queue ensures that as soon as the current operation returns, the next token is ready to be consumed. In addition, CAP enables application programmers to specify at a high level of abstraction the parallel application structure. The parallel application is concisely described as a set of threads, operations located within the threads and flow of data and parameters between operations.

The parallel stream server library, based on CAP and on the stripe file component library, offers the basic continuous media services (i.e., an admission control algorithm, the isochrone behavior of the server nodes and a disk scheduling algorithm) to deliver stream data from multiple server disks to the client while guaranteeing the stream's real-time delivery contraints. Continuous media applications are supported by allowing the suspendion of the token flow during regular time intervals. Application programmers are free to create specific operations and to combine them with the predefined continuous media services to implement their parallel continouous media servers. Although the parallel stream library has been developed to support parallel I/O and compute intensive continuous media applications, it is able to support any continuous media application, e.g. video-on-demand servers. By creating the parallel stream server library, this thesis contributed to the further development of CAP.

The presented 4D beating heart application shows that thanks to CAP and to the parallel stream library, parallel continuous media server requiring a high I/O throughput (e.g. for accessing from disks 4D extents intersecting the desired slices) and a large amount of processing power (e.g. to extract slices from 4D extents and resample them into the display grid) can be built on top of a set of simple PCs connected to SCSI disks. By implementing the 4D beating heart server we demonstrate the applicability of CAP and of the parallel stream library on a real-world application. The evaluation of the 4D beating heart server shows that the server's performance is close to the maximal performance deliverable by the underlying hardware. The 4D beating heart application also suggests that tomographic equipment manufacturers may offer continuous 3D volume acquisition equipment by interfacing the acquisition device with commodity components, e.g. a cluster of PCs interconnected by a Fast Ethernect switch, with each PC being connected to several disks.

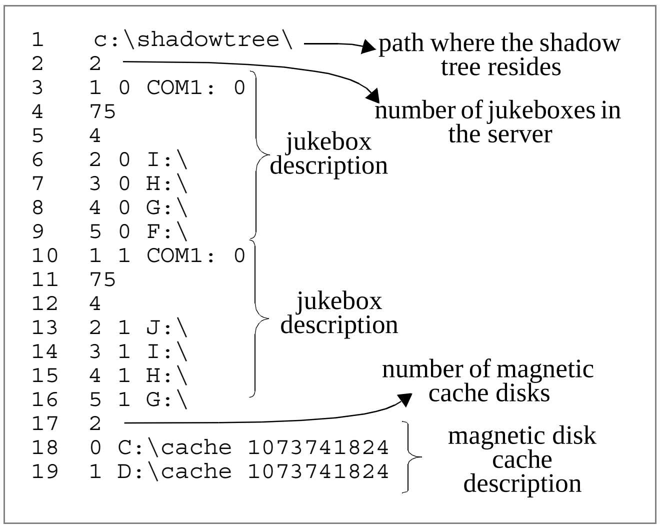
To minimize the cost of a multi-PC multi-disk architecture when storing large data volumes, we propose a scalable multi-PC multi-jukebox server architecture in order to offer highly accessible storage capacities and high-bandwitdh I/O throughput for digital libraries, image and multimedia repositories. For large data storage systems, optical jukebox storage systems are superior in terms of cost to storage systems based entirely on magnetic disks. The scalable terabyte server architecture we propose comprises several PCs, optical jukeboxes and possibly magnetic disks for caching purposes. One master server PC runs the server interface receiving client access requests and additional server PCs are connected each one to one or several jukeboxes. The master PC manages a shadow file directory containing the directory of all files residing on the jukeboxes' optical disks. It is also responsible for resource allocation. Slave server PCs are responsible for accessing files located on their jukeboxes as well as for caching optical disk files on magnetic disks. Thanks to its multiple server PCs, the scalable terabyte server provides high processing power in addition to its large storage capacity.

A WEB server running on a scalable terabyte optical jukebox server architecture has been created. The terabyte WEB server allows to access large files striped across multiple optical disks. The parallel access experiments show that the available throughput can be considerably increased by accessing in parallel multiple optical disks located in multiple optical disk drives. Parallel access to optical disks located each one in a different drive may offer new opportunities for high-quality continuous media editing and processing applications.

