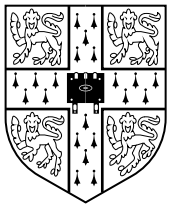
Network File Server Design for Continuous Media

Paul Wenceslas Jardetzky

Darwin College
University of Cambridge



A dissertation submitted for the degree of
Doctor of Philosophy

August 1992

Summary

Significant effort is currently being applied to the integration of diverse digital communication services including voice, video, images and text into computer networks. Voice and video, characteristic of continuous media, differ from traditional data in that the data delivery has strict time requirements. This temporal nature places demands on a system which are difficult to satisfy with current computer architecture, networking technology and software design.

The real-time characteristics of continuous media have not been taken into account in the design of popular computer networks or operating systems. Ethernet, the most pervasive local area network, can introduce transmission delay which adversely affects the behavior of continuous media applications. Current operating systems, such as Unix and MSDOS, do not guarantee a low response time for services they provide and at high utilisation this ensures a performance decrease for concurrent processes. Consideration of the new constraints shifts design attention from the performance that is possible to the performance that is required to achieve real-time communication.

Furthermore, network services based on these systems are not capable of supporting concurrent continuous media applications without loss of quality at high loads. In particular, file systems for traditional data have not been designed to support voluminous time dependent data. This dissertation examines the issues faced by the network file system designer when servicing voice and video traffic. An implementation of a continuous media file server is presented to demonstrate the effectiveness of design decisions and to illuminate the impact of continuous media storage on computer architecture.

To my family

Preface

I am grateful to my supervisor Professor Roger Needham for the opportunity and encouragement to pursue this research.

My thanks and appreciation also go to numerous members of the Computer Laboratory. Particular members of the System Research Group deserve special mention including, Joe Dixon, Cosmos Nicolaou, Derek McAuley, Glenford Mapp, Cormac Sreenan, Eoin Hayden, Tim Wilson, Richard Black and Mark Hayter. Their valuable advice and constant stimulation provided an excellent environment in which to develop my understanding.

For reading and commenting on earlier drafts of this dissertation, I thank Cormac Sreenan, Richard Black, Simon Crosby, Cosmos Nicolaou, Roger Needham, Tim Wilson, Heather Leitch, Mark Hayter and Derek McAuley.

My deepest appreciation is reserved for my family, who have supported and encouraged me through every aspect of the experience. This research was partially funded by the University of Cambridge Computer Laboratory.

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

Contents

List of Figures vi

List of Tables vii

Glossary of Terms viii

1 Introduction 1

1.1 Motivation 1

1.2 Continuous Media 3

1.3 Environment 3

1.4 Aims of the Dissertation 4

1.5 Caveats 5

1.6 Outline 6

2 Related Research 7

2.1 The Etherphone System 7

2.2 ISLAND 8

2.3 UCSD Multi-media File Server 10

2.4 MIT High Bandwidth File Server 12

2.5 NSS 13

2.6 Olivetti Research Video Repository 14
2.7 UCB Continuous Media File Server 15
2.8 Summary 18

3 Problems and Requirements **21**

3.1 Resource Management 21
3.2 Quality of Service 26
3.3 Flow Control 30
3.4 Synchronisation 32
3.5 Summary 33

4 Design Considerations **35**

4.1 Distribution 35
4.2 File System Instances 39
4.3 Concurrency Control 48
4.4 CM File Design 48
4.5 Summary 50

5 Implementation **51**

5.1 Architecture 51
5.2 Interface 52
5.3 Disk Management 54
5.4 File Representation 62
5.5 Caching 63
5.6 Concurrency Control 64
5.7 Flow Control 64

5.8 Synchronisation 65
5.9 Summary 67

6 Experimentation **69**

6.1 Pandora segments 71
6.2 CFR Throughput 72
6.3 Audio Streams 74
6.4 Video Streams 79
6.5 Combined Audio and Video 82
6.6 CMFS Failure 89
6.7 Recommendations 90
6.8 Summary 91

7 Conclusion **93**

Bibliography **97**

List of Figures

2.1 Decreasing Buffer Space as Concurrency Increases 11

2.2 A Client Read Session in the UCB CMFS 16

3.1 Client Rate Based Flow Control 31

4.1 Sequential (i) and Indexed Sequential (ii) Files 40

4.2 Access to a Direct File 41

4.3 Three Files in a File Allocation Table 42

4.4 File System Layout in Unix 43

4.5 A Fast Path From the Network to Disk 49

5.1 A File Service Hierarchy 52

5.2 CMFS File Inode Structure 55

5.3 Single File Read Ahead in the CMFS 57

5.4 Concurrent File Read Ahead in the CMFS 58

5.5 Thread control in the CMFS 61

5.6 An Open File Representation 62

5.7 Thread Synchronisation with Real-Time 65

6.1 CMFS test apparatus 69

6.2 Pandora Audio and Video Segments 72

6.3 Clock Interrupt Effects on Packet Inter-arrival Time 74
6.4 Audio Inter-arrival Times With 3 and 5 Streams Active 75
6.5 Total Thread Execution Time 77
6.6 Audio Packet Processing Times 78
6.7 Effects of Stream Contention on Audio Processing Times 78
6.8 Video Inter-arrival Times With A Single Stream 80
6.9 Video Inter-arrival Times With 2 and 3 Streams Active 81
6.10 Video Processing Times 82
6.11 Audio and Video Inter-Arrival Times 83
6.12 Audio and Video Inter-Arrival Times With Two Combined Streams . 84
6.13 Audio and Video Processing Times 85
6.14 Audio and Video Processing Times With Two Combined Streams . . 86
6.15 Audio and Video Playback Inter-Packet Read Times 87
6.16 Audio and Video Playback With Lower Frequency Audio 87
6.17 Audio and Video Playback With Two Combined Streams 88

List of Tables

3.1 Continuous Media Bandwidth Requirements 22

3.2 Storage Device Data Transfer Characteristics 24

3.3 Continuous Media File Sizes 25

5.1 Micropolis 1578 Disk Transfer Times 59

6.1 CFR throughput During Concurrent Disk Activity 73

6.2 Scheduling Jitter During Concurrent Disk Activity 73

Glossary

ACL	Access Control List
ANSA	Advanced Networks Systems Architecture
ATM	Asynchronous Transfer Mode
CAV	Constant Angular Velocity
CBR	Constant Bit Rate
CCITT	Comitée Consultatif International Télégraphique et Téléphonique
CD-ROM	Compact Disk Read Only Memory
CFR	Cambridge Fast Ring
CLV	Constant Linear Velocity
CMFS	Continuous Media File Server/Service
CR	Cambridge Ring
CRC	Cyclic Redundancy Check
DAN	Desk Area Network
DARPA	Defence Advanced Research Projects Agency
DMA	Direct Memory Access
ECC	Error Correction Codes
FAT	File Allocation Table
FCFS	First Come First Serve
FIFO	First In First Out
FM	Filing Machine

FS	File Server
IP	Internet Protocol
ISO	International Standards Organisation
ISR	Interrupt Service Routine
JPEG	Joint Picture Experts Group
LAN	Local Area Network
LLSS	Low Level Storage Service
LRU	Least Recently Used
MPEG	Motion Picture Experts Group
MSNA	Multi-Service Network Architecture
MTBF	Mean Time Between Failure
MTTR	Mean Time To Repair
NSS	Network Sound System
PTM	Packet Transfer Mode
QOS	Quality Of Service
RAID	Redundant Array of Inexpensive Disks
RPC	Remote Procedure Call
SAR	Segmentation And Reassembly
SAS	Single Address Space
SSTF	Shortest Seek Time First
STM	Synchronous Transfer Mode
TOG	Thrown On the Ground
TCP	Transmission Control Protocol
UCSD	University of California at San Diego
VBR	Variable Bit Rate
WORM	Write Once Read Many

Chapter 1

Introduction

1.1 Motivation

Distributed multi-media applications are becoming feasible with the advancement of fast network technology, image compression techniques and distributed computing environments. Thus, concerted effort is currently being applied to the integration of various communication media including text, images, audio and video into the digital computer network environment.

Personal computers abound today; some 96 million machines are currently distributed worldwide. The desire to increase the number of applications for these machines has led to multi-media hardware designed to realise digital audio and video within a personal environment.

It is clear, however, that greater connectivity and data sharing are paramount to the success of multi-media computer communication. For this reason, attention to the performance of network based services is increasing and emerging computer architectures reflect the desire for computer systems that meet the demands of new, more diverse, data transfer requirements.

Although current technology is capable of producing systems that provide digital audio and video services on a single machine, popular computer infrastructure does not provide the ability to communicate efficiently using continuous media in real-time. Unacceptable delays are introduced by both hardware and software, when supporting multiple concurrent clients.

This dissertation concentrates on issues related to the provision of a network based storage facility for digital audio and video data. A catalogue of conventional network file systems is presented in [**Borghoff92**] and a short survey is presented in

[**Satyanarayanan89**] outlining basic issues essential to the design of a distributed file service. Many issues, such as location, naming, availability, system management, crash recovery and security, are outside the scope of this dissertation. Instead, attention is given to design characteristics which differ fundamentally from the conventional distributed file system due to the type of media stored.

File system design is motivated primarily by existing file access analysis. This has involved static and dynamic monitoring of a working system. Common observations of traditional file systems in an academic environment, compiled from [**Burrows88**], include:

- Files sizes tend to be small (approximately 4-16 KBytes);
- Read operations are more frequent than write operations;
- File read and write operations are generally sequential;
- Fine grain file sharing is rare;
- Most files are read and written by a single user.

An improvement in performance of current network file systems may be achieved by caching recently referenced blocks or whole files under the observation that recently accessed files are likely to be accessed again in the near future. The effect on performance can be dramatic due to the reduction in communication with remote server machines. Actual continuous media file sizes will be discussed in Chapter 3, but their expected large size may preclude caching of recently accessed multi-media files. The access characteristics of these files may warrant caching the first few seconds of audio or video on the assumption that if it is to be accessed again, the access may start from the beginning of the file. However, if access to continuous media data does not exhibit temporal locality, caching becomes ineffective. Consider the behavior of a cache employing a least recently used (LRU) algorithm with an incoming continuous media stream. The cache will perform as a circular buffer and data which was previously in the cache will be cleared. This is very significant because the design of modern distributed files systems relies on caching for performance improvement. Very little data about continuous media access characteristics is available and it will depend entirely on existing multi-media file system interaction with successful application programs. As a result, accurate design judgements are difficult to make with respect to this issue. Instead, a designer must gather the expected file access requirements and re-evaluate the validity of any assumptions made during the design process.

Clearly, at least one of these observations is not valid for a file system supporting audio and video. Multi-media file sizes are typically large and this affects parameters

such as efficient mapping of files to physical disk storage blocks. It is necessary then to examine the requirements for a file service which can support continuous media.

1.2 Continuous Media

Continuous media is the term emerging to describe audio and video data in a computer environment. Certainly, in a digital domain, this term is inaccurate when the mathematical definition of the word is applied. Continuity would imply that the absolute value of the time difference between video or audio samples may always be observed as arbitrarily close to zero. This is only true with an analog representation of audio and video.

However, the term is meant to describe only the individuals' perception of continuity in the time domain. Perhaps a more accurate description for digital audio and video, considering current digital technology, would be discrete periodic data which is sensitive to delay and delay variation. [1] This definition of continuous media is consistent with both fixed and variable rate digital sources and will be used throughout this dissertation to describe such data. Compression of continuous media can reduce the overall bandwidth requirement, but does not change its temporal nature. It is desirable to keep the variance in delay or *jitter* to a minimum and within some known bound.

Because audio and video data exhibit temporal characteristics, information that arrives late relative to the time necessary to maintain continuity may be useless and therefore might be discarded. In this respect, the reconstruction of an acceptable signal after transmission is tolerant of a certain amount of loss or error during transmission. This contrasts with conventional computer data transfer which generally allows for greater transfer delay to maintain data integrity.

1.3 Environment

Experimentation was carried out using the Cambridge Fast Ring (CFR) [**Hopper88**] [**Temple84**], a network with asynchronous transfer mode (ATM) characteristics. The CFR is a slotted ring architecture which provides fair bandwidth sharing by requiring that used slots be freed and made available to the the next station on the ring. The CFR slot or cell is 36 octets in length of which 4 octets are used for routing information and 32 octets are available for user data. Each station provides

[1] The term *isochronous* is used to describe the transmission of continuous media, implying finite size data samples generated at fixed intervals.

a memory mapped interface to a single cell receive and transmit FIFO. System interrupts are generated on cell reception and transmission and data transfer is accomplished by the host processor.

The WANDA [**Dixon92**] lightweight distributed operating system kernel is used as the execution platform. WANDA provides a high performance interprocess communication facility based on message passing, support for lightweight processes or threads, semaphores for thread synchronisation, and an internetworking protocol based on the Multi-Service Network Architecture (MSNA) [**McAuley90**].

Distribution and control of the file server is achieved with the Advanced Network System Architecture (ANSA) [**ANSA89**]. The ANSA Testbench software provides a Remote Procedure Call (RPC) mechanism over MSNA. A central naming facility, known as the Trader, acts as a negotiating point for clients wishing to locate network services. This is achieved by servers informing the Trader of their services. Clients initiate connections with network services after contacting the Trader for service location information.

Real-time audio and video streams are provided by the Pandora Box [**Hopper90**]. This is a multi-media peripheral which is attached to a general purpose workstation. The main components include a video digitiser, a video mixer, an audio processing board and a CFR network interface. Audio and video are digitised and processed separately from the host processor. The resulting video images are mixed with the host video display output and presented in a seemingly integrated manner. The X11 Window System is extended to support control of the Pandora subsystem via a transputer link interface. Distributed control of the device is accomplished through a library interface supporting communication with the host workstation via the Transmission Control Protocol (TCP).

1.4 Aims of the Dissertation

This dissertation is presented as a guide to be used in the design of file services that support real-time digital audio and video. The goal is to demonstrate that a distributed file service in support of these media may be built without special purpose hardware if designed properly. There are four main objectives within the dissertation. Firstly, it is necessary to identify those parameters that affect the file system performance and thus provide the criteria for making desirable design decisions.

The second objective is to show the important design criteria which maximise performance of a system through implementation and measurement of a file server prototype using conventional Winchester disks and a high speed network.

The third objective is to extend the notion of Quality of Service parameters (QOS), often used in network terminology, as an operating systems paradigm. QOS specification is an active area of research motivated by the desire to support integrated traffic on networks which, in their basic form, do not guarantee network bandwidth to individual clients. Negotiation by clients with network service nodes may provide the desired guarantees while preserving network flexibility. Resource management is the primary objective of QOS in the operating system, providing multi-media applications with the required guarantees. Resources other than the network are involved in a multi-media file system and QOS negotiation should take them into account.

Finally, it will be shown how a specially designed Continuous Media File Service (CMFS) may be integrated into a general distributed computing architecture.

1.5 Caveats

It is always possible to engineer a high performance system when one is desired or deemed necessary using special purpose or proprietary hardware. However, the cost effectiveness of such solutions will continue to dictate most engineering decisions. The work presented in this dissertation supports the design of a continuous media file service in a distributed environment using common computer components. The machinery consists of a VME bus based CPU (Motorola 68030), two 380MByte Winchester disks with a SCSI interface supporting direct memory access (DMA), 8MBytes random access memory and 2MBytes of battery backed non-volatile memory.

Although, coding and compression algorithms affect the transmission characteristics of continuous media, the details of algorithms remain beyond the scope of this text. [2] The proposed solution to the storage of continuous media data is independent of the encoding and compression algorithms used. The only requirement for stream playback is the availability of timing information.

[2]The Consultative Committee on International Telephone and Telegraph (CCITT) provides several digital audio and video coding recommendations including G.711, G.721, G.722 and H.261. The International Standards Organisation (ISO) and the CCITT formed the Joint Picture Experts Group (JPEG) aimed at providing compression with a wide range of image quality. ISO also formed the Motion Picture Experts Group (MPEG) to develop standards for video compression. A discussion of these may be found in [**Ang92**].

1.6 Outline

The chapters in this dissertation are arranged as follows:

Chapter 2 describes current work with high bandwidth file servers designed specifically for use in a multi-media distributed system.

Chapter 3 outlines the problems and requirements identified when designing a low level file system for continuous media. Resources critical to the performance of a system are identified.

Chapter 4 lists the design considerations for a distributed file service highlighting important differences for audio and video support. A general distributed file system architecture is reviewed for the integration of a continuous media file system. The design considerations for local file systems are also presented and separated from the distributed case.

Chapter 5 describes the design and implementation of the continuous media file server (CMFS) prototype used for experimentation within this dissertation. A detailed justification for the design decisions is given.

Chapter 6 presents measurements of the implemented server and identifies features of the design that affect the overall performance and scalability.

Chapter 7 reviews the work presented and identifies areas in which current computer architecture and system software can be modified to support continuous media effectively.

Chapter 2

Related Research

Several approaches to the storage of continuous media are reviewed. This is not intended as an exhaustive survey of the integration of digital audio and video into digital computer networks. Instead, the projects outlined are the most relevant research for the provision of continuous media storage services. Each system is discussed with respect to the following issues:

- Network capability: The ability to transport continuous media streams on a digital network.
- Disk usage: How continuous media files are stored on disk.
- Buffer management: How the available memory is managed to support concurrent access to continuous media files.
- Synchronisation: The ability to synchronise related continuous media streams.
- Admission control: The ability to maintain quality of service by explicitly checking client requirements.
- Flow control: The ability to control transmission or display rates.
- Capacity: The number of supportable continuous media streams.

2.1 The Etherphone System

The Etherphone System [**Swinehart83**] employed a voice storage server [**Terry88**] which used a single large file on a dedicated Cedar File System [**Gifford88**]. An important observation was that, in terms of data throughput, voice services were beyond the capabilities of the general purpose Cedar File System and a discussion of the voice server implementation given in [**Ades87**] indicates that this intermediate

solution was adopted by the project to allow the system to work in an existing environment.

The voice file server maintains an exclusive lock on the single large file and provides a voice file system above it. Voice files are immutable. Directory structures and editing facilities are implemented as separate higher level services, reducing the general file system overheads to ensure lightweight access to voice files. One writer and many readers are allowed concurrent access to telephone quality audio files at 64kbps over a 1.5 Mbit Ethernet [**Metcalfe76**] network. In fact, client machines are located on a 3 Mbit Ethernet and the audio server on a 1.5 Mbit Ethernet and are separated by a single gateway. Performance reports indicate that it can handle about eight simultaneous connections.

There is no acceptance control in the system and no service guarantees are made. Voice intervals, or *ropes*, are stored as individual files and the file server provides enough buffering to play several ropes without introducing any delay between each rope. However, under heavy load, service times may degrade below those required to provide acceptable audio quality. Security is implemented through the use of access control lists and voice encryption. No flow control mechanism is provided in the interface. Voice files are played at the same rate as they were recorded. Since voice files were rarely synchronised with each other, no inter-stream synchronisation primitives were designed. Integration with textual documents is provided through the Tioga multi-media editor.

The file server may be left in an inconsistent state after a server crash, so utility programs are provided to correct any errors. Although a sufficient level of functionality was achieved to demonstrate the feasibility of an audio storage service without the need to build a special purpose file system, it would be interesting to determine to what extent the underlying Cedar file system decreased voice file server performance.

It should be noted, however, that the main goal of this project was to provide useful abstractions for the integration of voice in a computer network. This was done successfully without specific attention to audio file system design.

2.2 ISLAND

The ISLAND architecture [**Calnan88**] uses a similar approach to the Etherphone project except the performance of the underlying file system is not optimised by creating a single large file in which to implement an audio storage service. The system is built over the TRIPOS [**Richards79**] operating system and the Filing Machine (FM). ISLAND includes a translator between a telephone operating on the Cambridge Ring [**Wilkes79**] and file storage serving two purposes. The first,

is to serve as a central point of control for audio devices and file store, facilitating synchronisation between the two entities. That is, a client requiring access to an audio file makes a request to the Translator to buffer audio packets. The Translator contacts the filing machine to perform file operations. The second function is to convert voice streams into large blocks to be delivered to the filing machine at a lower rate. This method of rate adaptation again permits the use of a general purpose file system.

The storage server used is actually divided into two parts. The File Server (FS) [**Dion81**] is the basic storage service which is accessed through the Filing Machine. The FM provides transparent use of both remote and local disks. A loss in performance was noted due to the need for the extra level of indirection through the FM. However, experiments with the Translator using only a local disk indicate this overhead is small. The Translator was not able to support more than one voice stream. The CPU was indicated as the performance bottleneck due to low level protocol overhead and the limited Cambridge Ring bandwidth. Although the total ring bandwidth is 10 Mbits/s the actual point to point bandwidth available to a single transmitting station on a single slot ring is 1.3 Mbits/s [**Temple84**].

Voice files are broken into the following components:

- Phrases: This is a sequence of one or more talk spurts;
- Sections: One or more phrases;
- Paragraph: One or more sections;
- Speech: A collection of one or more paragraphs.

The result is a tree structured voice file with clear boundaries for the implementation of a voice editor. The Translator uses silent pauses to subdivide phrases, sections and paragraphs. For example, silent periods greater than one quarter of a second are used to subdivide the sequence of phrases into sections. Similarly, paragraphs are divided into sections by pauses of about ten seconds. During a recording session, these subunits comprise a speech. This representation was found to incorporate large overheads during subdivision of voice because it requires that new subdirectories and/or files be created whilst recording. This introduced delay in command processing due to the extra file system processing.