The present terabyte WEB suggests that scalable terabyte optical jukebox servers can be built with server units comprising each one a PC, magnetic disks and a number of jukeboxes. However, a number of issues remain open. Above a certain number of server units, the master server will need to be duplicated in order to avoid bottlenecks when accessing meta-information such as the directory tree. In addition, there is a need to ensure graceful degradation of the terabyte server in case of a server unit failure or in case of the failure of the master node. Since in most cases, the robotic device is the limiting factor, both new jukebox concepts and designs need to be explored in order to minimize the latency due to the media exchange operations. For example, it would be advisable to build a second level robotic device for moving magazines containing optical disks from shelves to the jukeboxes and vice-versa as well as between jukeboxes, for example for load balancing purposes.

Annex A. Terabyte server configuration file

To run the terabyte server, two configuration files must be provided. One for the CAP runtime system specifying the mapping of terabyte server threads to Windows NT processes and one for the terabyte server specifying the jukeboxes and the magnetic cache disks. This terabyte server's configuration file is read and interpreted by the *InterfaceServer* thread. It consists of three parts: the first part indicates the path where the shadow directory tree resides, the second part describes the jukeboxes in the server and the third part describes the magnetic disk cache.



```
1    c:\shadowtree\
2    2
3    1 0 COM1: 0
4    75
5    4
6    2 0 I:\
7    3 0 H:\
8    4 0 G:\
9    5 0 F:\
10   1 1 COM1: 0
11   75
12   4
13   2 1 J:\
14   3 1 I:\
15   4 1 H:\
16   5 1 G:\
17   2
18   0 C:\cache 1073741824
19   1 D:\cache 1073741824
```

path where the shadow tree resides
number of jukeboxes in the server
jukebox description
jukebox description
number of magnetic cache disks
magnetic disk cache description

Figure A-1: Configuration file describing the terabyte jukebox server architecture

Fig. A-1 shows an example of a terabyte server configuration file. Line 1 specifies the path where the shadow directory tree resides. From line 2 to 16, the jukeboxes forming the terabyte server are described. Line 2 indicates the number of jukeboxes. In our example, there are 2 jukeboxes, the first is described from line 3 to 9 and the second jukebox from line 10 to 16. For each jukebox, we have to specify, in the same line, the type of jukebox[1], the index of the *RobotServer* thread which controls the robotic arm, the serial port to communicate with the jukebox and the jukebox ID[2]. For example, line 3 specifies a jukebox of type 0 which is controlled by the *Robot-Server* thread with index 0 through the serial port COM1: and the jukebox ID is equal to 0. In the next line, the number of optical disks active in the jukebox is indicated (e.g. line 4 specifies 75 optical disks), and in the next line, the number of drive units (e.g. line 5 specifies 4 drive units used in this jukebox). For each unit drive, the drive unit index, the *ExtentServer* thread index who reads from this drive unit and the access path of this drive unit are specified. For example,

1. Currently, two types of jukebox are supported. Type 0 specifies a NSM Mercury jukebox. Type 1 specifies a NSM Satellite jukebox. The type of jukebox defines the command protocol to use for controlling the robotic arm.
2. When several jukeboxes are connected to the same serial port, the jukebox ID is used to target one of them.

in line 6, the drive unit with index 2 is accessed by the *ExtentServer* thread 0 via the path I:\. From line 17 to 19, the magnetic cache disks are described. Line 17 indicates the number of magnetic disks forming the cache. For each magnetic cache disk, the *ExtentServer* thread index, the access path and the maximum space used for the cache are specified in the same line. For example, line 18 specifies a magnetic cache disk accessed by the *ExtentServer* thread with index 0 through the path C:\cache and 1 MB of disk space is used for caching purposes.