Again, the goal of this project was to create a range of audio services including audio representation in documents and audio editing. No flow control is provided and acceptance control is not implemented, so performance will degrade as the number of concurrent clients increases.

2.3 UCSD Multi-media File Server

The University of California at San Diego (UCSD) Multi-media File Server [**Vin91**] was motivated by the Etherphone project and provides abstractions such as *strands* and *ropes*, defined below, to organise audio and video data as multi-media files. The prototype was implemented on a PC-AT equipped with video capture hardware and a local disk. Communication with the PC is achieved via TCP/IP over a local area Ethernet. It is important to note that while control of the file server is distributed, multi-media data are recorded and played on the local PC hardware. To minimise bus bandwidth requirements, the server reads video blocks directly into the video display hardware. This optimisation is not available when considering transmission of continuous media over a network and the management issues related to transmission are not addressed. The list of basic abstractions is as follows:

- Frame: The basic unit of video.
- Sample: The basic unit of audio.
- Strand: An immutable sequence of continuously recorded audio samples or video frames. Large strands are implemented by a three level index structure allowing random access.
- Rope: A collection of multiple strands tied together with synchronisation information.
- Block: the basic unit of disk storage. Blocks may be homogeneous, containing data belonging to one medium, or heterogeneous containing data belonging to multiple media.

As indicated above, multi-media files may be stored separately with homogeneous blocks or interleaved in heterogeneous blocks. The file system does not assume contiguous allocation for strands in order to maintain flexibility for editing and to reduce disk fragmentation. The model stipulates a *continuity requirement* that the time to read a block does not exceed the duration of its playback. Two questions are identified for a disk layout policy that will satisfy the continuity requirement of a continuous media file:

- Granularity: What should the block sizes be?
- Scattering parameter: What should the physical distance between successive blocks of a strand be?

To answer these questions three possible architectures are outlined to maintain continuous retrieval of media data:

- Sequential: This serialises read and display operations. Each block of data is transferred from disk and played in full before initiating the next block transfer. The continuity requirement is met only if the sum of the time to read and display the block does not exceed the duration of its playback;
- Pipelined: this architecture performs read and display operations in parallel. Two buffers are used to maintain the parallelism. One for the disk and one for the display. The continuity requirement is satisfied if the time to read a block does not exceed the duration of its playback;
- Concurrent: In a concurrent architecture, multiple disk read operations are performed in parallel. If p is the degree of concurrency, then the continuity requirement for playback will be met if the time to read a block does not exceed the duration for playback of $(p-1)$ blocks.

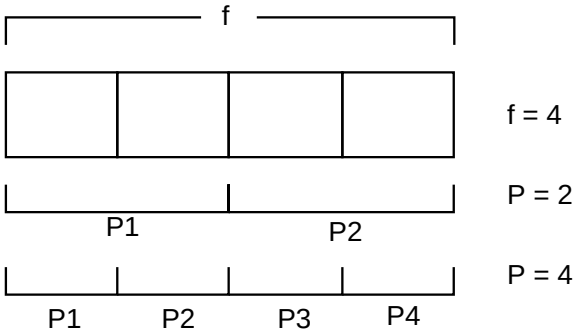


Figure 2.1: Decreasing Buffer Space as Concurrency Increases

The pipelined architecture was chosen and continuity equations are derived. The granularity of storage is determined to be f/p, where f is the number of internal frame buffers at the video display and p is the number of permitted concurrent accesses. The calculation is based on direct transfer of data from disk to the video display. This suggests that as the number of concurrent requests increases, the number of frames per block should decrease. Figure 2.1 illustrates the number of concurrent clients at two and four when there are a total of four frame buffers. When p is equal to two, two frame buffers are available to each client. By increasing p to four, only one frame buffer is available to each client. This may be a problem if servicing individual disk blocks for $p_1, \ldots, p_4$ introduces unacceptable delay in the streams. Note that a decrease in buffer space as the number of concurrent clients increase is not correct, since we might expect each stream to require a larger buffer for jitter removal as the number of concurrent clients increases.

The admission control algorithm discussed is based on an analysis of how to ensure that the continuity requirement for each request is satisfied within the limits of the hardware. The major limiting factor of this algorithm is that it only takes into account the disk as a potential bottleneck in the system. Synchronisation is

accomplished through block level correspondence. That is, a one to one mapping of blocks between media streams is stored in the multi-media rope data structures. This is limited in that synchronisation may only be achieved between one audio and one video stream at a time. No support for synchronising multiple ropes is provided.

2.4 MIT High Bandwidth File Server

The MIT server described in [**Vaitzblit91**] uses the Unix operating system and a 100 Mbit/s FDDI ring network [**Flatman86**] to implement a prototype high bandwidth file server. A convincing argument is presented for the unsuitability of conventional file servers for continuous media based on the probability of exceeding the maximum capacity of the system during concurrent activity. Assuming 5 active clients at an average transfer rate of 2 KBytes/s [1] (with a standard deviation of 4 KBytes/s), Vaitzblit calculates the probability of exceeding the bandwidth capacity of a 1 Mbytes/s SCSI disk at less than 8×10^{-5}. Assuming that 5 video clients transfer data at an average rate of 1.5 Mbits/s, the probability of exceeding the SCSI disk capacity is 50%. This is orders of magnitude greater than the conventional case. Statistical multiplexing of continuous media streams yielding *best effort* service is inappropriate in the multi-media environment because service may degrade below an acceptable level. Resource reservation can deterministically allocate bandwidth, reducing the probability that concurrent access will exceed the maximum capacity of the system.

Continuous media files are grouped within *containers* such that the playback of all files stored in the container does not exceed the maximum disk transfer rate. Essentially, a container provides the mechanism for reserving disk bandwidth. Files created within the container are managed in a log structured manner to reduce access latency. That is, all write requests are serialised and converted into sequential disk writes. Played individually, a container definition ensures that its streams' retrieval requirements are met. Unfortunately, no real-time guarantee can be made in the case of simultaneous access to multiple containers.

An interesting feature of the server is the use of a client based flow control mechanism. Using this scheme, a client requests each individual segment of audio or video from a server. The rate of stream delivery is determined by the rate of client requests. This contrasts with server based flow control in which a client requests that a server deliver a stream at a specific rate and from that time onward the server maintains the delivery rate. The advantages and disadvantages of each scheme will be discussed in Chapter 3. An admission control algorithm based on a deadline

[1] This is the observed average data transfer rate for a single user in a conventional environment [**Ousterhout85**].

driven scheduling algorithm is suggested, but not implemented in the prototype. Synchronisation of related streams is implemented by the inclusion of timestamps in every frame and the ability to reference frames by temporal offsets.

The disk layout strategy outlined includes synchronous disk interleaving. This is used to increase disk throughput by synchronising multiple disks in parallel operation. This will be discussed in more detail in Chapter 4. The architecture also includes node striping where file data may be located on independent storage servers, all part of a larger logical storage system. This has the obvious disadvantage that any node failure in the system renders the entire logical storage system unavailable.

The file server prototype was implemented on a DECstation 5000 with SCSI and FDDI interfaces connected via the Turbochannel I/O bus. Two storage servers were configured each with two RZ56 disk drives. The standard UDP/IP communication protocol was used for network transfer of continuous media data. A sustained data transfer rate of 2 MBytes/s is reported for a 1 node, 2 disk configuration. This is equivalent to 10 simultaneous audio/video streams, where the audio data rate is 64 Kbits/s and the video data rate is 1.5 Mbits/s.

While many of the ideas within the design are defensible, important features such as admission control, extendible files and synchronous disk interleaving are not implemented. Furthermore, the use of the Internet Protocol (IP) is questionable for continuous media traffic as indicated in [**McAuley90**]. A connection-less datagram protocol such as UDP may introduce unacceptable delays in an internetworking environment. It should also be noted that modifications to the Unix kernel were necessary to reduce the number of data copy operations between user space and kernel space. This is a further illustration of the unsuitability of Unix for continuous media data.

2.5 NSS

The Northwestern University School of Music implemented a network sound system (NSS) for Unix [**Roth85**]. This system integrates a specialised sound server together with the Unix file system by mapping Unix directories to remote audio files. The goal was to maintain the convenience of manipulating sound files on a local file system. This involved the addition of Unix like commands specialised for the management of sound files. Because sound files are Unix directories, all other Unix utilities such as *find* can be used to operate on sound files. The directory entry holds the network address of the sound system containing that sound file. A central sound system server maintains the consistency between sound file links and the file data. This process is also responsible for resource allocation management within the sound file

system.

The sound file system is stored on a dual ported disk and management access is via the Unix raw device interface. Another system on the network is connected to the second port of each disk and performs real-time access to sound data. The details of this system are not available in the literature. Disk allocation is performed in contiguous data unit sizes called *Allocation Units*, configurable at installation and potentially different on each disk in the system.

Sound file representation is split into three categories:

- Digitised sample file;
- Midi sound file;
- Splice sound file.

A physical sound file descriptor specifies the type of file on the sound file system. Although it is useful to classify the different sound file representations, this approach requires the file system to have knowledge of specific sound representations. It is unlikely that there will be a fixed number of ways to encode sound and this method is not easily extendible when new sound file representations are to be integrated.

Issues such as admission control, guaranteed continuous access, flow control and network bandwidth management are not addressed and no performance characteristics are available.

2.6 Olivetti Research Video Repository

The Olivetti Research Laboratory developed a continuous media file repository for project Pandora [**Hopper90**]. The repository is built over TRIPOS and its FH5 filing system. The configuration consists of an Acorn R260 (30MHz ARM3 processor), 16 MBytes RAM, a CFR interface and a SCSI interface serving 4 Wren 6 drives offering a storage space of 2.5 GBytes.

This approach is another example of using a general purpose file system for continuous media. The success of the video repository relies upon the lightweight design of TRIPOS and large (16 MBytes) random access memory. The large memory, used in conjunction with read ahead and write behind, aids in the removal of jitter from either the disk or network interfaces. The viability of this approach is supported also by the large margin between disk bandwidth and the network bandwidth used in the system. The Wren 6 disks used are each capable of 1.5 MBytes/s sustained

transfer rate and the CFR interface used is capable of approximately 3.5 MBits/s. The addition of a second network interface and load balancing software is used to distribute traffic among the two interfaces, but this has yet to be measured. A 50% increase in network bandwidth is expected, but this still does not bring network bandwidth equal to disk bandwidth. Furthermore, the use of a much higher data rate network will shift the bottleneck from the network to another resource in the system. With a single network interface the repository can support 3 audio/video streams each with a combined data rate of 928 Kbits/s, giving a total of 2.6 Mbits/s.

The repository is used for applications such as video mail, multi-media bulletin boards and the latest television news. Some features include:

- Record and playback of synchronised audio and video;
- Random access to and within continuous media files;
- Real-time operation for multiple concurrent clients;
- Limited editing (Cut, Join, Pause operations).

Continuous media streams represent independent threads of execution and different media are stored in separate files. An index file is associated with each data file and maps timing information to data file offsets in support of editing operations. Synchronisation between streams is achieved relative to a system clock with a resolution of 5 ms. This synchronisation mechanism is also used in the prototype system for this dissertation and will be described in detail in Chapter 5.

Rate based flow control is provided by the server. Clients may vary the speed of any continuous media stream during playback. No acceptance control is implemented.

2.7 UCB Continuous Media File Server

The Continuous Media File Server CMFS outlined in [**Anderson91**] is designed to support real-time and non real-time access to files. Simulation results supply a comparison of different design choices with a variety of hardware assumptions. The actual implementation of the CMFS provides only a subset of the following design issues needed to provide data rate guarantees.

- Real-time semantics: The CMFS proposes a well defined, but flexible interface for real-time access to continuous media files;

- Disk Layout: A single spindle disk drive is assumed and no layout assumptions are provided. Layout is only constrained by the access bounding functions described below;
- Acceptance test: How to decide when new sessions may be accepted;
- Disk head scheduling: How to order disk read and write requests.

A real-time session is established through a *request_session* interface specifying the required data rate and a maximum workahead parameter which describes the amount of read ahead (or write behind) desired. Once established, a session is a contract between the CMFS and its client; each guaranteeing that its behavior will obey certain rules. Sessions are monitored by a logical clock which begins at time zero when a session is created. The advancement of the clock is determined by the rate parameter and the clock may stop if a client exceeds the maximum work ahead.

Flow control is implicitly provided by the work ahead parameter. That is, a client may pause and resume a session simply by not reading any data. The server clock will eventually catch up to the time calculated from the work ahead parameter and stop processing the stream. Figure 2.2 shows a client read session with the following parameters:

Y: the maximum work ahead argument to the *request_session* call.

P(t): the index of the next byte to be put into the FIFO by the CMFS at time t.

G(t): the index of the next removable byte by the client at time t.

C(t): the value of the logical clock at time t.

B: the size of the buffer in bytes.

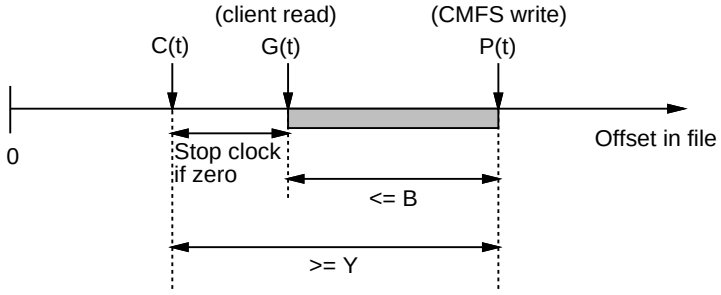


Figure 2.2: A Client Read Session in the UCB CMFS

This is essentially a hybrid flow control scheme between a client based and server based strategy. Essentially, if the client stops reading at some time t, the server will stop writing to the FIFO when C(t) is equal to G(t). This approach to flow control

will be discussed in more detail in Chapter 3, but has one major disadvantage. It does not allow clients to easily change the rate of a stream. The stream is either always at the rate specified at the beginning of the session or the rate is zero. No renegotiation is allowed. It may be possible to create a new session with new characteristics, but this will most certainly introduce unnecessary overhead when only renegotiation is desired.

Synchronisation of related streams is performed by the client after the return of all *request_session* calls. At which point, all session logical clocks are at zero. The client begins reading from sessions after this time and the buffer space needed for synchronisation depends on the skew of the start messages.

As previously mentions, the CMFS makes no assumptions about a particular disk layout. Instead, the following bounding functions are defined:

- For a given file F and n, $U_F(n)$ is an upper bound on the time to read n logically contiguous blocks of F, independent of the initial position of the disk head and the starting block number to read;
- $V_F(i, n)$ is an upper bound on the time needed to read n blocks of file F starting at block i.

These bounding functions are meant to take into account any possible contributions to the total read times such as, CPU scheduling overhead, disk seek times, rotational latency, and interrupt handling latency. Allocation and compaction strategies are not addressed in the literature and only fixed sized files are supported in the prototype.

Admission control is based on notion of a *Workahead-Augmenting Set* (WAS) which provides file system state as a vector of concurrent real-time accesses to the system. The abstraction of *slack time* is defined to be the amount of time that an operation may be postponed without starving any session and is calculated as the difference between a feasible WAS and the *Minimal Feasible* WAS. The acceptance test checks for the existence of a feasible WAS based on the layout parameters and available buffer space. Buffer space allocation among existing sessions is adjusted to optimise the system performance.

Three disk scheduling policies are examined relative to the WAS abstraction:

- Static/Minimal policy: This policy simply repeats the order of operations from the Minimal Feasible WAS previously calculated;
- Greedy policy: This policy performs the largest possible read for each session. At each iteration, the session with the smallest workahead is chosen and the

largest number of blocks n is read such that 1) the bounding function $V_F(i,n)$ is less than the current slack time and 2) the n blocks fit into the currently available buffer space;

- Cyclical plan: This is similar to the Greedy policy except that the read ahead is distributed over all sessions so that the slack time can be maximised for non real-time access.

The simulation results indicate that the Greedy and Cyclical plans are significantly better than the Static policy, but there is little difference between the two former policies. Performance results are reported from simulation. They indicate that 90% of the disk transfer capacity[2] can be achieved with 4 MBytes of memory supporting approximately 6 1.4 Mbit/s streams or 85 MBytes of memory supporting approximately 140 64 Kbits/s streams.

The UCB CMFS prototype runs on the SunOS 4.1 version of Unix, but the Unix file system is circumvented by the use of a raw SCSI disk interface. Communication with the CMFS is achieved using the TCP/IP protocol suite. The most important criticisms of this design are the following. Firstly, the CMFS does not support variable size files. Furthermore, the overhead of allocation strategies and index block I/O are not considered. Although the CMFS is a distributed file service, network transmission of data is not attempted and not surprisingly the performance results indicate that the number of concurrent sessions is limited by the disk transfer rate. Lastly, The acceptance control algorithm does not take into account the management of the network interface bandwidth.

2.8 Summary

Of the research reviewed, it should be noted that the Etherphone, ISLAND and Olivetti video repository use existing file systems to implement a continuous media file service. This is motivated primarily by the benefits associated with not having to write a specialised file system. Indeed, in most cases, a naming scheme, access control information and a directory structure are provided by the general purpose file store. The above projects represent the most relevant research to the subject matter in this dissertation. Other notable projects include [**Schmandt88**] [**Lazar87**] and [**Steinberg89**]. They are also involved with providing audio and video services on a digital network, but do not specifically address the issues of providing a network based continuous media file service.

[2]These calculations assume the use of a CDC Wren 5 drive with an average access time of 39 ms and a transfer rate of 11.8 Mbits/s.

Many of these systems endeavor to provide audio and video file abstractions for structured organisation of multi-media data. This is not necessarily an advantage when considering the support of the data rates desired. Few of the systems have addressed the issue of admission control.

Chapter 3

Problems and Requirements

A continuous media file service may be classified as a *soft real-time* system as opposed to a *hard real-time* system. A hard real-time system commonly describes a system where it is absolutely necessary that responses occur within the specified deadline. The critical nature of some decisions in a hard real-time system is often expressed by the level of hardware or software redundancy required. A soft real-time system describes a system where response times are important but the system will function correctly if deadlines are occasionally missed. This is further distinguished from an interactive system where there are no explicit deadlines.

A general discussion of system requirements for continuous media is given in [**Nicolaou91**]. This chapter outlines the problems and requirements for the design of a distributed continuous media file service. The prevalent issue is that of system resource management. Specific resources of a continuous media file server which must be managed are outlined. The provision of Quality of Service (QOS) support to multiple clients accessing continuous media data is discussed.

Other requirements such as flow control and synchronisation of related continuous media streams are presented.

3.1 Resource Management

3.1.1 The Network

Access to distributed multi-media data significantly increases the load on a communication network and it is important to describe the characteristics of a network which must meet the requirements.

- Large aggregate bandwidth;
- Bounded total delay which includes bounded access time and bounded point to point transmission delay;
- Low and bounded jitter characteristics.

Actual bandwidth requirements depend on the resolution and encoding of the continuous media stream and may be roughly summarised in Table 3.1, reproduced from [**Dixon92**]:

Service	Coding	Transfer Rate (bps)		Example Service
		Mean	Peak	
Standard Audio	CBR	64K	64K	Telephony
	CBR	106K	106K	Pandora Audio [1]
	VBR	12K	24K	Telephony
High Quality Audio	CBR	2M	2M	HI-FI Distribution
Low Quality Video	CBR	2M	2M	Videophone
	CBR	1.56M	1.56M	Pandora Video [2]
Standard Quality Video	CBR	34M	34M	TV Distribution
	VBR	10M	34M	TV Distribution
High Quality Video	CBR	140M	140M	HDTV Distribution
	VBR	70M	140M	HDTV Distribution

Table 3.1: Continuous Media Bandwidth Requirements

The coding column indicates either a constant bit rate coding (CBR) or a variable bit rate coding (VBR). Synchronous networks have been developed to facilitate transfer of constant bit rate sources such as digital telephony. This is an ideal architecture from the viewpoint of an application requiring a fixed data rate. Techniques such as silence suppression can be used to decrease the data rate of digital telephony. The result is a variable bit rate stream so that a fixed bandwidth channel is no longer the best solution as it results in inefficient use of channel bandwidth. Packet computer networks are more efficient, but introduce delay and jitter to the application.

Advances in high speed networking provide part of the infrastructure needed for the successful integration of digital audio and video into a distributed environment. Asynchronous Transfer Mode (ATM) networks provide the ability to handle variable rate traffic. However, further research is required to provide QOS to clients

[1] 8K frames/sec (40% framing overhead).

[2] 25 frames/sec at 256x240 resolution (2% framing overhead).

with varying bandwidth requirements in networks which provide no inherent bandwidth guarantees. As a compromise between Packet Transfer Mode (PTM) and Synchronous Transfer Mode (STM), the goal of ATM networks is not only to increase network bandwidth and increase channel utilisation, but to reduce network communication latency. These goals are suited to the advancement of integrated real-time communication. The ATM cell is small and of a fixed size (32, 48, or 64 bytes) to minimise switching delays in network routers and reduce network packet jitter, while keeping header overhead down. A more complete discussion of this and other network technology is given in [**Ang92**], focusing specifically on the use of continuous media on digital networks. A discussion in [**McAuley90**] calls for new communication protocol development to maximise the advantages of ATM in an internetworking environment. The relevant conclusions from both authors are summarised below:

- The use of virtual circuits reduces delay and jitter in network routers;
- Fixed rate transmission simplifies admission control policies;
- Under high utilisation, short term congestion and cell loss is inevitable;
- A network management scheme should include (1) connection admission control, (2) bandwidth enforcement, (3) source rate control, and (4) end-to-end flow control.

3.1.2 Storage Devices

A storage device must be able to perform at a *minimum* rate equal to the sum of the rates of any open continuous media streams. Without this constraint, a system will quickly exhaust its buffer capacity. It is reasonable to assume that access to continuous media objects will be mostly sequential. However, a random access device is better suited to the support of concurrent applications. Of the most common storage devices listed in Table 3.2, Winchester disk technology best serves the requirements of access to multiple continuous media streams. Digital tape and current optical disks suffer from high seek times, reducing the number of supportable concurrent clients. Delay in servicing individual disk requests is introduced during concurrent access. It is therefore important to have device seek times at a minimum.

CD-ROM and some WORM (Write Once, Read Many) drives organise data in a single spiral track with a uniform density. The drive maintains a constant linear velocity (CLV) by increasing speed as the head moves toward the outer edge of the disk. This approach reduces the speed of random access since the tracking apparatus must follow the spiral information. The system was designed to increase data density and random access was considered less important for current applications; namely

Device	Data Transfer Rate (Bytes/s)	Ave Seek Time (ms)	Ave Latency (ms)
4mm DAT h-scan	150-200K	500-2000	20
8mm DAT h-scan	250-550K	500-2000	90
CD-ROM, WORM	1-5M	80-100	10-20
R/W Optical Disk	1-5M	80-100	10-20
Magnetic Disk	1-5M	14-18	5-15

Table 3.2: Storage Device Data Transfer Characteristics

single user access to high fidelity audio or video. The seek times for these applications are considered acceptable, or at least tolerable by users of CD-ROMS.

Magnetic disks and read/write optical disks use constant angular velocity (CAV) technology where tracks are organised as concentric rings and divided into sectors. Using this method, drive speed remains constant and bit density increases toward the center of the disk. The advantage of magnetic disks is a lighter disk head assembly which takes a shorter amount of time to physically move during a seek operation. The popularity of optical disks is still likely to increase because of their huge capacities.

A theoretical discussion of magnetic storage technology is given in [**Wilson92**]. It outlines the performance issues for a typical Winchester disk and serves as a basis for maximising the data rate of this technology. Generally, the data rate is dependent on the type of controller attached to the disk subsystem. The rates shown in table 3.2 are typical of synchronous and asynchronous SCSI interfaces with synchronous devices tending toward the higher values and asynchronous devices near the lower end of the spectrum.

Aggregate capacity is an important requirement for continuous media since files in this environment are expected to be large. Table 3.3 illustrates file sizes at the data rates given in Table 3.1.

Uncompressed Pandora audio and video file sizes are included in the general table to illustrate an example of an existing system. The calculation includes packet header overhead and assumes a data rate of 25 frames per second. Initial experience with the Olivetti Video Repository shows a median multi-media file size of 4.99 Mbytes, corresponding to 24 seconds of audio and video.

The predominance of small files has influenced the design of most existing file systems. Optimisations in design including block size, allocation algorithms, and caching strategies, reflect observations made by researchers during the use of exist-

Service		File Size (Bytes)	
	(1 Second)	(15 Seconds)	(25 Seconds)
Standard Audio	8K	120K	200K
High Quality Audio	250K	3.75M	6.25M
Low Quality Video	250K	3.75M	6.25M
Television	4.25M	63.75M	106.25M
HDTV	17.5M	262.5M	437.5M
Pandora Audio	13.25K	198.75K	331.25K
Pandora Video	195K	2.93M	4.88M

Table 3.3: Continuous Media File Sizes

ing file servers.

Large continuous media files imply that the unit of allocation should be greater than a single block in order to minimise the disk transfer time for large disk transfers. In fact, it is essential to have the unit of allocation such that the time required to process the data is greater than or equal to the access delay and transfer delay combined. For example, a system may require on the order of a few hundred milliseconds to read enough data for several seconds of playback. An initial delay is unavoidable when reading the beginning of a file, but subsequent asynchronous disk accesses can be completed before the current file buffer is exhausted.

3.1.3 Memory, Bus and CPU

Memory and bus bandwidth are currently far greater than I/O subsystems. Memory provides about 50MBytes/s, if a 32 bit wide memory and 80 nanosecond access time is assumed. The use of high speed cache memory can improve this figure. A typical data bus can support transfer rates of 30MBytes/s. The TURBOChannel architecture is an example of a high speed bus. It defines a 32 bit, synchronous I/O channel that offers a peak Direct Memory Access (DMA) burst rate of 100 MBytes/s, although 78 MBytes/s is the maximum bandwidth achieved on current implementations. A more common example is the asynchronous VME bus which offers 16 and 32 bit data transfers at a maximum of 25 Mbytes/s. Other examples include the FutureBus (37 MBytes/s) and the MultiBus II (20 MBytes/s). Processor speeds have increased more rapidly than memory speeds and provide ample capacity compared to current I/O speeds.

The most important issue with respect to these resources is their efficient utilisation. Resource contention will be the dominant issue in a continuous media server. This

demands an effective management policy. For example, consider disk and network interfaces both performing DMA access and the CPU performing normal memory accesses. Only one resource may be permitted access to the memory via the system bus at any given time. The length of time each resource holds the bus influences the delay experienced by other resources in the system. Without a carefully managed resource allocation policy, this delay may become unacceptable.

It is desirable to keep the overhead of the controlling software to a minimum to achieve the maximum performance of each of the resources in the system. For this reason, a lightweight micro kernel supporting user space threads is advocated. In such a system threads represent independently schedulable paths of execution which share a common address space. Thread switching times are reduced by eliminating the overhead of saving and reloading entire process address spaces and context.

3.2 Quality of Service

Quality of Service is most often used in network terminology and the desire is to use it as a resource management tool within a computer network. A continuous media storage service designed with conventional technology manages the resources outlined in section 3.1.

Of these resources, service bandwidth provided to all clients is limited by the bandwidth of the least capable resource in the system. The elimination of one bottleneck may uncover another. The goal of QOS is to provide an infrastructure facilitating negotiation between client and server for acceptable service within the capabilities of a given system.

Since this negotiation will involve resources other than the network, an entity managing QOS must be aware of the current state and limitations of all resources at a node. The interaction of these resources will determine the QOS available to concurrent clients. Resource state information should be maintained in real-time to provide a QOS manager with accurate information regarding system utilisation. This suggests that operating systems should manage QOS or provide a lightweight interface to a QOS manager implemented outside the kernel. The interface is not specified here, but may be a traditional system call, an event based system, or an RPC mechanism. The system call and RPC scheme essentially provide ways to query device state. An event based system might be used by devices to update state information asynchronously.

3.2.1 Admission Control

Admission control is necessary when there is a desire to support multiple concurrent clients. A client wishing access to the file system presents a set of requirements to the server. The server decides whether to accept or reject the request based on its current ability to support the specified requirements. An incorrect algorithm may have one of two undesirable consequences. The system may be under-utilised and deny service to a client with requirements actually within the servers capacity. Providing the decision to accept the client is not too costly, the client should be allowed to enter the system. Alternatively, the system may be over-utilised, in which case the QOS may degrade below an acceptable limit for some client.

As previously mentioned, admission control may be achieved by maintaining state information on each resource in the system. Each resource may supply a value representing the state of that resource. All of these values together comprise a *vector* representing the systems' instantaneous state. The details of an algorithm are an open question, but it should operate on the state vector. This ensures that the algorithm behaves correctly when any one bottleneck is removed. For example, it is plausible that current network interfaces limit the throughput of a storage service. Should the network interface bandwidth increase beyond that of another resource in the system, an admission control policy based solely on network load would no longer be appropriate. A decision to accept a new client could result in some QOS degradation because another resource can no longer support the required data rate.

Timely server state computation is a requirement for a successful admission control policy. This limits the overhead of admission decisions. This can be accomplished by distributing the responsibility for state information to each resource driver. The driver can present an interface to query the device state, or may make *up calls* placing critical information into a specified data structure. The latter method has the advantage of rapid access to state information by an admission control process.

Many continuous media encodings are likely to exist in future and it may be difficult to describe the characteristics and requirements of each method. A common service class scheme may be advantageous in this heterogeneous environment. This is a flexible way to represent stream requirements since actual requirements can change without the need to modify the classing scheme. For example, [**Dixon92**] describes multi-service traffic by their bandwidth and burstiness[1], but other parameters may be added to a class without changing its representation.

A disadvantage of service classing is that the acceptance of a stream may not fully utilise the system. Classes will no doubt represent a range of bandwidth requirements and the upper bound of this range must be supportable when applying the

[1]The term burstiness is used to describe how the rate of transmission of data varies with time.

acceptance control scheme. It is possible to accept streams nearer to the lower bounds of a class and deny new clients access based on a calculation relying on the upper bound service.

3.2.2 CPU Scheduling

The primary function of CPU scheduling is to provide efficient and adequate access to the CPU for all threads. The isochronous nature of continuous media demands that threads which manipulate stream data behave in a time critical manner. The threads are scheduled periodically and must process data in an average time which is less than the thread period. For simplicity, consider a constant bit rate audio coding where audio packets arrive with a period of 4ms. A thread receiving the audio must on average, process each packet in less than 4ms for it to maintain an acceptable level of service.

Prioritisation of threads is essential to preserve an order of execution determined by the requirements of each thread active in the system. Consider two threads; one processing video and another processing audio. Video data is voluminous. Thus, one might assign a higher priority to the thread processing the video to avoid holding large buffers for longer than necessary. However, audio packets arrive more frequently and take less time to process. With video at the higher priority it is possible to starve the audio thread of CPU long enough to cause it to exhaust its buffer capacity. This phenomenon has been observed in both the Pandora Box [**Hopper90**] and the continuous media file server described in Chapter 5. Common scheduling algorithms include:

- First-in-First-Out (non preemptive): Threads are executed in order of their arrival on the run queue.;
- Shortest Job First (non preemptive): The thread with the smallest estimated run time is executed;
- Highest Response Ratio Next (non preemptive): Higher priorities are given to threads which have short execution times or threads which have long execution times and have been waiting for a long time [2];
- Round Robin (preemptive): This is a FIFO mechanism which allocates time slices to each process;
- Deadline (preemptive): Threads with the earliest completion time or the least amount of time to completion are run first;

[2]The specific calculation is priority = $\frac{servicetime+timewaiting}{servicetime}$.

- Multi Level Feedback Queue (preemptive): A set of priority FIFOs are provided. Threads which do not complete before their time slice are placed on the next lower priority FIFO. The lowest priority queue behaves like the round robin scheme.

Of these scheduling algorithms, only the preemptive policies are desirable because non preemptive algorithms may introduce unacceptable delay in processing times. A strict round robin scheme is difficult to tune. If a small time slice is used, too much of the CPU time is wasted in context switches. As larger time slices are used, unacceptable delay may be introduced to some threads. This policy degrades to FIFO behavior when large time slices are used. A multi-level feedback queue may be applicable, but hard real-time systems research suggests the use of a deadline driven algorithm.

In [**Liu73**] a dynamic deadline driven scheduling algorithm is described and proved, which achieves 100% processor utilisation before an overflow occurs. The environment consists of threads on a uniprocessor which are periodic. Using this algorithm, priorities are assigned to threads according to the deadlines of their current requests. A thread receives the highest priority if the deadline of its current request is nearest. Similarly, it receives the lowest priority if its current request deadline is the furthest.

The correctness of the algorithm is certainly not in question, but three assumptions are made which are subject to review when applied to continuous media services. The first assumption is that all threads are independent. The second assumption is that threads with deadlines are periodic and have a constant interval between requests. The final assumption is that the execution time for each thread is constant and does not vary with time. The second assumption is invalid when considering either jitter introduced by network transmission or a variable bit rate encoding in which the arrival period of data packets will vary. The third assumption may be invalid because thread processing time may vary with time; for example, consider the video processing thread which has exhausted its current disk extent. Reception of another video packet requires the thread to perform a disk allocation which will increase the processing time. The first assumption is not necessarily valid when considering multi-media synchronisation. Threads may be made dependent on each other after they have been synchronised. Although the analytic model may not apply directly, suitable approximations to thread periodicity can be made, placing bounds on processor utilisation.

The specifics of an applicable scheduling policy are an open question. However, there is a need for an operating system with real-time characteristics to provide the system response times necessary to service time constrained data.

3.2.3 Interrupt Processing

In common interrupt driven systems, interrupts occur and are serviced through a hardware priority mechanism without regard to the active scheduling policy. An interrupt preempts the currently scheduled thread and may spend too much time in the service routine, causing the thread to miss a deadline. The factors contributing to the delay include:

- Startup latency - The context switch to interrupt service routine;
- Service time - Time spent in the service routine;
- Return latency - The context switch to the currently executable thread.

Thus, the requirement for the guaranteed deadlines in a real-time system implies that a CPU scheduling policy take interrupts into account. The interrupt handler might include a specification for the amount of time allowable in the service routine. These time constraints may be integrated into the scheduling mechanism. [**Dixon92**] shows the effects of bounding network interrupt service times on stream jitter. The results are commensurate with the above requirements in that restricting interrupt service times decreases the amount of stream jitter.

3.3 Flow Control

The term *flow control* is used to describe the ability to alter the transmission characteristics of a data stream, e.g. the ability to increase or decrease the rate of transmission. There are two common ways to accomplish this goal:

- Rate based flow control
- Sliding Window flow control

Sliding window flow control is most often used for reliable transfer protocols, but is less desirable for continuous media applications because of the delay introduced at window nodes.

Rate based flow control mechanisms may be client based or server based. Client based flow control is implemented by requiring each client to request data from the server at a frequency determined by the client. Figure 3.1 illustrates a client of a video server that is viewing twenty-five frames of video every second, requesting a frame every twenty-fifth of a second. In this case, it is important that the client

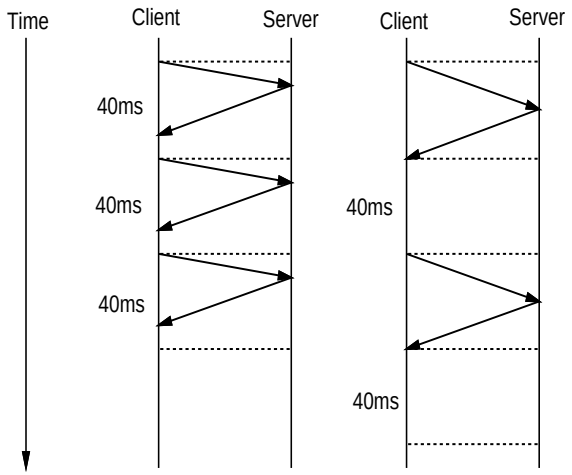


Figure 3.1: Client Rate Based Flow Control

make the request every twenty-fifth of a second, inclusive of the time to receive the video frame. The figure also illustrates a contrast with waiting one twenty-fifth of a second between frame receptions, which would reduce the overall frame rate. The server need only deliver a frame whenever one is requested. This scheme is commonly used in conventional data servers and works well because there is no time critical element to the data transfer.

Using the same example, a server based flow control mechanism works by having the client make a request to the server of the form: send twenty five frames every second. The client may renegotiate a different rate when desired and the mechanism of renegotiation may be in-band or out-of-band. If the server has no knowledge of the stream representation, such as frame boundaries, the client can express the request in the form of a number of bytes per second.

The client based approach is attractive because it allows the server interface to be simpler and there is no overhead associated with changing the rate of a stream. However, when servicing concurrent clients, it becomes more difficult for a server to maintain QOS to each client at peak loads. This is because the server has no *a priori* knowledge of stream behavior and can only predict load based on previous statistical behavior. Policing clients is also more difficult when clients do not adhere to their negotiated QOS. In contrast, the server based approach empowers the server with knowledge of current and expected load. Not only can this information be used to implement acceptance control, but clients may not alter their behavior without specific renegotiation.

A further disadvantage of the client based strategy is the overhead associated with making frequent requests. Typically, this might be done with an RPC mechanism or an unreliable request-reply protocol. In either case, there will be an increase in

traffic on the network and an increase in transmission delay associated with this scheme.

An advantage of the client based strategy is the greater flexibility in synchronisation of related streams. The client possesses the ability to read ahead maintaining its own synchronisation information. This is a further advantage when using node striping such as that described in section 2.4. Client requests for data to independent servers are synchronised by the client. A server based flow control mechanism using node striping would require a potentially complicated inter-server synchronisation scheme to maintain stream synchronisation.

The method used for flow control in [**Anderson91**], although it is not explicitly stated, is a hybrid between the client and server based approaches. Essentially, a client will request a continuous media stream at a data rate R and the server will begin to buffer the stream according to that rate. The client is then expected to read the stream from the server at rate R. If it tries to read at a rate greater than R, the requests may be ignored and the server suffers only the performance degradation imposed by repeated requests. If the client stops reading a stream or reads at a rate less than R, the server will eventually stop processing the stream. This method has the obvious advantages of policing client access and providing a rate control mechanism without explicit renegotiation.

3.4 Synchronisation

Synchronisation of related media streams is an important consideration for the integration of continuous media into the network environment. A file service should provide enough synchronisation support to minimise skew between streams and minimise single stream jitter, but it is the view within this dissertation that strict synchronisation is performed in a manner such as that described in [**Sreenan92**]. In that work, it is proposed that synchronisation be performed at the receiver or at a node close to the receiver. This ensures that synchronisation is maintained because the local delay and jitter characteristics are more accurately determined and controlled. This design facilitates synchronisation in both the local and wide area network environment.

Synchronisation requirements can be divided into two classes:

- Continuous synchronisation - a mechanism for synchronising inter-related streams;
- Event based synchronisation - implemented using a signalling mechanism.

Inband continuous synchronisation is aided by the inclusion of time stamps within

each data packet. Time stamps provide a strict time ordering of packets and local clocks are used to recreate this ordering from the time information provided. [**Nicolaou91**] proposes a scheme for stream synchronisation that relies on stream jitter characteristics. By monitoring stream jitter, a loss of synchronisation can be detected when jitter rises above a threshold. This method assumes that initial synchronisation is performed at the source. Initial synchronisation of different sources is more complex and the use of a synchronised clocks or an event synchronisation mechanism may be advantageous.

Another method of continuous synchronisation is to interleave related stream data onto a single stream and store them within one file. The main advantage of interleaving streams is the precise synchronisation achievable. However, this approach has several disadvantages. First, it requires that related streams originate from a single source and terminate at a single destination. Secondly, individual streams may have different delay and jitter characteristics and this scheme precludes the possibility of selecting the most appropriate channel for the transmission of each stream. In addition, removing jitter from each stream may be more complex when data has been interleaved [**Nicolaou91**]. Storing continuous media files separately provides the ability to synchronise previously unrelated streams.

Event based synchronisation is used to inform applications of asynchronous events. A *call back* mechanism is an acceptable method for achieving this goal. Notification of events can be made with acknowledgement, as in the RPC paradigm, or can be sent to an application without response. Timely invocation is essential in the multi-media environment, but some latency is always introduced by the underlying communication system.

Assuming that a server based flow control mechanism is used in a continuous media file server, inter-stream synchronisation does not have to be maintained during a record session. After initial synchronisation information is received, the server can accept and store data asynchronously without further attention to synchronisation since no presentation to a user is required. In this respect, the functionality provided by the server is asymmetric since synchronisation is only performed on playback of the related streams.

3.5 Summary

The special requirements for continuous media were outlined. Careful resource management is required of a system that is to provide clients with Quality of Service (QOS) guarantees. File server system resources such as the network, disk devices, memory, bus bandwidth and CPU were identified and their general capacities pre-

sented. The requirement for a lightweight multi-threaded kernel with real-time characteristics was specified and several possible CPU scheduling algorithms were reviewed.

The requirement for QOS to clients was discussed and some requirements for an operating system were outlined. CPU scheduling and interrupt handling remain important factors. In addition, QOS negotiation may be achieved through the monitoring of resource state information in conjunction with an admission control policy.

The advantages and disadvantages of three approaches to flow control were discussed, including a client based strategy, a server based strategy and a hybrid scheme.

The need for inter-stream synchronisation was presented. Some general methods for providing this functionality were discussed. Synchronisation in a continuous media file server is asymmetric in that the maintenance of inter-stream synchronisation need only be performed on playback.

Chapter 4

Design Considerations

The following is a discussion of general design considerations for a distributed file system. Traditional monolithic operating systems provide full file system implementations in the system kernel. Two closely related issues arise from distributing a file service. Distribution implies separating function onto separate processors or nodes. The consequence of this is to remove functions from the otherwise monolithic operating system leaving the operating system as a support system. This is the approach taken by current micro kernel architectures.

Section 4.1 discusses issues related to file system distribution. Section 4.2 outlines design considerations for specific file system instances. The concern is with low level issues of device management and file representation.

Finally, continuous media file design considerations are discussed in section 4.4.

4.1 Distribution

4.1.1 Naming

Naming describes the function of translating a unique identifier, often human readable, to the location of an object. A naming strategy may be object-oriented or hierarchical or both and provides the mechanism for locating local or remote objects. An important goal of naming in a distributed system is to achieve location independence for objects.

The above goals are achieved often by replicated directory tables facilitating easy access to information in a local area and naming system robustness. A problem

created as a consequence of table replication is updating directory copies. Inconsistency in the name database is the result of not being able to update all service directory table copies simultaneously. There is always delay in making the name space consistent after update.

Another central issue of naming is the provision of a scalable name space. A solution to this problem is proposed in [**Lampson86**]. A hierarchical name space is described in which there is no dependence on a well known root. This design feature provides a good way to provide an adaptable name space with decentralised control.