Bibliography

1. M. J. Ackerman, Accessing the Visible Human Project, D-Lib Magazine: The Magazine of the Digital Library Forum, October 1995, http://www.dlib.org/dlib/october95/10ackerman.html
2. ALP Electronics SA, http://www.alpelectronics.com or http://www.swisstorage.com,
3. D. P. Anderson and G. Homsy, A Continuous Media I/O Server and Its Synchronization Mechanism, Computer, Vol. 24, No. 10, October 1991, pp. 51-57
4. D. P. Anderson, Y. Osawa and R. Govindan, A File System for Continuous Media, ACM Transaction on Computer Systems, Vol. 10. No. 4, November 1992, pp. 311-337
5. A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, V. S. Sunderam, A User's Guide to PVM Parallel Virtual Machine, Oak Ridge National Laboratory Report ORNL/TM-11826, July 1990
6. S. Berson, S. Ghandeharizadeh, R. Muntz and X. Ju, Staggered Striping in Multimedia Information System, Proc. of ACM Multimedia, 1994, pp. 391-398
7. J-C. Bessaud and R. D. Hersch, The Visible Human Slice Sequence Animation Web server, Proc. IS&T/SPIE Conf. on Internet Imaging, San Jose, California, Janvier 2001, SPIE Vol. 4311, in press
8. W. J. Bolosky et al. The Tiger Video Fileserver, in Proc. 6th Int'l Workshop on Network and Operating System Support for Digital Audio and Video, IEEE Computer Society Press, 1996, pp. 97-104
9. M. M. Buddhikot, Design of a Large Scale Multimedia Storage Server, Computer Networks and ISDN Systems, Vol 27, 1994, pp. 503-517
10. K. Byrne, Evaluating Jukebox Performance, IEEE Spectrum, July 1997, pp. 70-73
11. S-H. G. Chan and F. A. Tobagi, Hierarchical Storage Systems for On-Demand Video Servers, SPIE, Vol. 2604, 1996, pp. 103-120
12. E. Chang and A. Yakhor, Scalable Video Data Placement on Parallel Disk Arrays, SPIE, Vol. 2185, 1994, pp. 208-221
13. P. M. Chen and D. A. Patterson, Maximizing Performance in a Striped Disk Array, ISCA, 1990, pp. 322-331
14. A. L. Chervenak, Tertiary Storage: An Evaluation of New Applications, PhD dissertation, University of California at Berkeley, Computer Science Department, December 1994,

Technical Report UCB/CSD 94/847, http://www.cc.gatech.edu/fac/Ann.Chervenak/papers/ChevernakThesisNov94.ps.Z

15. A. L. Chervenak, D. A. Patterson and R. H. Katz, Choosing the Best Storage System for Video Service, Multimedia '95, San Francisco, California, 1995, pp. 109-119

16. A. L. Chervenak, D. A. Patterson, R. H. Katz, Storage Systems for Movies-on-Demand Video Servers, Proc. 14th IEEE Symposium on Mass Storage Systems, 1995, IEEE Press, pp. 246-256

17. T. Chiueh, Performance Optimization for Parallel Tape Arrays, In Proceedings of the 1995 International Conference on Supecomputing, Barcelona, Spain, July 1995, pp. 385-394

18. H. Cluster, Inside Windows NT, Microsoft Press, 1993

19. H. Cluster, Inside Windows NT File System, Microsoft Press, 1994

20. T. H. Cormen and D. Kotz, Integrating Theory and Practice in Parallel File Systems, Proceedings of the 1993 DAGS/SP Symposium on Parallel I/O and Databases, June 1993, Hanover, NH, pp. 64-74

21. A. Dan and D. Sitaram, An Online Video Placement Policy based on Bandwidth to Space Ratio (BSR), Proceedings of ACM SIGMOD '95, San Jose, 1995

22. A. Dan, M. Kienzle and D. Sitaram, A Dynamic Policy of segment replication for load balancing in Video-on-Demand Server, Multimedia Systems, 1995, Vol. 3, pp. 93-103

23. F. Davis, W. Farrell, J. Gray, R. Mechoso, R. Moore, S. Sides and M. Stonebraker, EOS-DIS alternative architecture, Technical Report Sequoia 2000 TR 95/61, University of California at Berkeley, April 1995. http://www.research.microsoft.com/research/BARC/Gray/EOS_DIS/default.htm

24. Y. N. Doganata and A. N. Tantawi, Storage Hierarchy in Multimedia Servers, Multimedia Information Storage and Management, S. M. Chung, (ed), Kluwer Academic Plublishers, 1996, Chapter 3

25. Y. N. Doganata and A. N. Tantawi, Making a cost-effective video server, IEEE Multimedia, Vol. 1(4), Winter 1994, pp. 22-30

26. A. L. Drapeau and R. Katz, Striped Tape Arrays, Proc. 12th IEEE Symp. Mass Storage systems, Monterey, CA, April 1993, pp. 257-265

27. A. L. Drapeau and R. Katz, Striping in Large Tape Libraries, Supercomputing '93 Proc., Portland, OR, November 1993, pp. 378-387

28. D. H. C. Du and Y-J. Lee, Scalable Server and Storage Architectures for Video Streaming, Proc. IEEE International Conference on Multimedia Computing and Systems, June 1999, pp. 62-67