In the context of a distributed multi-media file system, the naming scheme may be identical to that used on any other component of the system. There is no special consideration needed except that the interface conform to the naming system chosen.

4.1.2 Access Control

Access control is used to provide security of information in file systems. The simplest method to providing some file protection is not to reveal file naming information. That is, since users do not know file names, they cannot access file data. This suffers from the problem that many file names may be guessed easily. There are two other methods for implementing access control; *Access Control Lists* (ACL) and *Capability Systems*.

The most prevalent method is the access control list which specifies domains and rights to an object associated with a domain. Anyone wishing access to a file will be authenticated by the operating system (or access control manager) against the access list. For example, in the Unix file system, three access domains exist for every file: the owner, a group and everyone else may be permitted read, write and execute access to files separately. The user and group identifiers are stored as part of the file meta data at the time of creation. A disadvantage of the access control list is the necessity for list management. This does not scale well without special attention. Furthermore, it can be inefficient to authenticate clients since lookup operations may span multiple control lists.

A capability system is analogous to a ticketing system where a client who holds a ticket (or capability) for an object is allowed access to that object. A capability consists of an object name and a set of access rights to the object. Although capabilities can be efficient in that they can be validated quickly, a capability system suffers from insecurity in that capabilities may be intentionally copied and distributed or they may be stolen. In addition, possession of an object's capability always allows access to the object. In short, these problems are free propagation and free access. Three schemes exist to improve this vulnerability.

The first scheme relies on the provision of a secure capability management system such that client capabilities are not accessible by the clients themselves [**Rajunas86**]. The second scheme [**Karger88**] allows capability copying, but prohibits free access. A client must authenticate itself with a server before access to an object is granted. This policy entails considerable overhead in that each access requires a client authentication. Lastly, [**Gong90**] eliminates both problems by requiring capabilities to be identity dependent. That is, a capability is issued only to the entity which may exercise the access rights to an object.

4.1.3 Reliability and Integrity

Perhaps the most important aspect of a file system lies in its ability to provide reliable stable storage to clients in the event of failure. The reliability of a system may be defined by the success with which its actual behavior is consistent with its specified behavior. Reliable communication with hardware is often checked by the inclusion of Cyclic Redundancy Checks (CRC) or Error Correction Codes (ECC). Another illustration of reliability is reliable software communication which can be achieved by either a Remote Procedure Call system (RPC) implementing message retransmission in the event of failure, or a reliable transport protocol such as TCP.

A measure of reliability is expressed by the Mean Time Between Failure (MTBF).

$$\frac{1}{MTBF}$$

is a measure of the frequency of failure. A system with multiple components, each of which has its own MTBF, has a frequency of failure equal to the sum of the component frequencies of failure. i.e. if $M_1, M_2, \ldots, M_n$ are the individual MTBFs, then the system mean time between failure is described by

$$MTBF_{sys} = \left[\frac{1}{M_1} + \frac{1}{M_2} + \ldots + \frac{1}{M_n}\right]^{-1}$$

The reliability of a continuous media storage server is likely to be most affected by the use of multiple disk drives. This will be discussed in more detail in section 4.2.3.

The integrity of a system describes its ability to keep data in a consistent and correct state. File system integrity on Unix may be checked with the system *fsck* command, repairing any inconsistencies it finds. This is necessary because it uses large volatile disk buffers to improve the performance of the general file system. Inconsistency may result from a system crash which causes the loss of important volatile meta data.

Transaction processing techniques can increase the integrity of a system, but adversely effects the performance due to the overhead associated with writing through to stable storage.

4.1.4 Availability

Due to a network partition failure or file server crash, a network file system may be unavailable for some period of time. It must provide a robust way of handling disconnection or error. The ability of clients to continue useful work in the face of short term error is described as the file service availability. This can be expressed numerically by the systems Mean Time to Repair (MTTR) and the MTBF. The MTTR may be defined to be the elapsed time between the point of failure to the point of repair. In general, the availability of a resource can be defined as a probability as follows:

$$A = \frac{MTBF}{MTBF + MTTR}$$

The combined probability of availability in a system with multiple resources which are all are required is $A_{sys} = A_1 \times A_2 \times \ldots \times A_n$.

In a file system greater availability of data is usually accomplished by some form of replication; either by client caching of remote files or replication of files on multiple file servers. In this case, the calculation for file availability is performed in parallel. For example, if either one of two file copies may be used in a system the overall availability calculation becomes $A_{sys} = 1 - (1 - A_1)(1 - A_2)$. The disadvantage of replication is the necessity to maintain consistency between multiple copies of a file. In a continuous media file system replication is prohibitively expensive because of the large file sizes.

4.1.5 Scalability

The scalability of an architecture is determined by its ability to maintain an acceptable level of service with an increasing number of clients.

The primary characteristic of current scalable file systems is judicious use of client caching. The applicability of this feature depends very much on particular data access patterns. Caching reduces the communication traffic between client and server machines and distributes real work among many clients. In fact, [**Nelson88**] reports that caching can reduce server load by 50% and network traffic by 90%. [**Burrows88**] presents the MFS as a scalable network file system which addresses the problem of cache consistency. To support concurrent fine granularity write shar-

ing and propagation of updates, a token passing scheme is used together with a client *callback* mechanism. The overheads of this scheme are negligible because write sharing is rare and the correct behavior of the system is an advantage when conflicting updates are performed. Further optimisations follow from access characteristics of a given file service. Off-loading work to individual client machines is the primary method supporting distribution.

As discussed in Chapter 1, client caching of continuous media data is inappropriate. A more important factor in the scalability of a continuous media file system is the ability to integrate separate file system nodes into one logical file system. Continuous media nodes may only support a limited number of concurrent clients. As both network and disk bandwidth increase, this number should also increase, but a more dramatic increase in service may only be achieved by adding new nodes to the system.

4.2 File System Instances

Each file system instance must maintain information about files in a precisely defined fashion. The many ways of organising a system depend on the performance criteria outlined for the design. Certainly, some of the features described in the previous section apply to individual file system instances, but within the context of this dissertation, a separation is made between issues in section 4.1 and this section to simplify the low level design of file systems with different performance constraints.

A file system is responsible for presenting a device independent way of accessing data. It should provide an interface which gives users a logical view of their data and allow them to structure files in a manner most convenient to the application.

The file design process begins by gathering the file requirements such as the type of file data to be stored, the expected file sizes and security constraints. The resources available should be outlined and their performance characteristics clear.

A first file design is based on the access patterns expected. If this design meets the desired performance criteria, then a check against other criteria such as security or naming might be made to finalise the process. If at any point the system does not meet the original goals, then re-evaluation of the initial requirements is in order. The following sections discuss some design considerations for file system implementation.

Section 4.2.1 discusses the characteristics of different file organisation strategies and common environments in which they are applicable. Section 4.2.2 looks at typical file system layout policies. That is, how file system data structures are organised on disk and how references to file data are performed. Section 4.2.3 concentrates on

physical disk management and performance improvement of disk system.

4.2.1 File Organisation

The access characteristics of files will determine the file design. There are a few popular basic schemes in use depending on various file access requirements. These include:

- Sequential Files;
- Indexed Sequential Files and Indexed files;
- Direct Files;
- Log Based Files.

The sequential file is no longer commonly used. The design was well suited to batch oriented data processing where records are added periodically and processed sequentially. One block follows another and are stored and retrieved according to their physical order. A simple way of expanding a file is to include a pointer to the next block in the current block, creating a linked list of blocks in the file. Unfortunately, random access with this scheme is expensive, since a potentially high number of accesses must be performed to read the desired block.

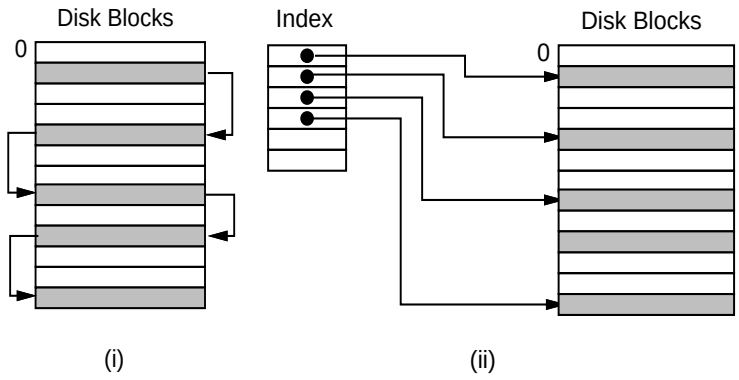


Figure 4.1: Sequential (i) and Indexed Sequential (ii) Files

Indexed sequential files are commonly used in mainframe environments. They were designed to improve the performance of random access to sequential files. In large database systems, index keys are held within each sequential record as required, but in popular workstation file systems, the index is often kept separate from file data. The following section describes some methods used to organise these indexes. Indexed files are used in such applications as airline reservation, bank transaction,

and more general database systems. Data in these systems is rarely processed serially and timely access is critical. The compromise in performance for serial access is outweighed by the flexibility in record format and placement.

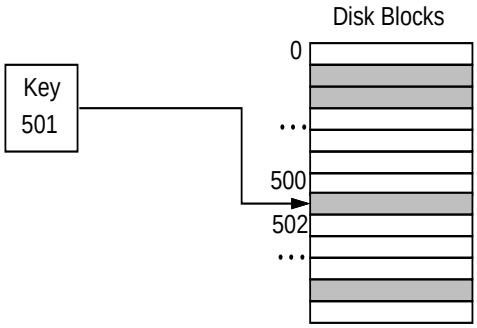


Figure 4.2: Access to a Direct File

Direct files are frequently used for directories, tables, schedules, etc. These applications require small and fixed record sizes where fast access is essential. Figure 4.2 illustrates access to a direct file using a single key used for retrieval of data. This makes access simple and efficient. Hashing techniques are often useful in locating data in direct access files. This method generally suffers from dependence on hardware, because the key is usually a hardware address, such as a disk block number.

To improve the write performance of file systems a log structured file may be used. Using this technique, the file systems only representation of a file is an append only log. In fact, the entire file system is a circular append only log. The main advantage is to eliminate all seeks during write operations. Other advantages include faster crash recovery, file versioning and automatic clustering of files written at the same time. The main disadvantage is the performance overhead of traversing a log to *replay* a file during a read operation. A proposed solution in [**Ousterhout84**] relies on indexing different parts of the log with *log map* blocks to facilitate random access of the log and a large cache to satisfy most read operations.

A file system must support one or more of the above file types. Often, only one type is made available to the user, but the use of more than one type is essential to providing an efficient service. For example, a direct file might be used to access user file indexes, as described in the following section, in order to expedite access to the indexes themselves, whereas user files would be accessed from the index information. MVS supports the first three of the file types to the user and is tailorable to specific applications. It is noted for its flexibility, but at the cost of complexity needed to specify I/O requests.

4.2.2 File System Layout

The file system maintains a sequence of allocated disk blocks for each file. Current popular operating systems organise these lists in various ways. MSDOS organises a free list of blocks in a *File Allocation Table* (FAT) [1] at the beginning of the disk. The FAT contains one entry for each disk block. The first disk block allocated to a file is contained in its directory entry and is used as an index into the FAT to locate the next disk block for the file, facilitating block chaining in a central map. Figure 4.3 shows three files mapped inside a FAT. This approach was designed for small systems and does not scale well when considering systems that support large devices and large files. The FAT quickly becomes too large to manage efficiently. However, for small disk devices, it is an efficient way of managing space. OS/2 also uses the FAT scheme, but it partitions disks into logically smaller disks each containing a FAT to maintain compatibility with MSDOS.

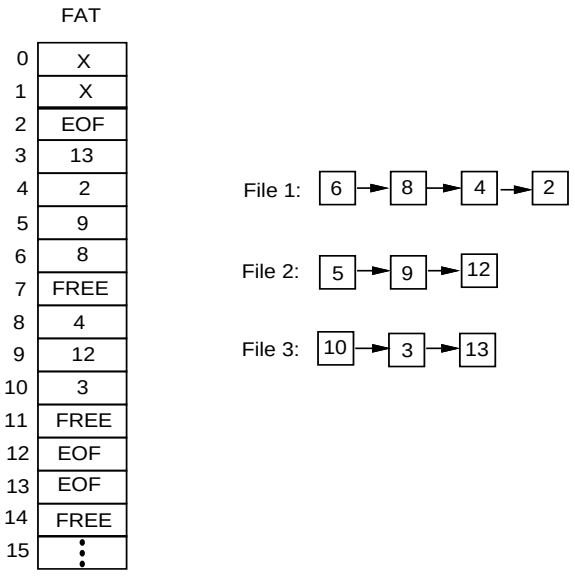


Figure 4.3: Three Files in a File Allocation Table

The Unix file system is more complex and offers a robust scalable interface to a hierarchical file system. In addition, device interfaces become part of the file system abstraction. The system uses a *super block* to partition the disk into a variable number of *cylinder groups*. Each cylinder group contains a copy of the super block followed by a *cylinder block*, an *i-node* table and an array of data blocks. Figure 4.4 illustrates the layout of a Unix file system from the top view of disk platter. Cylinder group information is carefully placed on the disk to avoid system wide failure in case of a single head crash.

I-node tables hold information about each file or directory in the cylinder group.

[1] This is also commonly referred to as a Table of Contents.

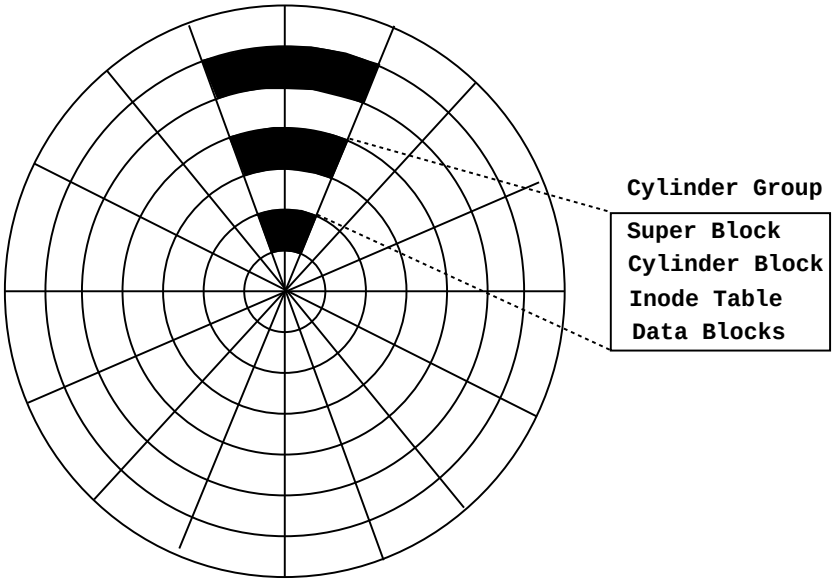


Figure 4.4: File System Layout in Unix

This abstraction gives a flexible way to represent a variety of different file types [2] by building on this single abstraction. It is the I-node entry that holds file meta data, such as the file type, the owner, creation time, the file size and a list of disk blocks allocated to the file.

This approach is preferable to the use of a FAT because it localises block lists per file. Accessing a file using a FAT scheme may involve holding the entire FAT in memory or paging the FAT as necessary.

4.2.3 Disk Management

In addition to a sensible file system layout, an effective space management strategy is required to meet the demands of a particular design. Space may be allocated to a file in contiguous blocks, groups of contiguous blocks, or independent and individual blocks.

Contiguous allocation implies that a user specifies in advance the size of the area needed to hold a file. This allows blocks to be physically allocated adjacent to one another and is efficient for data access speed. A disadvantage of this scheme is that it is difficult to expand or contract files without considerable overhead or waste. A further disadvantage is that periodic compaction is necessary to avoid unwanted disk fragmentation. One advantage of contiguous allocation is that random access

[2] Unix provides abstractions such as: byte stream files, file linking, directories, character and block devices and sockets.

to file data is easily calculated from the starting block number.

Non-contiguous allocation is preferred for its ability to support dynamic allocation. The allocation unit may be individual disk blocks, a fixed number of contiguous blocks also called blocks or clusters, or *extents*. Extents are characterised by a variable number of contiguous blocks. One important point in the use of extents is that the representation of an allocation unit must change. Instead of having a single value identifying the block number, a starting block number and a length are required to identify the extent. This makes random access in a table based (i.e. FAT) representation more costly in terms of space. However, in both cases, a search is required for random access to file data. Usually, the search is linear, but a faster algorithm such as a B-tree search may be used.

Block management can be accomplished by using a free block list or a bitmap (one bit to represent each block). The bitmap approach requires less physical memory and is preferred in a system that can hold the entire bitmap for the disk in memory.

Disk Scheduling

In a simple system, disk requests are serviced on a first come first serve basis. This is perhaps the most fair method of disk access, but when the request rate increases, this policy can introduce long waiting times. This scheme also introduces a random and non-uniform seek distribution on the disk. The conflicting goals of disk scheduling are to maximise disk utilisation, minimise the system response times and the variance of response times. Most scheduling policies concentrate on minimising seek times for a set of pending disk requests. Although rotational latency optimisation is also a consideration of scheduling policies, new disk technologies reduce rotational latency by introducing track buffers into the drive controller[3]. Popular scheduling algorithms include:

- First Come First Served (FCFS);
- Shortest Seek Time First (SSTF): The request with the least physical distance to travel from the current position is serviced next;
- SCAN: Disk requests are serviced in one direction until there are no more requests in that direction. Then requests are serviced similarly in the opposite direction.

[3]Block interleaving is another method of reducing rotational latency when the disk rotates faster than the controller can read the next block. This is done by physically separating logically contiguous blocks by one or more blocks. Again, the effect of this technique is reduced by track buffering.

- C-SCAN: This is a circular scan algorithm. Instead of changing direction at one end of the disk, the system begins again at the other end;
- N-step SCAN: The disk sweeps back and forth like the SCAN method with the addition that new requests issued are segmented into subqueues of length N. Subqueues are then processed using the SCAN policy before stepping into the next queue;
- Eschenbach Scheme: A modification of the C-SCAN method such that every cylinder with a queue may be allotted 1-k rotations, where k is the number of tracks on the cylinder. More than k requests will not be serviced in the current cycle.

As previously mentioned the FCFS policy is not advantageous in a real-time environment. The SSTF policy offers high disk utilisation by maximising the amount of time spent in disk transfer and the SCAN algorithm offers a better service distribution over SSTF. C-SCAN improves the variation in service time of the SCAN policy.

All three algorithms, SSTF, SCAN and C-SCAN may suffer from unfair access to the disk. For example, if the request rate for a particular block of a cylinder is high, these policies will tend to monopolise the entire device. The N-step SCAN policy improves upon these three in that it guarantees access to each queue in a cycle. The Eschenbach scheme is effective when the request rates are very high and was designed for message switching systems. This algorithm provides a high utilisation rate at the expense of waiting times and queue lengths. It also does not guarantee a particular level of service.

While it is important to understand the advantages of disk scheduling, it is not commonly used in current operating systems for the following reasons. Seek times are not linearly proportional to the distance the disk arm assembly must travel. The initial acceleration time and deceleration time are important factors in the overall seek time. Thus, minimising the seek distance by reordering requests may not have a great impact on performance. Furthermore, many large systems choose to reduce request queuing time and disk utilisation by increasing the number of disks, rather than minimising seek time on a per disk basis.

However, in the continuous media case, the disk request rates may be high, where the benefits of disk scheduling are noticeable. A comparative analysis of the above disk scheduling policies is given in [**Teorey72**]. These results indicate that at relatively low loads the SCAN policy is best, but at loads where disk requests exceed 30 per second, a switch to the C-SCAN policy out performs other policies. Of course, the point at which the scheduling policy is switched may vary for another disk configuration.

Multiple Disks

A suggested approach to improve I/O throughput is to use file interleaving (or striping) on an array of disks. Interleaving may be done synchronously [**Salem84**] or asynchronously [**Kim86**]. It is used to increase throughput by distributing data over multiple disks. The two advantages of interleaving are: 1) it facilitates parallel access to files and 2) it provides the means to perform disk load balancing.

Synchronous striping requires that all disks be controlled by a single controller, so that disk arm positions and rotational synchronisation is maintained. A block of data is divided into n subunits and each written to one of the n disks. The size of the striping unit may vary from one byte to an entire block. A system with n disks reduces the data transfer time by a factor of $1/n$. The advantage of this approach is that the access delay is the same as for a single disk.

Asynchronous interleaving does not require synchronised control of disk machinery. Disks act as independent servers and blocks of a file are distributed across the n disks in the system. The block sizes may vary on each disk. The primary advantage of asynchronous striping is that of disk load balancing. During periods where the request rate is high, a policy of using the least loaded disk may be used. The worst case access delay for asynchronous interleaving is n access delays. An analysis of asynchronous interleaving [4] gives the expected added access delay as $\sigma\sqrt{2logn}$, where σ is the standard deviation of the mean access delay of each disk and n is the number of disks.

Synchronous striping exhibits obvious advantages over the use of independent disks and asynchronous striping. The above studies indicate a significant improvement in performance for large block transfers. However, there are a number of considerations which may preclude its use. First there is a requirement for specialised hardware. Second, it becomes more difficult to add more disks to a system. File blocks must be redistributed to take advantage of the added parallelism. Alternatively, the assignment of disk striping groups can be made so that new disks are accessed separately, but then this is not a strictly synchronous scheme.

A common criticism of both types of interleaving is a decrease in reliability since each disk can fail independently. From the previous reliability discussion, the $MTBF$ of an n disk system is $MTBF_{singledisk}/n$. A description of using redundancy in arrays of inexpensive disks (RAID) to overcome this problem is given in [**Patterson88**] and [**Chen88**]. A range of redundancy schemes from RAID level 1 to RAID level 5 are detailed and does not warrant reproduction in this text. However, it is useful to point out that a detailed analysis of RAID reliability is presented in [**Shultze88**], showing that the $MTBF$ of a RAID may be greater than an individual conventional

[4] The full calculation may be found in [**Kim86**].

disk. Schulze calculates the $MTBF$ of a 49 disk RAID using parity groups at 55,000 hours. This is compared to individual $MTBFs$ ranging from 30,000 to 50,000 hours. Of course, this result is achieved at considerable cost in providing hardware support.

Performance Optimisations

The most common technique in both a local and distributed environment to increase performance of a file system is the use of a data cache. Using this scheme, recently accessed blocks are kept in main memory. The observation that the same block is likely to be accessed again soon reduces the number of disk accesses to satisfy disk requests. A block cache consists of a finite amount of memory and several techniques may be used to manage blocks within the cache. These can include a first in first out policy or a least recently used algorithm (LRU). Usually, a modified LRU approach is used which takes into account the importance of the block to file system consistency because writing critical blocks quickly reduces the probability that a crash will leave the file system in an undesirable state.

Attention to the placement of meta data and block allocation strategies is also important in reducing the number of seeks to the disk device. For example, multiple copies of frequently accessed file system data can be placed on the disk to reduce the average time of access to that data. This is more useful for read only data since there is no danger of creating inconsistency as there is with multiple copies of read/write data.

During normal usage, disk space becomes fragmented which may impair the performance of the system. Disk reorganisation can be performed periodically to compact files into contiguous disk blocks and to coalesce free disk areas.

A new approach to disk performance improvement implemented in the disk controller is discussed in [**English92**]. Two data structures are kept in the controller memory: an indirection table and a bitmap of available blocks. The indirection table stores the addresses of the physical segments indexed by logical block address. It is used to facilitate block translation for read requests. The free map is used to find a block that can be reached quickly from the current head position on a write request. Essentially, this scheme optimises disk write requests by allowing the disk controller to perform the placement of blocks of the physical device. The simulation results indicate a performance increase of 130%-164%, but reports a decrease in read performance between 8%-18%. Using this scheme, read performance only benefits if the read request is in the direction the disk head is travelling. Otherwise, read performance suffers from the necessity to stop the head and accelerate it in the opposite direction.

4.3 Concurrency Control

Concurrency control is necessary when multiple independent threads (or paths of execution) exist and must share common resources. It is used to prevent one thread from modifying shared data during the time another thread depends on that data, as this might cause an incorrect result. Mutual exclusion is used to prevent simultaneous execution of program critical sections and may be implemented using a variety of methods including: locks, semaphores, monitors, and eventcounts and sequencers.

The file system designer is faced with the decision of what the granularity of concurrent access to files will be. This may range from access to a single byte within the file to access of the entire file. A relationship exists between the amount of data locked and the amount of time associated with that data. That is, locking one disk block of audio excludes other threads from accessing this data for the duration of its playback.

4.4 CM File Design

Applying the approach to file design described in section 4.2, the first goal is to define the access characteristics of a continuous media file. Sequential access is the most likely access pattern. However, several other access patterns may be desirable when viewing digital video. A fast forward mechanism might be implemented by accessing every other frame, or every tenth frame. Some amount of random access is perhaps desirable for viewing portions of a video file. Reverse playback may also be a consideration.

Sequential access suggests a measure of contiguous allocation to minimise the access and transfer delays from disk. The desire for random access and extendible files implies the use of extents and an indexed structure to map extents into a file.

There are several considerations to be addressed when choosing extent sizes. The tradeoff between small and large extents depends on the timing requirements of the data within the extent. A thread processing packets at a high frequency experiences more jitter as the extent size increases. This is because the disk device requires the data bus for a longer amount of time during some phases of bus access, denying a high frequency processing thread access to the bus. On the other hand, a small extent size increases jitter to all threads in the system since more frequent access to the disk is required. With low frequency processing threads, the effect of the extent size is less noticeable because the disk controller may request the bus for relatively short periods of time before relinquishing it to the host processor. It

then renegotiates access to the bus to complete the transfer. If the data transfer is performed by the host CPU, then a large extent size will affect at least the maximum jitter experienced in the stream.

The degree of concurrency is another factor determining the extent size. A larger extent size increases the delay between successive requests for each stream, but with a high degree of concurrency, threads may have to wait for longer periods of time because the disk is busy servicing each large request. Furthermore, large extent sizes may increase the internal and external disk fragmentation. Internal fragmentation occurs when an extent is not fully utilised. External fragmentation is characterised by noncontiguous disk free space which, if not periodically coalesced, could not be used to allocate large extents.

4.4.1 The Data Path

Of particular importance to the performance of a network based continuous media file system is the data path a packet must follow from the network to the disk. Typically, a packet will arrive on the network interface and the data will be transferred to memory via the CPU or a DMA controller. Once the packet is in a network buffer, it may be transferred to disk. The most efficient path does not involve a data copy from the network buffer to the disk device. That is, the network buffer is passed directly to the disk controller. This is difficult to achieve unless the network packet size is approximately the size of a disk block. A small packet will waste disk space and larger packets will need to be divided into disk blocks.

The total number of copy operations necessary is dependent on the host operating system facilities. For example, implementing a user space server on Unix requires at least two data copies. One to transfer a packet from the kernel network buffers to a user space buffer and another to copy the user buffer to a kernel disk buffer.

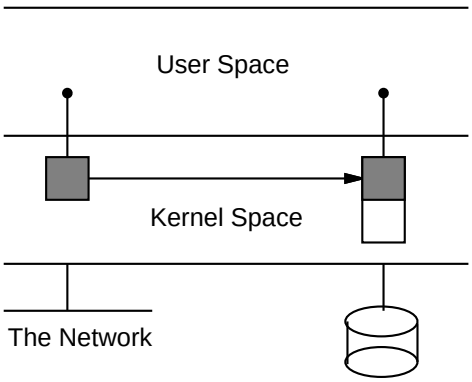


Figure 4.5: A Fast Path From the Network to Disk

WANDA provides a general I/O buffer interface such that a user process acquires a buffer from the kernel and may pass this buffer without a data copy to another device or the network. Essentially, the interface implements shared memory with the added constraint that only the owner of the buffer may write to its data area. Figure 4.5 illustrates kernel buffers owned by a user process. One data copy is involved with processing different sized network packets into fixed sized disk buffers. Transferring a buffer to another process or device involves the transfer of ownership only and does not involve a data copy.

4.5 Summary

Some general considerations for the design of a distributed file service were outlined. The functionality of a file system was divided into elements of distribution and local file management instances. Distribution includes issues such as naming, access control, scalability, reliability and availability. The solutions to these issues have little impact on the design of a continuous media file system because they are likely to be the same as those used in a traditional file system architecture.

The issues of specific file system design were presented. These include file organisation policies, file system layout on disk, file representation and disk management schemes.

Some design considerations for continuous media files were discussed. Allocation of contiguous disk blocks, called extents, was described with an indexing mechanism to facilitate random access to files. Issues for the determination of extent sizes were also outlined.

Chapter 5

Implementation

This chapter outlines the design of a prototype Continuous Media File Service (CMFS). The approach contrasts with some other methods presented in Chapter 2 in that a specialised service is provided. This is motivated by (1) the desire to use fewer components, reducing the cost of a system and (2) to reduce the overhead imposed by a general purpose file system. It should be noted that the CMFS represents a software solution only and is designed to run on general purpose hardware.

The design decisions made for this prototype reflect the differences from a traditional file service required for the provision of a continuous media file service. Many of the solutions to the design considerations presented in Chapter 4 are the same for the continuous media case. The CMFS design addresses the central issue of providing concurrent access to continuous media files in real-time over a network. However, nothing in the design precludes the use of the CMFS on a single machine where continuous media are transfered directly to a multi-media device.

5.1 Architecture

As previously mentioned, there are features of a distributed file system which are not dependent on the type of data stored. Specifically, implementations for naming and access control may be consistent with that of a conventional file service in a continuous media server. For example, a directory service separate from other file system services is presented in [**Seaborne89**]. The proposed continuous media file server may be viewed as another instance of a Low Level Storage Service (LLSS) within the hierarchical model of a distributed storage service presented in [**Bacon91**]. The two level storage service hierarchy includes services with different requirements.

Two low level storage servers illustrated in Figure 5.1 are examples of different ser-

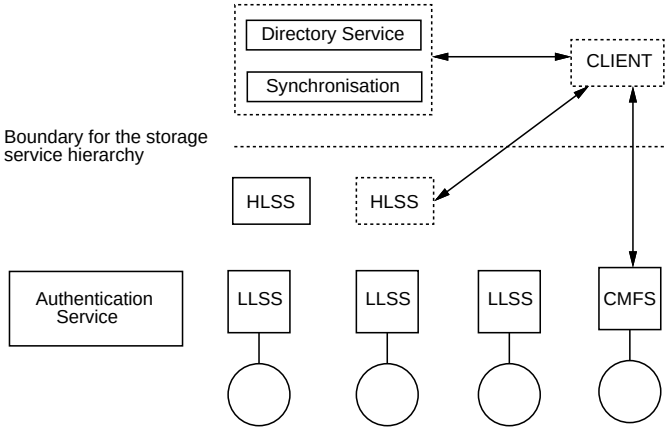


Figure 5.1: A File Service Hierarchy

vices; one supporting non real-time access to byte sequences; another supporting real-time continuous media streams. The higher level storage services provide additional functionality such as a directory service, access control, and synchronisation support.

Client programs may include database applications, traditional file systems like Unix, mail systems, and multi-media systems. Each uses higher level services provided, but can choose the lower level service most appropriate to the application and access it directly for efficiency.

5.2 Interface

The CMFS interface is based on the ANSA RPC system [**ANSA89**]. An instance of the server registers with a Trader as described in section 1.3. A client program contacts the Trader to determine a file server location and creates a binding with the service to gain access to files. This is analogous to mounting a network volume, although no explicit mount point exists within the local system. This scheme serves as a simple naming service for server nodes.

Differences in the CMFS interface from a conventional file server follow from the temporal characteristics of continuous media. Read and write operations no longer fit the access paradigm when using a server based flow control mechanism. The addition of play and record functions provide the abstractions for continuous access. However, both read and write operations are included to support a client based flow control mechanism and non real-time access to files. The latter is useful when considering file editing, replication, migration, or back up.

There are two notable consequences which follow from discussing the differences

between read/write operations and play/record operations. The first is to point out the difference in data path needed when using a server based flow control mechanism by contrasting read and play operations. The second is to distinguish between file operations and session operations. A read operation, in an ordinary file system, generally includes a buffer address as a parameter so that data may be returned to the client. The semantics of a read remain the same locally and across the network. A play operation requires an existing connection with a sink across the network for delivery of the data. Consequently, the open operation includes parameters for connection setup to or from a continuous media device. The result of an open is a *Session ID* rather than a file handle. In this way, operations on the connection are distinguished from operations on a file. For example, a pause operation manipulates server threads accessing a continuous media file and is not a direct file operation. The following is a description of the CMFS interface broken into the two categories mentioned. Only a general description is given as the specific implementations may require different parameters than those given in the CMFS prototype.

File Operations

Open/Close The open call creates a new file or opens an existing continuous media file. The parameters include an owner id, access mode and a file type. The close operation flushes any file buffers in the receive or transmit queues and writes file meta data to non-volatile store.

Delete Removes a file and returns its resources to the system.

Stat Returns static information about the file such as the file type, access mode, size, owner, time of creation and the current position within the file.

Read/Write/Seek These are standard file operations and behave similarly to conventional data servers in the CMFS.

Session Operations

OpenSession/CloseSession OpenSession establishes a connection between the file server and a multi-media device. Session state information such as play, record or pause is maintained in the session structures along with synchronisation information. CloseSession disconnects the multi-media device from the file server.

Play/Record/SetSpeed/Pause These operations are provided for the server rate based flow control mechanism as described in section 3.3. The speed and pause operations allow rate renegotiation.

Synch This operation facilitates synchronisation between separate continuous media files. Session groups are formed from one or more session IDs. All subsequent session operations on a session group ID are applied to each session in the group. Essentially, this allows fine grain timing between sessions at the

server. For example, the server will organise a common absolute starting time for each session in a group when issued a play command.

5.3 Disk Management

This section describes the disk usage in the CMFS prototype. Current disk interface speeds, outlined in Chapter 3, out-perform popular network interfaces. The Etherphone project used 1.5 Mbit/s and 3 Mbit/s Ethernets for experimentation. ISLAND experiments were performed using a 10 Mbit/s Cambridge Ring (CR), where the point to point bandwidth for a single station on a single slot ring is 1.3 Mbits/s [**Temple84**]. Thus, maximising disk performance was not a dominant issue in these projects. This is the primary reason for the success of continuous media file servers which use a general purpose file system.

The following sections describe the disk layout chosen, the allocation strategy and disk scheduling used.

5.3.1 File System Layout

A simple layout is used to manage each disk. Each disk is formatted using the largest block size available (4096 bytes) in anticipation of large file sizes. A block allocation bitmap is located at the head of each disk and is loaded into battery backed RAM (or non-volatile RAM) when the file system is booted. One disk is designated the system disk and holds the inode table and all indirect inode blocks. The term inode is borrowed from the Unix operating system. It refers to index nodes that describe file characteristics and contains a mapping of disk blocks holding the file contents. For simplicity, only one inode table is provided and its size is statically determined. This has the advantage that the entire data structure size is deterministic and the system designer can provide the required amount of non-volatile RAM cache. The disadvantage is the difficulty with which expansion of the maximum number of files is achieved.

The use of non-volatile RAM offers one significant advantage. It eliminates the disk overhead associated with keeping the file system data structures on disk in a consistent state. Critical information is written to disk from non-volatile RAM during periods of low load. Thus, special priorities for disk blocks holding file system data are not required and the file system integrity can be maintained in the event of a system crash. In addition, the location of inode blocks becomes relatively unimportant, since all inode blocks are stored in the non-volatile ram and are written to disk only at periods of low load.

5.3.2 Files

File allocation lists are held in an inode structure shown in Figure 5.2. The similarity to the Unix inode reflects the design goals of mutable files with low overhead random access. The viability of this approach relies on the number of extents per file being relatively small because the overhead of accessing indirect blocks increases as the number of extents increases. This approach is justified by two points. First, initial observation of the Olivetti Video Repository reports a median file size equivalent to 24 seconds of data. This may fit entirely in direct extent blocks depending on the extent size chosen. Secondly, the overhead is incorporated into a non real-time operation, such as an open call which does not affect the playback of the file extents.

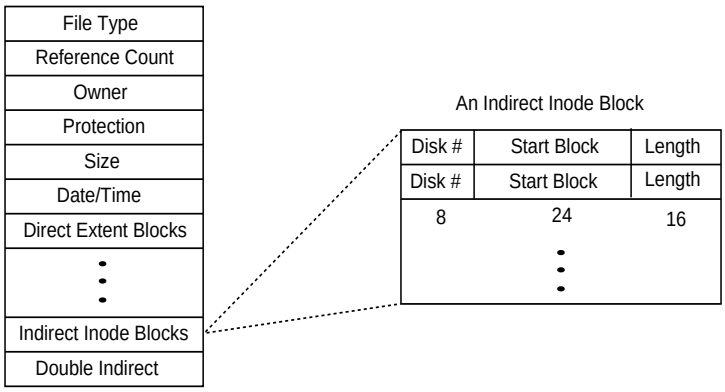


Figure 5.2: CMFS File Inode Structure

Indirect Inode blocks are always allocated on the system disk and are represented by 32 bit quantities in the inode structure. Direct extents are represented by three values in a 48 bit quantity:

- An 8 bit disk identifier;
- A 24 bit starting block number [1];
- A 16 bit extent length field (in blocks).

A file typing mechanism is used for flexibility in the representation of files and to distinguish the extent sizes for files with different timing characteristics. Three types of files are implemented: (1) audio, (2) video and (3) data. A reference count included in the inode structure is provided to facilitate sharing of files so that a delete operation will not remove the file until the reference count is zero. Higher level server functions must manage this effectively to provide support for this scheme.

[1] This yields 16,777,216 possible block addresses on a single device. There are 82,214 total blocks on a 380MByte drive formatted with 4KByte sectors.

Space for the owner of the file, the date and time of creation are provided in the implementation. A protection information field is also included, but no scheme is implemented. Although security is an important consideration in file system design, a protection mechanism need not be different from other file system implementations when considering access to continuous media files.

One 4096 indirect byte block can hold 682 extent entries. One double indirect block describes 87,296 extents. Simple arithmetic yields a total of 87,984 extents per file. Although the number of extents is fixed, the maximum file size varies with the extent size chosen. Hence, this is not a severe limitation.

5.3.3 Allocation

Although disks are managed at the block level, file allocation granularity is larger. A file allocates variable sized extents, such as those described in Chapter 4. For experimentation purposes a video extent size of 90 blocks (368640 bytes) and an audio extent size of 24 blocks (98304 bytes) was chosen. Pandora video arrives at 92,000 bytes/second and one extent of 90 blocks at 4096 bytes/block allows 4.0 seconds of video storage. Audio arrives at 24,000 bytes/second and one extent holds 4.1 seconds of audio. The extent sizes chosen reflect the median length of 24 second file sizes found on the Olivetti Video File Server in that this amount of audio or video data may be referenced entirely by direct extent values in the inode structure. Disk allocation is accomplished through a first fit algorithm. To avoid external disk fragmentation, periodic disk compaction is used to coalesce free disk extents. Internal fragmentation is reduced by shortening the last extent in a file. The result of any edit operations may increase the amount of internal fragmentation. This issue is not directly addressed in the CMFS. Edit operations are not performed in real-time and once an edit is complete, the server can rearrange the file extents in such a way that the real-time requirements can still be met. The CMFS design does not limit the use of this approach.

File inodes provide six direct access extents. Both indirect and double indirect blocks are provided to support large files. With the extent sizes chosen above, a video file can be 3.24×10^{10} bytes long (97.9 hours) and an audio file 8.65×10^{9} bytes (100.1 hours). If a larger continuous media sequence is desired, a mechanism for chaining files together for playback or record is feasible, but is not implemented in the prototype.

Index blocks are allocated using a First-Fit algorithm as data blocks and are read into non-volatile ram when a continuous media file is opened. During a record session a new extent is allocated when the current extent is full. The allocation is synchronous in that there is thread processing time overhead involved during the

allocation. This time is kept to a minimum through the use of non-volatile ram for critical file system data structures. Incoming streams are packed into the pre-allocated extent blocks and placed on a disk write queue. Each network packet is copied into a disk buffer to ensure efficient use of disk space. The network packet sizes are generally small so that the time to copy the data is small per packet. The only requirement is that the copy overhead be less than the packet arrival period on the average.

Similarly, during a play session, extents are asynchronously read into a file transmit queue before the extent currently being transmitted is exhausted. Unlike the record operation where allocation takes place at the end of an extent, the next extent of the file is placed on the disk read queue when the current extent has only been partially transmitted.

5.3.4 Read Ahead

Figure 5.3 illustrates a single file read ahead mechanism.

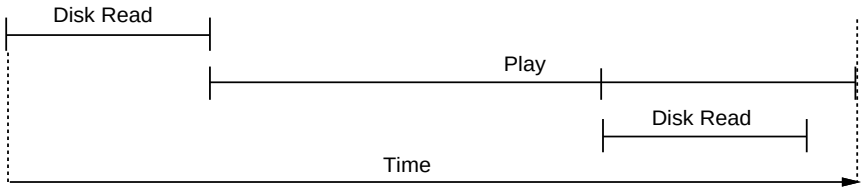


Figure 5.3: Single File Read Ahead in the CMFS

Essentially, a *water mark* is used to decide when to request the next data extent. Its value is chosen to preserve stream continuity. The use of higher fractions will reduce the overall amount of buffer space needed per stream, but increases the risk of stream starvation during concurrent access to other files. The water mark may be dynamically assigned depending on the file server utilisation. This may be achieved by looking at the size of the disk queues during playback and deciding how far in advance to make the request. The decision takes into account the maximum seek times and access delays during multiple concurrent access to the disk. The implementation uses a fixed water mark of 2/3 of the extent size.

Figure 5.4 shows multiple clients accessing disk extents on a single disk. Each client reads the first extent into memory and begins transmission of the data. Each disk transfer completes before the next request may be processed.