29. C. Federighi and L. A. Rowe, A Distributed Hierarchical Storage Manager for a Video-on-Demand System, Symposium on Electronic Imaging Science & Technology, IS&T/SPIE, San Jose, CA, Februray 1994

30. D. G. Feitelson, et. al. Parallel I/O Subsystems in Massively Parallel Supercomputers, IEEE Parallel & Distributed Technology, Vol. 3, No. 3, Fall 1995, pp. 33-47

31. E. A. Fox, Advances in Interactive Digital Multimedia Systems, IEEE Computer, October 1991, pp. 9-21

32. D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, Multimedia Storage Servers: A Tutorial and Survey, IEEE Computer, 1995

33. B. A. Gennart, B. Krummenacher, L. Landron and R. D. Hersch, GigaView Parallel Image Server Performance Analysis, Proceedings of the World Transputer Congress, Transputer Applications and Systems, IOS Press, Como, Italy, September 1994, pp. 120-135

34. B. A. Gennart, J. Tárraga and R. D. Hersch, Computer-Assisted Generation of PVM/C++ Programs using CAP, In Proc. of EuroPVM'96, LNCS 1156, Springer Verlag, Munich, Germany, October 1996, pp. 259-269

35. B. A. Gennart and R. D. Hersch, Program Parallelization with CAP, http://lspwww.epfl.ch/publications/gigaview/captutorial.pdf

36. S. Genusa, Special Edition Using ISAPI, Que Corporation, 1997, ISBN: 0-7897-0913-9

37. S. Ghandeharizadeh and D. DeWitt, A Multiuser Performance Analysis of Alternative Declustering Strategies, In Proceedings of Data Engineering '90

38. S. Ghandeharizadeh, and L. Ramos, Continuous Retrieval of Multimedia Data using Parallelism, IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 2, August 1993

39. G. Ghandeharizadeh and C. Shahabi, On Multimedia Repositories, Personal Computers, and Hierarchical Storage Systems, Proc. of 2nd ACM Multimedia Conference, pp. 407-416

40. L. Golubchnik, R. R. Muntz and R. W. Watson, Analysis of Striping Techniques in Robotic Storage Libraries, In Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems, Monterey, CA, September 1995, pp. 225-238

41. A. S. Grimshaw, Easy-to-use Object-Oriented Parallel Processing with Mentat, IEEE Computer, Vol. 26, No. 5, May 1993, pp. 39-51

42. A. S. Grimshaw, Dynamic, Object-Oriented Parallel Processing, IEEE Parallel & Distributed Technology, Vol. 1, No. 2, May 1993, pp. 33-47

43. A. Guha, The Evolution to Network Storage Architectures for Multimedia Applications, Proc. IEEE International Conference on Multimedia Computing and Systems, June 1999, pp. 68-73

44. C. Hasan, Java Mpeg Player Software, chsan@dcc.uchile.cl, Department of Computer Science, University of Chile, Santiago, Chile, downloaded at http://www.mpeg.org, November 1998

45. R. D. Hersch, B. Gennart, O. Figueiredo, M. Mazzariol, J. Tárraga, S. Vetsch, V. Messerli, R. Welz and L. Bidaut, The Visible Human Slice Web Server: A First Assessment, Proc. IS&T/SPIE Conf. on Internet Imaging, San Jose, California, Janvier 2000, SPIE Vol. 3964, pp. 253-258

46. D. Jadav and A. Choudhary, Designing and Implementing High-Performance Media-on-Demand Servers, IEEE Parallel and Distributed Technology, Summer 1995, pp. 29-39

47. R. Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, Inc. ISBN 0-471-50336-3

48. Jukeman software from Smart Storage Inc., http://www.jukeman.com or http://www.smartstorage.com

49. R. Katz, High Performance Network and Channel-Based Storage, Computer Science Division Report No. UCB/CSD 91/650, U. C. Berkeley, 1991

50. R. Katz, et. al. Robo-line Storage: Low Latency, High Capacity Storage Systems over Geographically Distributed Networks, Computer Science Division Report No. UCB/CSD 91/651, U. C. Berkeley, 1991

51. E. E. Kim, CGI Developer's Guide, SAMS.net Publishing, 1996, ISBN: 1-57521-087-8

52. A. Kraiss and G. Weikum, Vertical Data Migration in Large Near-Line Document Archives based on Markov-Chain Predictions, Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997, pp. 246-255