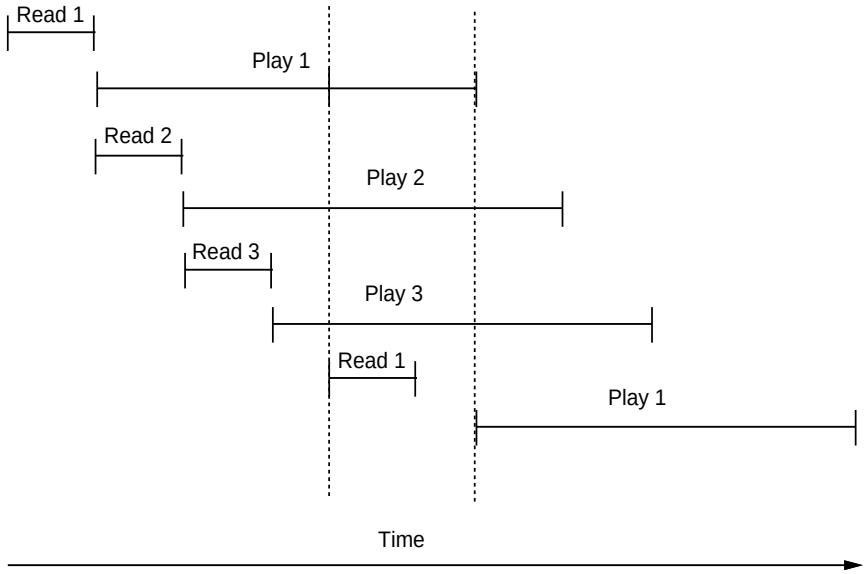


Figure 5.4: Concurrent File Read Ahead in the CMFS

5.3.5 Disk Arrays

A disk array is used in the server and disk striping is implemented to reduce the aggregate load on each disk. The striping is asynchronous and disk allocation on a single disk is independent from allocation on other disks in the system. A round robin scheme is used to choose a current disk for each stream. Synchronous striping would have required specialised hardware. In the actual implementation, each disk shares the same data channel. However, there is no architectural limitation on the use of multiple channels in the system. The main advantage of using separate channels is the increase in throughput by reducing channel contention. Though, contention for the bus from the parallel DMA access requests will increase.

In this case, striping is not implemented for parallel access to files, but is used as a disk load balancing scheme and to facilitate files which span multiple devices. Parallel access with an asynchronous system may require synchronisation of the data transfer after both transfers have completed. For example, suppose there are two disks in a system and a thread requests data from disk 0 and disk 1 simultaneously. Suppose, the transfer from disk 1 completes before the transfer of disk 0. The thread must wait for the disk 0 transfer to complete and send the data in the proper order. Parallel access to data for a single file is not necessary when considering the data rates used. For example, reading 4 seconds of video from disks takes an average of 462 ms [2]. An initial delay for reading the first extent is expected. Then, subsequent extent read requests are made far enough in advance as described above so that a thread is never starved of data. In this example, the request should be made at least

[2]This value is measured using an extent size of 90 blocks on a Micropolis 1578 SCSI disk drive.

462 ms before the playback of the current extent ends. The decision to implement parallel access should be reconsidered when the stream data rate increases such that the time to read data from disk approaches the duration of its playback.

Minimising Transfer Time

Many block devices offer performance optimisations within the device controller. These include DMA, multiple command channels and scatter/gather operation. DMA reduces the load on the CPU, since the CPU is not responsible for performing device to memory operations. Bus contention is a result of DMA capability when many devices desire access to the memory. Multiple command channels or command chaining reduces the latency for issuing commands to the controller. This is useful since the device can process multiple commands without necessarily releasing the data bus and the device driver does not have to wait for command completion before issuing the next command. Scatter/gather capability allows much greater throughput to the device when blocks are not contiguous in memory. Non-contiguous blocks of memory are transferred onto contiguous sections of the disk. The alternative is to request each block transfer individually, greatly increasing the total transfer time.

Transfer Time (ms)	Number of Blocks (4096Bytes)					
	5	25	50	75	100	125
Individual	84.8	435.5	847.8	1477.7	1798.9	2129.1
Scatter/Gather	33.3	113.7	209.7	313.1	408.1	507.4
% decrease	60.7	73.9	75.3	78.8	77.3	76.2

Table 5.1: Micropolis 1578 Disk Transfer Times

Table 5.1 illustrates the difference in these transfer times for a Micropolis 1578 asynchronous SCSI disk drive. The results are the average of 50 read and write operations from different sections of the disk. The final row represents the percentage decrease in transfer time accomplished by using scatter gather. The CMFS uses scatter/gather to achieve maximum disk throughput.

Disk Scheduling

Chapter 4 presented the considerations for disk scheduling. A CSCAN algorithm was implemented for its superior performance characteristics under heavy load [**Teorey72**]. The disk request rate would be quite high if disk requests were made for each block. This is undesirable since performing many disk access will demand much from the CPU and introduce delay to other threads due to the context switch times and

thread processing times. It is advantageous to reduce the number of disk requests, thus reducing the amount of overhead imposed by the controlling operating system.

In the CMFS, the disk request rate is reduced by transferring whole extents instead of individual blocks. Hence, the effectiveness of disk scheduling will also be reduced. The results of this design decision will be reviewed in Chapter 6.

5.3.6 CPU Scheduling

Wanda provides a simple priority based scheduler. Threads of the same priority run to completion unless preempted by higher priority threads or preempted by blocked threads at the same priority woken by an external event. Five main priorities exist within the CMFS and are described in decreasing order below:

- *Control Priority*: This highest priority is reserved for threads handling control of the file server. i.e. RPC calls and connection setup requests;
- *Disk Read Priority*: read requests to the disk devices are made at a high priority. This facilitates file read ahead for file playback threads.
- *Audio Priority*: Audio processing threads for both play and record run at a higher priority than video threads. Audio packets are smaller and arrive at higher frequency. It is therefore advantageous for audio threads to preempt video threads to minimise audio jitter;
- *Video Priority*: Video processing threads.
- *Disk Write Priority*: write behind is accomplished by making write requests to disk at the lowest priority.

Figure 5.5 illustrates the main threads of control for the CMFS. The highest priority is assigned to control requests for two reasons. Firstly, under normal operation, control requests are relatively infrequent when compared to the period of stream processing threads and do not generally disturb the operation of these threads. Secondly, it was desired that the server be as responsive as possible during control operations. Disk queue management is performed by two independent threads per disk; one thread for each of the read and write queues. A central point of control facilitates the easy implementation of queue management schemes. The network device is multiplexed by independent threads processing packets into the different disk queues..

Higher priority for disk read threads ensure that read requests to the disks are serviced above other devices. Thus, read ahead of files into a file transmission queue

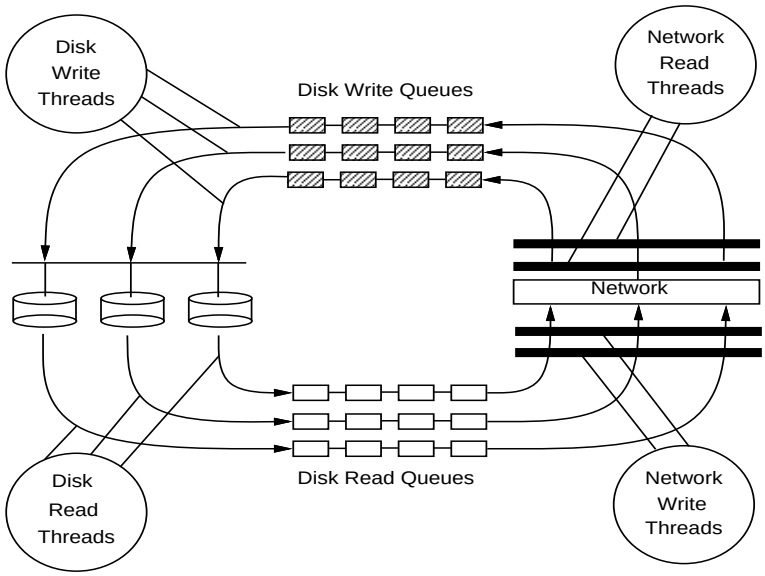


Figure 5.5: Thread control in the CMFS

can be accomplished before the transmission queue is depleted. Disk write threads execute at a low priority so that audio and video threads reading the network device are not preempted. There exists a danger that the available buffer space will be exhausted before the disk write threads have a chance to execute. Generally, there is enough time to service write requests because threads are periodic and block, but if the read request rate is high, write queue lengths may increase beyond a desirable threshold. One solution is to implement a control priority thread which monitors the total system buffer space and controls the disk write queue lengths by increasing the priority of disk writes.

Priority inversion of threads is possible in WANDA if the hardware interrupt priorities are adjusted incorrectly. For example, suppose the network device operates at a higher priority than the disk device and consider two independent audio threads; one reading audio samples from disk to the network and one writing audio samples from the network to disk. Threads controlling each device issue requests to their respective devices and wait for interrupt acknowledgement before continuing. It is possible that the audio read thread does not get executed on time because network interrupts are serviced in preference to the disk. Essentially, raising the hardware priority of the network may starve the otherwise higher priority disk read thread of CPU when the network in heavily loaded. For this reason, the hardware priorities for the disk and network are the same.

There is no explicit real-time CPU scheduling algorithm employed inside the kernel. However, periodic threads in the system calculate their own deadlines and request that the kernel schedule the threads according to these deadlines. The result is essentially a rate monotonic earliest deadline first CPU scheduling algorithm. This

approach differs from conventional real-time operating systems because no explicit interface is used to specify the period of a thread. Also, the kernel is not responsible for maintaining scheduling information for each thread. This mechanism will be discussed in more detail with respect to stream synchronisation in section 5.8.

5.4 File Representation

Files are represented as a stream of bytes. No assumptions are made about the structure of a continuous media stream. This allows for great flexibility when designing higher level services. This section discusses the representation of a file on disk and in memory while a file is open.

As previously discussed, file allocation granularity is the extent. Extent sizes are chosen carefully such that the average size of a file should fit into a small number of extents; and a single extent should provide enough buffering to prevent starvation at the servers maximum utilisation, i.e. when the maximum number of continuous media streams supported by the hardware are active.

An open call for access to a file creates a volatile data structure with the information shown in Figure 5.6. The file inode is read from either the disk or non-volatile RAM.

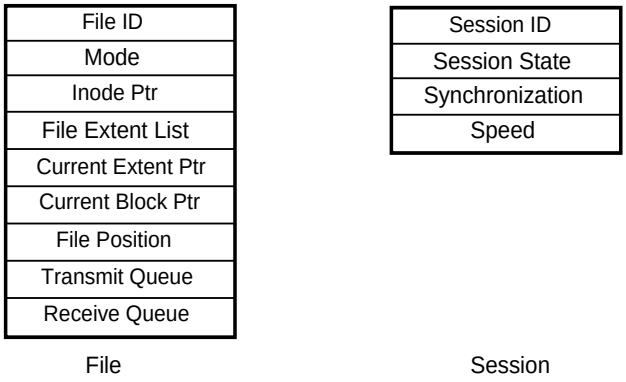


Figure 5.6: An Open File Representation

File receive and transmit queues are blocks which are ready for writing to disk or network transmission, respectively. Current byte offsets into the file are kept and clients are required to maintain the temporal relationship between byte indices.

Figure 5.6 also shows the representation for a session. The state information shows the current behavior of the stream. The options are playing, recording, idle, or paused. The synchronisation information consists of a timed semaphore and will be described in section 5.8. The speed parameter indicates the files current playback speed relative to the timestamps provided in the streams' data packets.

Once created, a file is represented by its capability to clients. The capability structure implemented includes a server address, an owner, and the file inode number in the server. This provides a simple mechanism for file location and is not intended as a general purpose capability architecture. A secure capability system is beyond the scope of this dissertation, but nothing in the CMFS design prohibits a secure system.

5.5 Caching

From the discussion of the characteristics of continuous media in Chapter 1, file data caching is not performed on the server. A data cache is inappropriate for high frequency and sequential access to files. Caching is a benefit when the file access characteristics exhibit temporal locality, that is, the behavior that recently accessed data will be accessed again in the near future. This is still true for continuous media file meta data, but not necessarily for the file contents. In this case, it is advantageous to remove recently accessed data from buffer space quickly to allow new data to enter the system. It was decided only to cache critical *meta* data, such as file allocation blocks (i-node information) and disk allocation bitmaps. 2 MBytes of battery backed (non-volatile) ram is used for the data cache and is periodically written to disk.

The use of non-volatile ram serves two purposes. The first is to ensure greater file system integrity in case of failure by providing data replication and fast access to critical file system data structures. The second is to reduce disk head movement when updating file system data. For example, during a video recording, any disk allocation will require the update of the disk free block bitmap and inode table entries. These structures are stored at a fixed location on the disk [3] and updates involve at least one seek and one write. A read may also be required if the desired data is not yet in the cache. The use of non-volatile ram eliminates unwanted disk head movement, reducing disk scheduling jitter introduced by scheduling and running threads that perform the extra disk accesses.

A disadvantage with the use of non-volatile ram is its susceptibility to corruption in the event of a system failure. A discussion of non-volatile ram integrity is given in [**Wilson92**]. It is argued that ram reliability may be improved through the use of an access protocol, reducing the risk of accidental memory corruption during a system crash. Several such protocols are suggested, but the experimental results are inconclusive.

[3] This data may also be replicated on the disk for greater integrity.

5.6 Concurrency Control

The need and use for concurrency control was discussed in section 4.3. Locking in the CMFS occurs at several different levels and exists for file system data structures such as disk allocation bitmaps, the open file table, and disk extent structures. Two independent threads may access the same file, but are not allowed disk block level write sharing. That is, only one thread may hold a lock on any given disk block belonging to the file at any one time and may access any or all bytes within that block.

This is because of a lack of any other well defined boundary for concurrent write sharing. For example, a system dependent on knowledge of the specific encoding used for a video data stream might impose the unit of access as a single frame. The choice of locking at the disk block level is not a problem for two reasons. Firstly, a thread writing small samples into the disk block will only hold the lock for the length of time needed to copy the sample into the block buffer. Secondly, the data rate for streams is high enough that any one block is not locked for a long period of time. The problem of concurrency control becomes more complicated when client caching of file data is considered. However, no client caching is implemented in the CMFS.

5.7 Flow Control

A server based flow control mechanism is used in the CMFS. This was primarily motivated by the design of the Pandora Box [**Hopper90**]. The Pandora model dictates that continuous stream delivery begins directly after connection setup. Flow control for video in the Pandora Box is rate based and consists of modifying the frame rate. No flow control mechanism for audio is provided in the Pandora Box. This has proved inconvenient for synchronisation of continuous media streams because it requires a client to discard data on individual connections until all streams are started.

The CMFS interface also supports a client based flow control mechanism. Read and write operations use the same underlying server file access routines and a client can make these calls as frequently as desired. The maximum allowable frequency will be determined by the performance of the underlying RPC mechanism, assuming reads are performed by a single thread.

5.8 Synchronisation

As stated in Chapter 3, inter-stream synchronisation is viewed as another service provided in the distributed system. However, some support for synchronisation to minimise stream skew is provided within the operating system. The important considerations contributing to the success of this approach are:

- An accurate clock with resolution on the order of the highest frequency system thread.
- A deterministic priority scheduling mechanism
- Lightweight processes or *Threads*

Independent threads are used to control streams and synchronisation is achieved through the use of a single clock [4]. Threads are synchronised to an absolute start time and delay between packet transmissions is calculated relative to the current value of the clock and the absolute start time. Adopting this scheme eliminates jitter in the per packet thread processing time as illustrated in Figure 5.7.

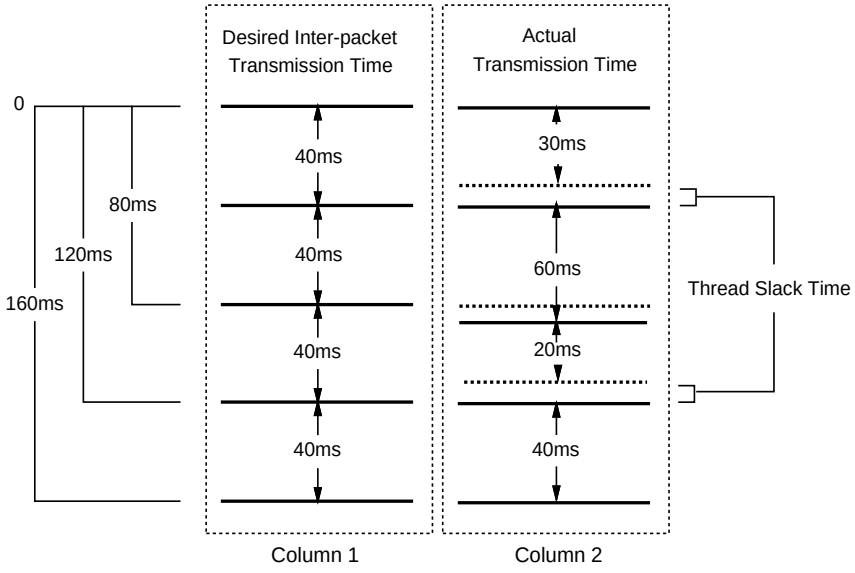


Figure 5.7: Thread Synchronisation with Real-Time

Each dark line represents the transmission of a video packet. A 40 ms interpacket transmission time is assumed for illustration purposes and column 1 shows the time progression desired for stream transmission.

[4]The system clock provided by the MVME147 68030 CPU board is operated at a resolution of 4 ms.

Column 2 shows the actual per thread processing time during execution. Jitter is introduced by interrupt service routines and other schedulable threads in the system. Time 0 is associated with the time of the first packet transmission. The thread processing the stream uses 30 ms to prepare for transmission, so it may sleep for 10 ms to transmit at the required time. The thread then sees 60 ms pass during the processing of the third packet and does not sleep because it has missed its previous transmission deadline. Instead, the thread transmits immediately and without relinquishing the CPU, begins processing the fourth packet which takes only 20 ms.

At this point, the amount of time that the thread sleeps must be calculated accurately to maintain real-time synchronisation. The actual time passed is 110 ms. Hence, the thread must sleep for 10 ms to make the desired transmission time of 120 ms from time 0. If the thread were to sleep for 20 ms, the interpacket transmission interval minus the last packet processing time, actual transmission will occur at time 0 plus 130 ms. This would introduce skew in the stream with respect to other threads synchronised to the same clock. All threads synchronised to a common start time remain synchronised with respect to the clock and with each other, if the operating system can maintain the scheduling requirements. A single timer queue is provided in the kernel in increasing time order with respect to the system clock and the scheduler executes the earliest deadline thread.

As previously mentioned, a timed semaphore is used to accomplish stream synchronisation. However, the timing mechanism is not standard. Normally, a timeout is implemented by a thread blocking for a specific length of time. This is undesirable because synchronisation in real-time depends very much on when the request to block is made. Although the length of time blocked is correct, there may be skew in the timing of the request in each thread because of the differences in processing time. Instead, each thread is responsible for placing itself on the kernel timer queue precisely when it wants to be scheduled. That is, threads calculate an absolute time to be scheduled in the future. The semantics are block until *this time* rather than block for this amount of time. This approach is no more costly since the calculation for an absolute time must be performed anyway. The difference is that the user thead makes the calculation rather than the kernel. In the example illustrated in Figure 5.7, each thread monitors its actual processing time and calculates the amount of time to block. A major disadvantage of this scheme is that no explicit mechanism for fair scheduling is provided and resource sharing relies on the friendly coexistence of threads. That is, all threads are expected to release the CPU at some point in time. However, this assumption is not unreasonable in the context of a dedicated specialised service.

5.9 Summary

A detailed implementation of a Continuous Media File Server was presented. A general distributed computing architecture to which this file service may belong was described. A multi-level service hierarchy facilitates the integration of different low level services without imposing performance overhead.

A distributed interface takes into account the differing access characteristics inherent with continuous media. The issues in disk management including file system layout, file representation, free space management, allocation strategy and disk arrays were described.

Real-time characteristics of the server include a C-SCAN disk scheduling algorithm and a rate monotonic deadline based CPU scheduling policy. Stream synchronisation and flow control mechanisms were presented. Each stream is synchronised relative to the system clock. A server based flow control mechanism is used to control the transmission rate of continuous media streams.

Chapter 6

Experimentation

This chapter presents measurements of the CMFS described in Chapter 5. The experimental apparatus is illustrated in Figure 6.1 and consists of an MVME147 with a Motorola 68030 CPU connected to both the laboratory 10 Mbit/s Ethernet and a 50 Mbit/s CFR. Two 380 MByte Micropolis SCSI disks are connected via a single SCSI channel. The Ethernet interface is used for control of the file server while audio and video data are transmitted over the CFR. There is no architectural constraint on the use of two network interfaces and client/server communication may well have been achieved through only the CFR interface using an Ethernet to CFR gateway. However, the use of an Ethernet increased communication reliability in the experimental apparatus due to the constantly changing CFR environment. The host operating system for the CMFS is WANDA compiled using the Gnu C compiler (GCC) with optimisation turned on. The Pandora Box connected to the CFR is controlled via a transputer link from an Archimedes running Unix.

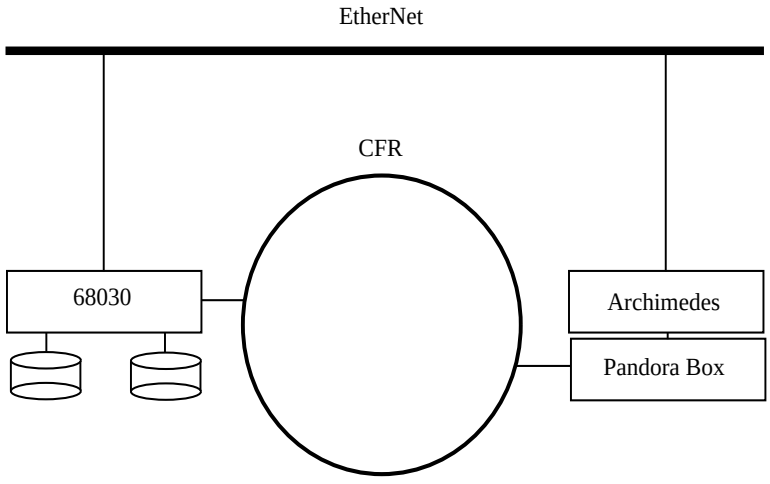


Figure 6.1: CMFS test apparatus

A number of experiments were carried out with multiple Pandora Boxes connected to the CFR. All timings are performed using a separate memory mapped 5 MHz global clock board fitted to the CMFS hardware. The experiments performed are designed to illustrate the effects of concurrent access to files on the CMFS. Two sources of stream jitter identified in [**Dixon92**] are:

- Thread Contention: Multiple threads requesting access to common resources;
- Interrupt Service Routine Contention: In particular, clock interrupts and CFR interrupts were identified as a source of stream jitter.

The reduction of stream jitter is important because maximum jitter values determine the amount of buffer space required per continuous media stream. To establish the effects of multiple streams on the CMFS, multiple stream processing will be compared with base cases measured for single audio and video streams.

Recording vs. Playback

This chapter explores both the recording and playback of audio/video streams in the CMFS. One might assume that recording and playback operations are symmetric. However, there are several differences between these cases which are worth considering. Certainly, the data rates may be different for each case. That is, the rate of recording may be different from the rate of playback. For a recording session it is advantageous to keep packet loss to a minimum, but we may tolerate more jitter in the stream since there is no presentation of the data to an application. This contrasts with the playback case where introducing jitter may affect the quality of service to the end viewer or listener. It also suggests that there must be more buffering in the server to absorb the greater variation in thread scheduling. Otherwise, packets arriving at the network interface may overflow the available buffer space when the processing thread has been preempted for long periods of time. The primary difference, at least for this implementation, is the need to maintain inter-stream synchronisation during playback.

Another difference is that during playback greater flexibility exists in the control of the stream at the server. For example, during a recording session the server is required to accept the incoming data as it arrives at the network interface. If audio packets arrive one every 4 ms, at least one context switch is required to service a packet every 4 ms. During a playback session, the server may transmit ten audio segments every 40 ms, reducing the required number of context switches by a factor of ten. The consequence of this is the need for a larger buffer in the client to manage the extra data sent. This is an advantage because it reduces the overhead imposed on the server and distributes more work to the clients.

This discussion suggests some asymmetry between recording and playback of streams and that the consequence would be the use of larger buffers in the server for recording and larger buffers in the client for playback. This is not necessarily a valid conclusion because it depends largely on the construction of multi-media devices. For example, an increase in buffer size at a source device means that more data can be transmitted less frequently during a recording session. Although this will increase the overall delay experienced between the source device and the storage service, this is less important for data storage applications.

In the following sections, measurements were taken for both the packet inter-arrival times (recording) and inter-packet read times (playback). These timings include all processing performed by a user space thread to service each packet from or to the network, respectively. Although jitter may be considered less important during a recording session, if there are significant variations in delay while processing incoming packets, packet loss may result because of the system inability to expand the available buffer space.

The particular experimental network interface used allows us to consider symmetry between recording and playback for two reasons. Firstly, packet sizes and thread periods are the same during recording and playback unless otherwise indicated. Thus, transmission and reception characteristics over the CFR will be similar. Secondly, even if the playback thread frequency is reduced by sending more data, context switching between user space and the kernel will not decrease because an interrupt is generated on every cell transmission in the CFR. For these reasons, the analysis of inter-packet arrival may be comparable to the analysis of transmission. Results for both cases are presented since there a differences in server operation between the two cases. For example, a video playback thread does not require time to perform a disk allocation.

6.1 Pandora segments

Pandora audio and video are packetised into segments and transmitted using the MSNL lightweight virtual circuit protocol [**McAuley90**]. Figure 6.2 illustrates the structure of a general pandora segment, a video segment and an audio segment.

Each entry in the segment header is a 32 bit quantity. Sequence numbers and timestamps are included in the general header for timing and error detection. Audio segments are 48 bytes long and include a segment header overhead of 20 bytes. An extra 28 bytes is imposed by protocol overhead to round the 68 byte packet length to the nearest multiple of 32 bytes needed for CFR transmission. Audio segments are generated every 4ms.

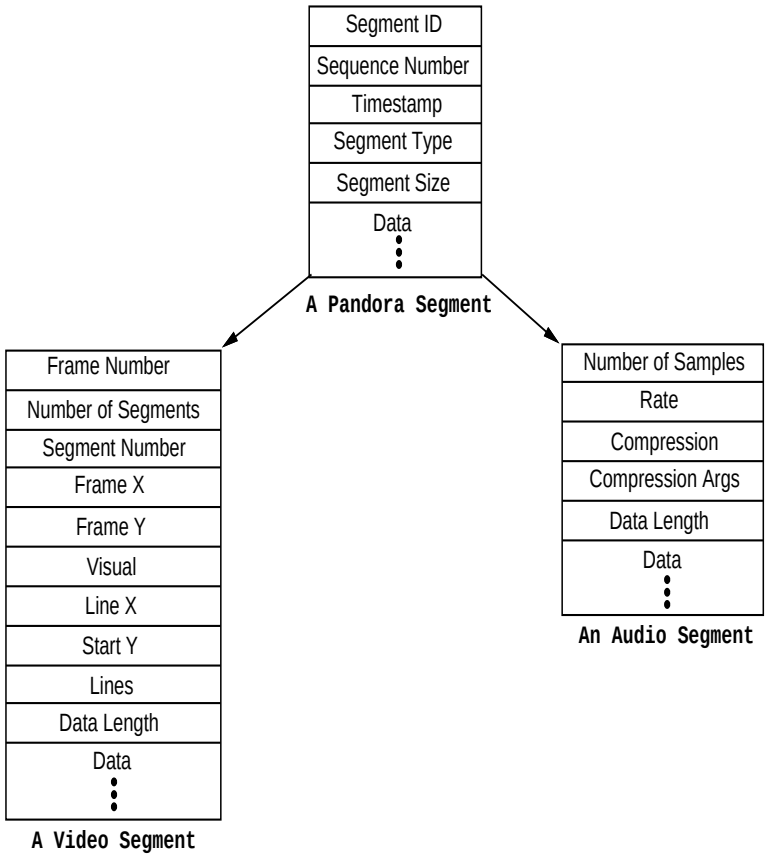


Figure 6.2: Pandora Audio and Video Segments

Normal video segments are 4144 bytes in length and two video segments comprise a video frame. Compressed video generates a segment size of 2144 bytes. Segment header and protocol overhead amounts to 48 bytes in both cases. At full frame rate, two video segments are transmitted every 40 ms. Experimentation is carried out using half rate normal video. Thus, two video segments are transmitted every 80 ms. Furthermore, the segment sizes are reduced slightly to accommodate the constraints imposed by the video display extensions within the X11 Window System. Thus, normal video and compressed video segment sizes are 3632 bytes and 1840 bytes, respectively.

6.2 CFR Throughput

Initial experiments were carried out to find the maximum throughput of a single station on the CFR using just the optimised WANDA kernel with no other concurrent activity. Two Motorola 68030 processors were placed on an otherwise unloaded single slot CFR; one sending 3680 byte packets at approximately 9 ms intervals and

one receiving packets from the network. The maximum throughput without packet loss was measured at 3.2 Mbits/s. This might be improved by coding critical sections of the CFR driver in assembly language, but is important only as a reference point for the following discussion.

The disk devices will contend with the CPU for access to memory during concurrent disk and network activity. The goal of the next experiment is to determine the effects of different size disk accesses on the network throughput. In the experiment, the data rate of the disk and network are fixed at 3.2 Mbits/s and the disk read request rate and request size are varied. Then, the throughput of the CFR is measured. The disk extent size is varied from one 4096 KByte block to 60 such blocks in one block increments. The request rate is adjusted accordingly to maintain a data rate of approximately 3.2 Mbits/s. For example the request frequency is one every 9 ms for the one block case and one every 0.61 seconds for 60 blocks.

CFR Throughput	Number of Blocks (4096Bytes)						
	1	10	20	30	40	50	60
(Mbits/s)	3.14	3.14	3.11	3.10	3.12	3.16	3.15

Table 6.1: CFR throughput During Concurrent Disk Activity

The results in Table 6.1 indicate that the throughput is not adversely affected by concurrent disk access. The throughput in each case was at least 3.1 Mbits/s which is 98% of the maximum throughput without disk contention. This is because the disk controller is asynchronous and must release control of the system bus to the CPU every 50 μsecs. The disk must renegotiate access during large transfers. An important thing to note about Table 6.1 is that the throughput does not exhibit an obvious relationship to the number of blocks. Intermediate values, not shown in the table, show a similar variance and are due to the characteristics of the disk. For example, crossing a track boundary during a contiguous transfer will introduce some delay and affect the length of time the disk requires the data bus. However, none of the intermediate values differ significantly from the ones presented.

Scheduling Jitter (ms)	Number of Blocks (4096Bytes)						
	1	10	20	30	40	50	60
Stand. Dev. from 9ms	2.4	2.4	2.7	3.1	3.3	3.7	4.2
Stand. Dev. from 50ms	1.1	0.98	0.82	0.64	0.60	0.54	0.55

Table 6.2: Scheduling Jitter During Concurrent Disk Activity

What is very noticeable is the thread scheduling jitter experienced by the network sending thread as a result of memory contention. Table 6.2 shows the jitter in

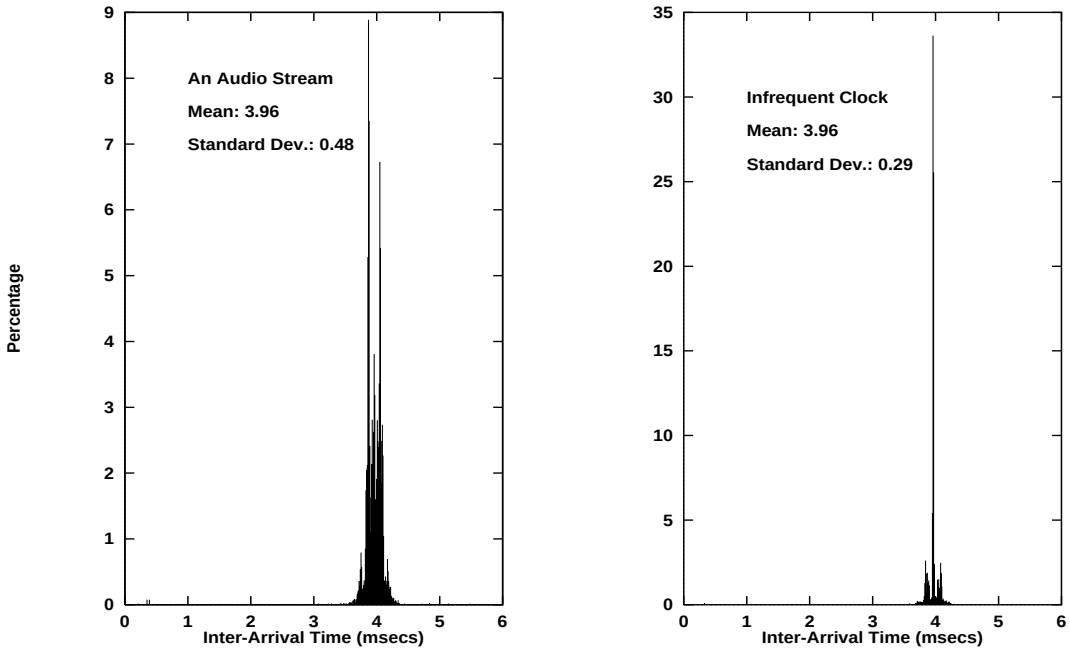


Figure 6.3: Clock Interrupt Effects on Packet Inter-arrival Time

scheduling in increments of 10 blocks. Again, these results do not imply a linear relationship between the number of blocks and the thread scheduling jitter, but are representative of the following trend. The jitter is represented by the standard deviation from the mean thread scheduling period. The thread sending a 3680 byte packet every 9 ms will contend with the disk for memory access more often as the size of the data transfer increases. This is because the data transfer takes a longer time to complete and a high frequency thread requires access to memory more often. The effect is less noticeable in lower frequency send threads since there is less overlap in the amount of time that both the disk and sending thread need access to the bus and memory. In fact, the scheduling jitter decreases because the sending thread does not contend with the scheduling of the disk thread. For illustrative purposes only, a thread period of 50 ms was chosen, which corresponds to about one half the data rate of a single combined audio and video session.

6.3 Audio Streams

This section details the characteristics of audio only streams in the CMFS. Audio stream processing threads operate in user space at the priority described in Chapter 5.

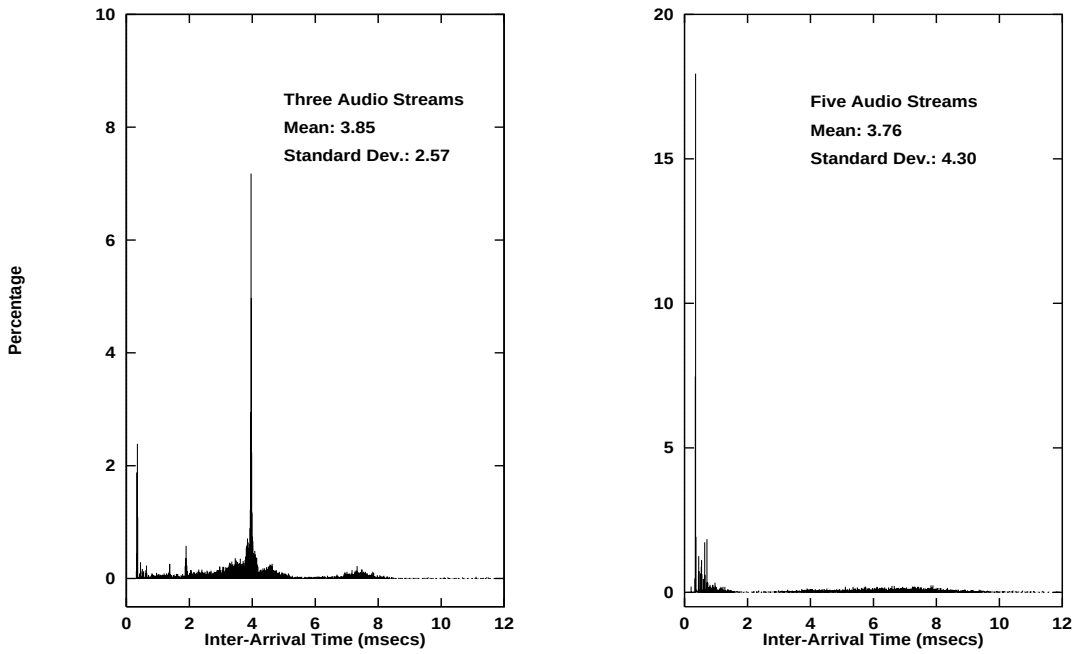


Figure 6.4: Audio Inter-arrival Times With 3 and 5 Streams Active

Figure 6.3 shows the packet inter-arrival times in user space during a recording session including the packet processing time. This represents the base case operation for audio streams. In the initial experiment, the clock interrupt frequency is set to one every 625 μsecs to contrast with the accompanying less frequent clock interrupt of one every 0.5 seconds. These values are only chosen to highlight the effects of clock interrupts on stream jitter and are not intended as values for the operation of the CMFS. In general, the clock interrupt frequency should be on the order of the period of the highest frequency thread to ensure accurate scheduling of these threads. The CMFS uses a 4 ms clock for inter-stream synchronisation during playback. The second experiment shows the result of decreasing the clock interrupt frequency to one every 0.5 seconds during a recording session. As expected, a higher percentage of packets arrive with less jitter. The experiment for playback of a single audio stream shows a mean transmission period of 3.91 ms and a standard deviation of 1.6 ms. The greater variance is due to the synchronous preparation of disk read requests.

An experiment with two audio streams shows only a slight increase in jitter experienced by one stream. The mean inter-arrival time remains 3.96 ms with a standard deviation of 0.73 ms. This is due to the low data rates associated with an audio stream. Three 32 byte CFR cells comprise one audio packet and one CFR interrupt is generated on every cell. Each stream has a data rate of 192 Kbits/s at the net-

work interface and may also be expressed by the arrival of 750 cells per second. This generates an interrupt once every 1.33 ms. It is possible to anticipate the arrival of another cell by dallying inside the interrupt routine and copying the next cell out of the FIFO without the overhead of generating a separate interrupt. This is called *optimistic reception*. However, this has the effect of introducing jitter in other streams when large packets are received, since more time is spent servicing incoming cells without returning from the interrupt service routine. For these experiments, the CFR interface is programmed without optimistic reception. Again, for the playback of two audio streams there is an increase in variance. The mean thread period is 3.81 ms with a standard deviation of 3.23 ms.

Figure 6.4 shows the inter-arrival times for three and five active streams. When three streams are active, thread contention causes network driver code to queue packets onto a protocol receive queue faster than they can be removed by each audio thread. In this case, a CFR interrupt occurs every 444 μsecs causing a context switch from a user space thread into the kernel CFR device driver. This effect is even more evident when five streams are concurrently running where the network interrupt frequency is one every 276 μsecs. Essentially, the average inter-packet arrival time decreases because packets have already been queued in the protocol code and reception of the next packet requires only a few hundred microseconds to remove from the head of the queue. The result illustrates the increase in buffer space needed when concurrent streams are active. The results for playback of three and five streams are (3.87 ms, S.D. 2.89 ms) and (3.79 ms and S.D. 9.17 ms). The thread contention with five streams introduces noticeable degradation in an audio stream during playback.

Packet loss is an inevitable result of increased load, due to the inability of the system to process incoming CFR cells. The network interface throws a packet on the ground (TOG), after a programmed number of hardware retries, if the cell currently being received is not removed from the single cell FIFO provided by the CFR interface before the next cell arrives. No packet loss was observed with a single audio stream and three contending audio streams. A value of 3-4% packet loss is observed when five streams are active. This increase in packet loss is due to contention for the CFR interface as described above. Although the bandwidth is well within the CFR measured limit of 3.2 Mbits/s, the interface has the limitation of not being able to receive the CFR cell immediately following one that has just been received. Thus, the reception of one cell from an audio stream might cause a following cell belonging to another stream to be dropped.

6.3.1 Audio Processing

The timings of the previous sections include all thread processing times of individual audio packets. The following discussion examines the time necessary to process each

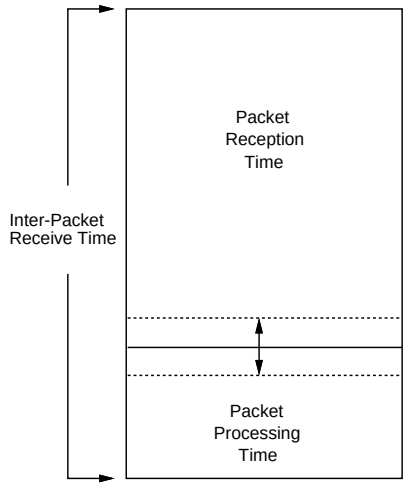


Figure 6.5: Total Thread Execution Time

packet from the network queue onto a disk queue. That is, the time after reception of one packet to the time just before the next packet is received from the kernel.

Figure 6.5 illustrates the total execution time of an audio processing thread. The packet receive time includes the amount of time a thread may block in the kernel awaiting a packet from the network interface. The processing time is only the amount of time to transfer the received packet to a disk buffer.

Figure 6.6 shows the processing times for each packet received from the network. The scale of this graph is adjusted for comparison with later graphs and does not show the few points where disk extent allocations The jitter shown near 450 μsecs is due to the varying amount of time needed to copy the data packet into a disk block buffer. For example, an audio packet might fit completely into the current disk block or the current disk block must be added to the disk extent queue for writing to disk and a new disk block must be removed from the file receive queue. have taken place. During this experiment, 9 disk allocations were required raising the average total processing time to 8.2 ms and accounts for the large standard deviation. The allocations are infrequent and the processing times relatively low so the audio processing thread can catch up after a single allocation. The playback processing time is 287.2 μsecs (standard deviation 35.07 μsecs), which is remarkably similar to the recording time. The lower standard deviation differs from the recording case because processing of an audio packet only includes the transmission time over the CFR and not any file processing.

Figure 6.7 shows the effects of multiple streams on the packet processing times of one of the streams. The processing time scale in each graph has been adjusted to compare the interesting regions. Individual packets in each case may experience up to 9 ms delay which accounts for the large standard deviation from the mean.