53. S-W. Lau and J. C. S. Lui, Scheduling and Data Layout Policies for a Near-line Multimedia Storage Architecture, Multimedia Systems, Vol. 5, No. 5, September 1997, pp. 310-323

54. D. Le Gall, MPEG: A Video Compression Standard for Multimedia Applications, Communications of ACM, April 1991

55. J-Y. B. Lee, Parallel Video Server: A Tutorial, IEEE Multimedia, Vol. 5, No. 2, April-June 1998, pp. 20-28

56. T. D. C. Little and D. Venkatesh, Probabilistic Assignment of Movies to Storage Devices in a Video-on-Demand System, In Proc. Fourth International Conference on Network and Operating System Support for Digital Audio and Video (NOSSDAV ‘93), 1993, pp. 213-224

57. C. Martin, P. S. Narayanan, B. Ozden, R. Rastogi and A. Silberschatz, The Fellini Multimedia Storage Server, Multimedia Information Storage and Management, S. M. Chung, (ed), Kluwer Academic Plublishers, 1996, Chapter 5, pp. 117-146

58. V. Messerli, O. Figueiredo, B. A. Gennart and R. D. Hersch, Parallelizing I/O Intensive Image Access and Processing Applications, IEEE Concurrency

59. V. Messerli, B. A. Gennart, R. D. Hersch, Performances of the PS2 Parallel Storage and Processing System, In Proc. of the 1997 International Conference on Parallel and Distributed Systems, Seoul, Korea, December 1997, IEEE Press, pp. 514-522

60. V. Messerli, Tools for Parallel I/O and Compute Intensive Applications, Ph.D. Thesis 1915, Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, February 1999

61. Microsoft Corporation, Window NT installable file system kit (IFS), http://www.microsoft.com/ddk/IFSkit/,

62. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, The International Journal of Supercomputer Applications and High Performance Compting, Vol. 8, 1994

63. Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, Technical Report, July 1997, http://www.mpi-forum.org

64. R. Nagar, Windows NT File System Internals: A Developer's Guide, O'Really & Associates, Inc., September 1997

65. NSM Storage Inc., http://www.nsmstorage.com,

66. Open System Resource Inc., http://www.osr.com,

67. B. Ozden, R. Rastogi and A. Silberschatz, A FrameWork for the Storage and Retrieval of Continuous Media Data, in Proc. of the IEEE International Conference on Multimedia Computing Systems, Washintong, 1995, IEEE Press, pp. 2-13

68. D. A. Patterson, G. Gibson, and R. H. Katz, A Case for Redundant Arrays of Inexpensive Disks (RAID), ACM SIGMOD Conf., 1988, pp. 109-116

69. K. K. Ramakrishnan, L. Vaitzblit, C. Gray, U. Vahalia, D. Ting, P. Tzelnic, S. Glaser and W. Duso, Operating system support for a video-on-demand file service, Multimedia Systems, Vol. 3, 1995, pp. 53-65

70. P. V. Rangan and H. M. Vin, Designing File Systems for Digital Video and Audio, Proceedings of the 13th ACM Symposium on Operating System Principles, ACM Press, Monterey, California, 1991, pp. 81-94

71. P. V. Rangan and H. M. Vin, Efficient Storage Techniques for Digital Continuous Media, IEEE Transactions on Knowledge and Data Engineering, Vol. 5, No. 4, August 1993

72. A. L. N. Reddy and J. C. Wyllie, Disk-Scheduling in a Multimedia I/O System, Proc. First ACM Int'l Conf. Multimedia, ACM Press, New York, 1993, pp. 225

73. A. L. N. Reddy and J. C. Wyllie, I/O Issues in a Multimedia System, Computer, Vol. 27, No. 3, March 1994, pp. 69-74

74. A. Ruegg, Processus stochastiques, Presses polytechniques romandes, 1989, ISBN 2-88074-168-8

75. M. Russinovich and B. Cogswell, Examining the Windows NT Filesystem, Dr. Dobb's Journal, February 1997, pp. 42-49

76. K. Salem, H. Garcia-Molina, Disk Striping, Proc. of the Second International Conference on Data Engineering, Washington, D.C., February 1986, pp. 336-342

77. C. Shahabi, M. H. Alshayeji and S. Wang, A Redundant Hierarchical Structure for a Distributed Continuous Media Server, 1997

78. P. J. Shenoy, P. Goyal and H. M. Vin, Issues in Multimedia Server Design, ACM Computing Surveys, December 1995, Vol. 27, No. 4, pp. 636-639

79. S. D. Stoller and J. D. DeTreville, Storage Replication and Layout in Video-on Demand Servers, Proc. 5th Intl. Workshop on Network and Operating System Support for Digital Audio and Video, Durham, NH, April 1995, pp. 351-363

80. K. Suzuki, K. Nishimura, A. Uemori and H. Sakamoto, Storage Hierarchy for Video-on-Demand Systems, SPIE, Vol. 2185, 1994, pp. 198-207

81. J. Tárraga, V. Messerli, O. Figueiredo, B. A. Gennart and R.D. Hersch, Computer-Aided Parallelization of Continuous Media Applications: the 4D Beating Heart Slice Server, Proc. ACM Multimedia 1999, Orlando, Florida, pp. 431-441

82. W. Tetzlaff, M. Kienzle and D. Sitaram, A Methodology for Evaluating Storage Systems in Distributed and Hierarchical Video Servers, In COMPCON 94, pp. 430-439, IEEE Computer Society Press, Spring 1994

83. F. A. Tobagi, J. Pand, R. Baird and M. Gang, Streaming RAID: A Disk Array Management System for Video Files, In First ACM Conference on Multimedia, August 1993

84. H. M. Vin and P. V. Rangan, Designing a Multiuser HDTV Storage Server, IEEE Journal on Selected Areas in Communications, Vol. 11, No. 1, January 1993, pp. 153-164

85. H. M. Vin, P. Goyal and A. Goyal, A Statistical Admission Control Algorithm for Multimedia Servers, In Proceedings of the ACM Multimedia ‘94, San Francisco, California, October 1994, pp. 33-40

86. H. M. Vin, A. Goyal and P. Goyal, Algorithms for Designing Large-Scale Multimedia Servers, Computer Communications, Vol. 18, No. 3, March 1995, pp. 192-203

87. A. Vina, J. Lopez, A. Molano and D. del Val, Real-Time Multimedia Systems, Thirteen IEEE Symposium on Mass Storage Systems, 1994, pp.77-83

88. P. G. Viscarola and W. A. Mason, Windows NT Device Driver Development, Macmillan Technical Publishing, 1999

89. S. Wolfram, Mathematica: A System for Doing Mathematics by Computer, Addison-Wesley Publishing Company, Inc., 1991

Biography

Joaquín Tárraga Giménez was born on September 12th, 1971 in Valencia, Spain. In October 1989 he studied at Universidad Politécnica de Valencia where he obtained in April 1995 his engineering diploma in computer science. He did his diploma work at the Peripheral Systems Laboratory of the Ecole Polytechnique Fédérale de Lausanne during 6 months thanks to an ERASMUS grant. From April 1995 to July 1997, he worked as an assistant at the Peripheral Systems Laboratory mainly for developing the CAP Computer-Aided Parallelization tool within the Parallel Virtual Machine (PVM) environment as well as a Windows NT file system filter driver for accessing transparently to files stored in optical disk jukeboxes. On July 1997, he started his PhD research. He pursued his research on tools for parallel I/O- and compute-intensive continuous media applications and on scalable server architectures based on optical disk jukeboxes. His research interests include parallel continuous media servers, parallel I/O streaming, parallel and distributing computing, computer-aided parallelization and scalable storage server architectures.

Publications

- B. A. Gennart, J. Tárraga and R. D. Hersch, Computer-Assisted Generation of PVM/C++ Programs using CAP, In Proceedings of EuroPVM'96, LNCS 1156, Springer Verlag, Munich, Germany, October 1996, pp. 259-269
- J. Tárraga, V. Messerli, O. Figueiredo, B. A. Gennart and R.D. Hersch, Computer-Aided Parallelization of Continuous Media Applications: the 4D Beating Heart Slice Server, Proc. ACM Multimedia 1999, Orlando, Florida, pp. 431-441
- R. D. Hersch, B. Gennart, O. Figueiredo, M. Mazzariol, J. Tárraga, S. Vetsch, V. Messerli, R. Welz and L. Bidaut, The Visible Human Slice Web Server: A First Assessment, Proc. IS&T/SPIE Conf. on Internet Imaging, San Jose, California, Janvier 2000, SPIE Vol. 3964, pp. 253-258