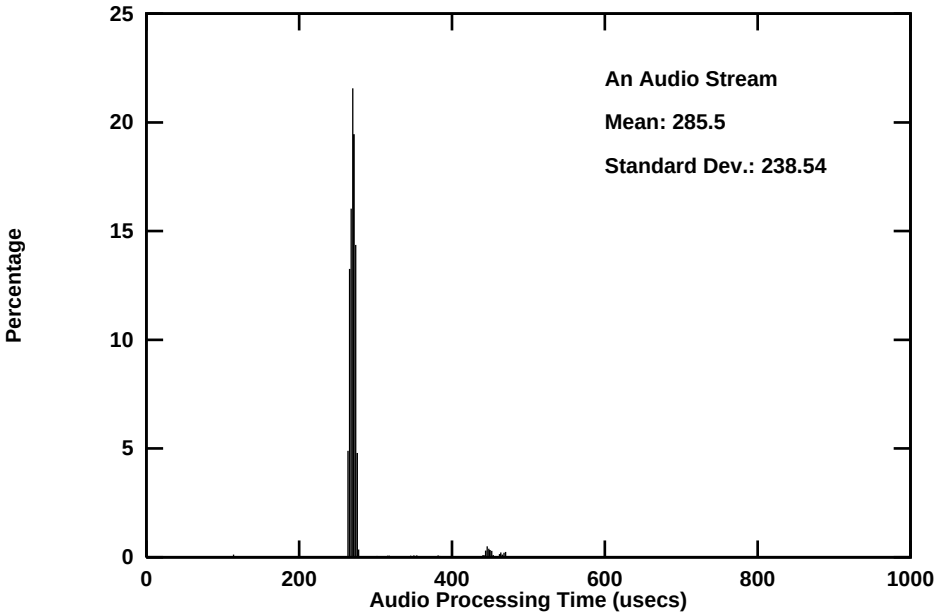


Figure 6.6: Audio Packet Processing Times

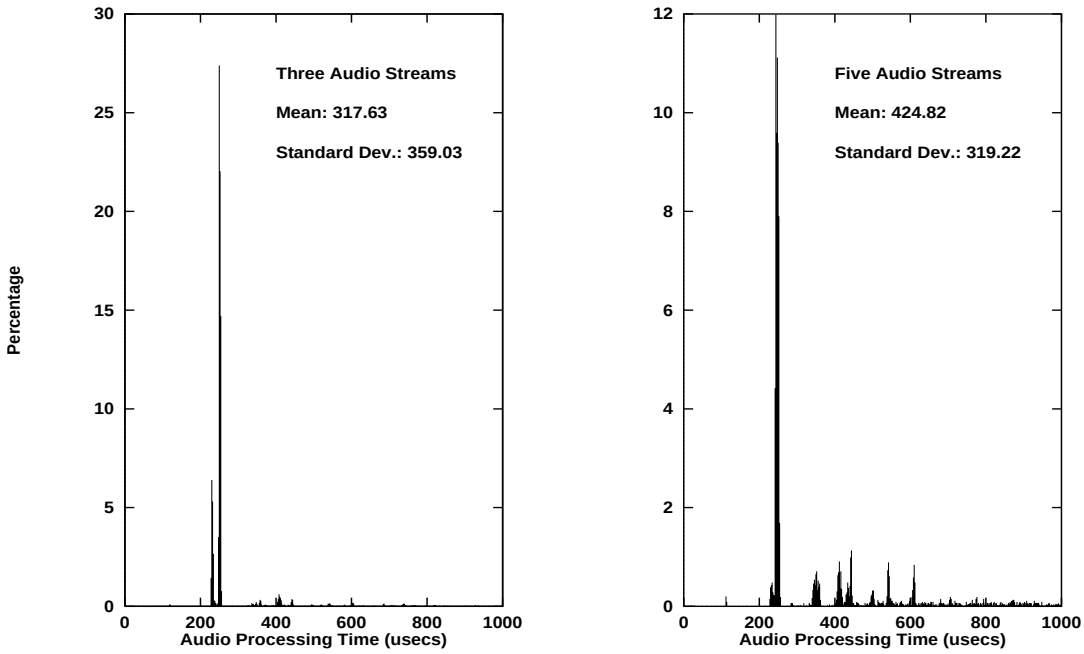


Figure 6.7: Effects of Stream Contention on Audio Processing Times

Audio threads run at the same priority and, under WANDA, run to completion unless preempted by a higher priority thread or a thread at the same priority which is woken by an external event. Only one higher priority thread, which accepts RPC control signals for the server, is active during the experiment. Thus, very little jitter is introduced when servicing one two or three streams because the average processing time per packet is much less than the interval between CFR interrupts.

When processing five streams, processing time jitter is more apparent due to the increase in contention with the CFR interrupt service routine. Each spike in Figure 6.7 roughly corresponds to the context switch latency introduced by WANDA switching among the five audio threads. The processing timings for the playback of three and five streams are (mean 299 μsecs, S.D. 1200 μsecs) and (mean 299 μsecs, S.D. 1700 μsecs), respectively.

6.4 Video Streams

Each video packet is 115 cells in length and two such packets are transmitted every 80 ms. With a single video stream active, 230 CFR cells arrive per frame. Without optimistic reception this would yield and interrupt frequency of one cell every 348 μsecs. However, for the following video experiments, the CFR interface was programmed with optimistic reception. That is, an interrupt is generated for the first cell only and the network driver dallys for a small amount of time awaiting the arrival of the next cell. This was done to minimize the contention effects from requiring a CFR interrupt for each cell during a large bursty transfer.

Figure 6.8 shows the inter-arrival times of video packets on a single stream without contention from other streams. The result is different from the audio case in that more packets have been queued in the network code before the user space thread has the chance to process the next packet. This accounts for the high percentage of packets which have small inter-arrival values. This is due to (1) the longer time necessary to assemble a complete video packet from cells in the network protocol code and (2) the longer amount of time to process a video packet in user space and (3) that a burst of two 3680 byte packets are sent every 80 ms. Each video packet requires an average of 18.3 ms before it is assembled in the network code. This contrasts with an audio packet length of 3 cells which takes an average of 330 μsecs to assemble. The relatively long assembly time for a video packet is not only a function of the WANDA networking code, but is due to the inter-cell transmission time at the source Pandora Box. That is, some delay exists between cell transmission at the source, which accounts for the overall delay in complete packet reception.

Because a burst of two segments is transmitted every 80 ms, the second segment in

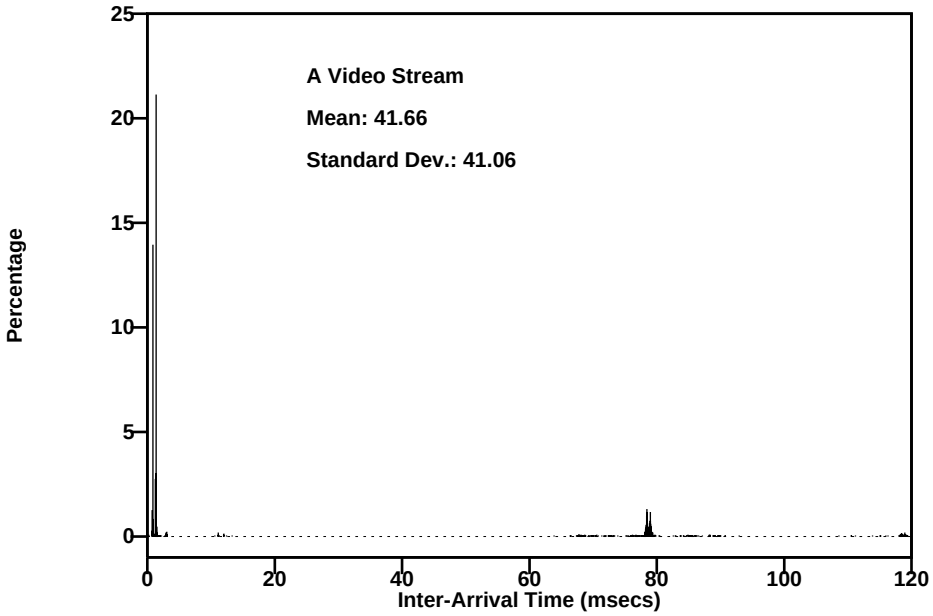


Figure 6.8: Video Inter-arrival Times With A Single Stream

the burst will be assembled and queued in the networking code, before the first video segment can be processed. That is, the second network interrupt transfers control into the kernel protocol processing code, preempting the user space processing of the first segment. The expected result is that 50% of the packets arrive near 80 ms and the other 50% near 0 ms. The actual number of packets which arrive near 0 is increased for two reasons. The first reason is that disk allocations performed during the record session are performed synchronously. This increases the time to process a packet by as much as 9 ms. The second and predominant factor is that the CFR driver, in this case, completely builds a video packet without returning from the interrupt service routine. This reduces the amount of time the video thread is scheduled while there is interrupt processing. The playback of one video stream is similar to the recording case. The mean transmission period is 41.50 ms with a standard deviation of 33.44 ms.

As expected, Figure 6.9 illustrates that thread contention and CFR interrupt service contention increase with two and three video streams active. With two video streams active, 5,750 cells arrive at the interface every second and with three streams active the interface must process 8,625 cells/s. This corresponds to an interrupt frequency of one every 174 μsecs and 116 μsecs, respectively. Notice with three streams active that almost all packets are queued in the network code before reception occurs in user space. This is because 40% of the CPU is used processing CFR cells. This calculation is based on the observed average processing time per cell of 46 μsecs.

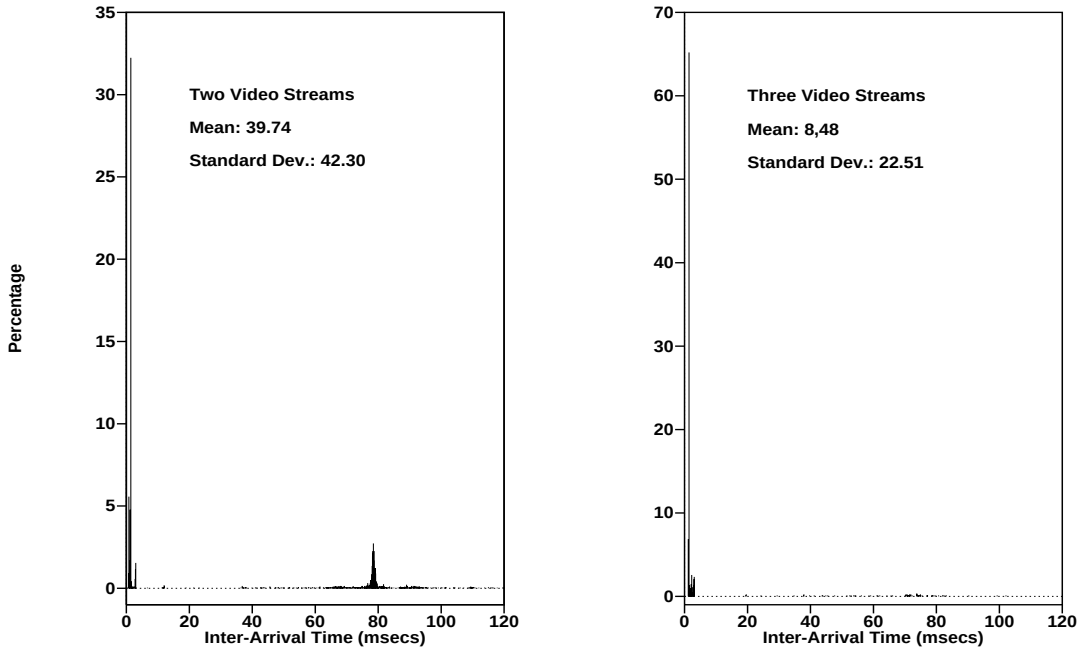


Figure 6.9: Video Inter-arrival Times With 2 and 3 Streams Active

During the playback of two streams the average thread period is 41.29 ms with a standard deviation of 32.53 which is a similar result to the recording case.

Less than 1% packet loss was observed in a single video stream. Packet loss increases to 3% for two streams because of contention for the CFR as described in section 6.3. However, a more significant increase in packet loss is observed in handling three streams. 30% of the packets are dropped because the CPU does not have time to process each video packet in user space before another CFR interrupt switches control back to the kernel. The result is that incoming video overflows the available buffer space and packets are dropped in the network protocol code rather than the CFR interface. The large difference between the two and three stream examples is due to the substantial increase in network interrupts which decreases the system capacity to schedule the user recording threads.

6.4.1 Video Processing

Figure 6.10 shows the processing time for one and two video streams. Disk allocations are performed synchronously and are visible near 11 and 12 ms. Two distinct spikes are visible; one just under 1 ms and one just over 1 ms. The result is similar to the audio case in that these represent the difference in time caused by the thread

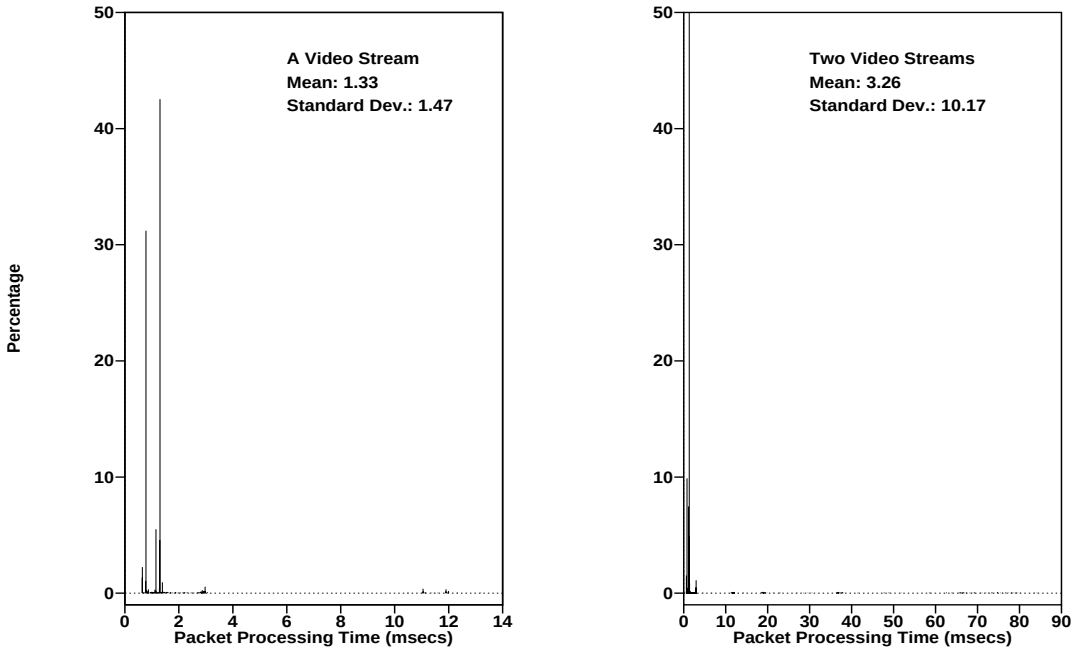


Figure 6.10: Video Processing Times

removing a new disk block buffer from the file receive queue. Either the 3680 byte video packet will fit entirely into the current 4096 byte block, or the video packet must be split into two blocks, requiring the current block to be queued in the extent another block to be fetched from the queue. Because of the comparatively large packet size to audio, a much higher percentage of video packets involve this operation. The remaining variance around the mean is due to contention with the CFR interrupt service routine as described in the previous subsection. Video processing on playback is shows a mean time of 5.31 ms with a standard deviation of 0.80 ms for a single stream. This represents the transmission time of a packet over the CFR.

The processing time increases in both mean and variance when three streams are active (mean 7.3 ms, S.D. 20.75 ms) since more contention exists between thread scheduling and the CFR interrupt service routine. Video playback processing for two streams show and increase in the mean to 5.77 ms from the single case and the standard deviation increases to 2.34 ms due to contention for the CFR.

6.5 Combined Audio and Video

Figure 6.11 shows the effects of audio and video streams together during a recording session. The two streams are initially synchronised to start at an absolute time

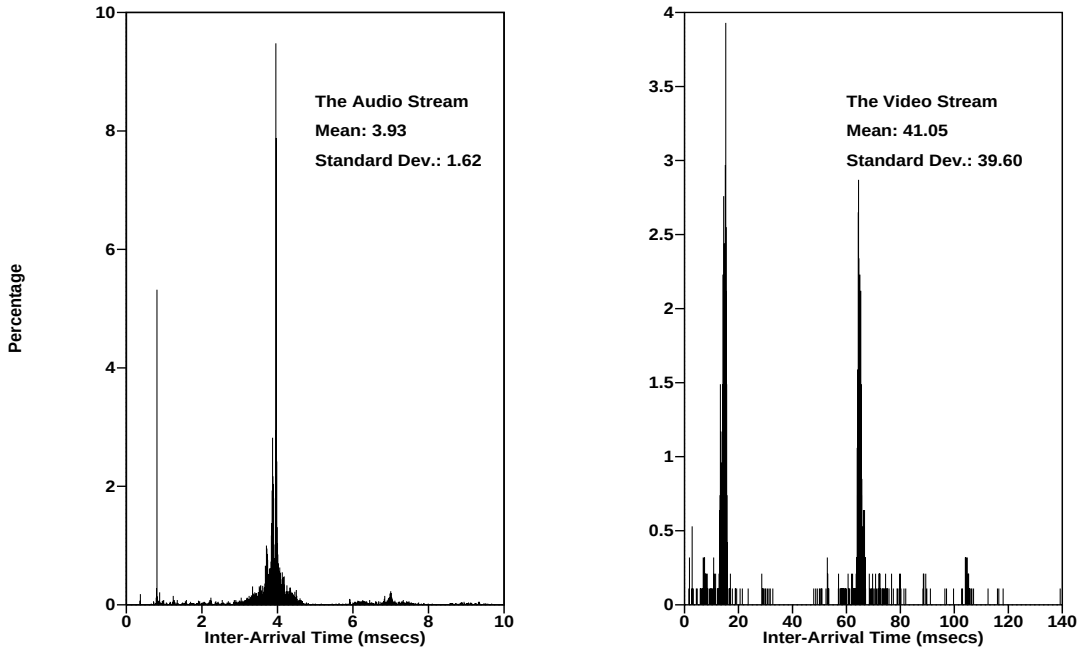


Figure 6.11: Audio and Video Inter-Arrival Times

as outlined in section 5.8. Audio is processed at a higher priority to video which reduces the amount of jitter experienced by the audio thread. Furthermore, the CFR interface is programmed without optimistic reception which increases the jitter experienced by video streams due to the necessity for generating an interrupt on the reception of each CFR cell. Disk allocations are visible in the audio stream between 6 ms and 8 ms, but still a high percentage of packets arrive near the mean of 3.93 ms.

More jitter is introduced in the video stream again due to contention with the audio thread and the CFR interrupt. In this case a CFR interrupt occurs every 276 μsecs. The scheduling of the audio threads every 4 ms, causes a delay in the reception of packets that have been queued by the network, accounting for the shift to the right of the first spike for video stream reception when compared to Figure 6.8. The shift to the left of the second spike is also a result of audio scheduling since the delay introduced by servicing audio packets means partially built video packets exist at the time a video thread is scheduled for reception of the next video packet. Thus, the video thread will not need to block for the full 80 ms during packet reassembly. Although it is difficult to localize, jitter will be introduced in both the audio and video during contention for file system data structures, such as the disk allocation bitmap and disk extent lists. In this case, priority inversion is possible if when a video thread has locked the disk allocation extent list when an audio thread is scheduled and needs to perform a disk allocation. The audio thread will be blocked

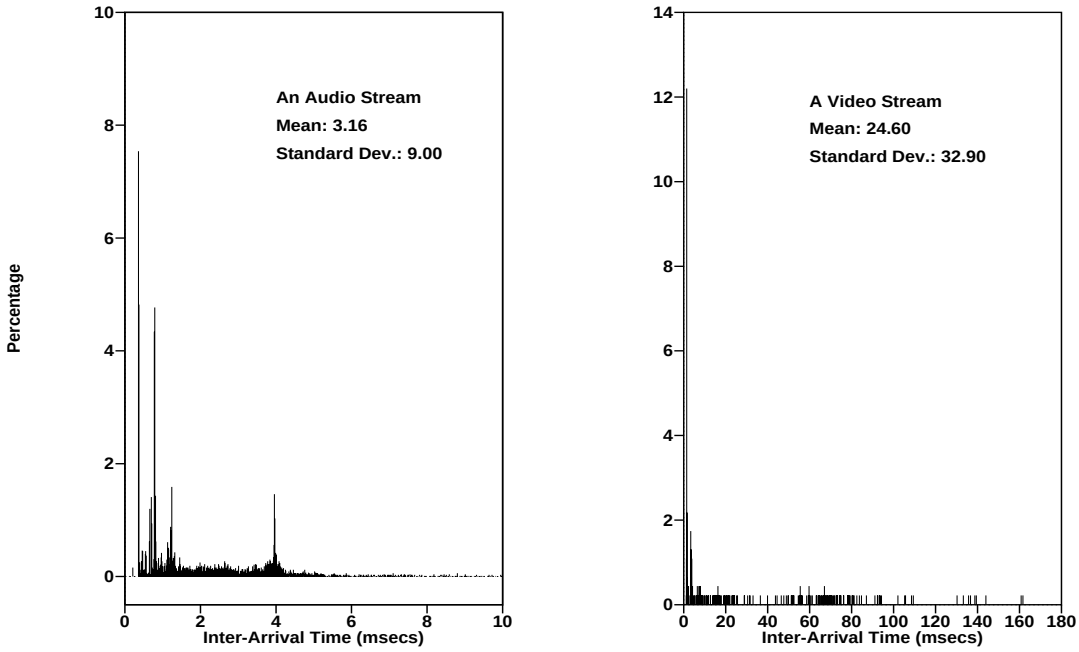


Figure 6.12: Audio and Video Inter-Arrival Times With Two Combined Streams

until the video thread finishes the operation. This situation does not arise very often because allocations are infrequent and the time to perform an allocation is small (approximately 2.5 ms for an extent size of 90 blocks). This includes all block buffer allocations for the extent.

For a single audio and video stream the amount of packet loss experienced by each stream is under 1%, which indicates that there is plenty of CPU to support the data rate of 928 Kbits/s.

Figure 6.12 shows the effect of two audio threads and two video threads running simultaneously. As above, each set of threads is initially synchronised to a common clock, but are otherwise independent. The results are as expected in that the increased contention causes a higher percentage of packets queued in the network protocol code. Again, because audio is processed in preference to video, the effect is less noticeable in the audio threads.

Packet loss in the audio streams was 3% and 4%. Video streams experienced a packet loss of 4% and 6%. The data rate entering at the network interface is 1.86 Mbits/s which corresponds to 7,250 cells per second and one CFR interrupt every 138 μsecs.

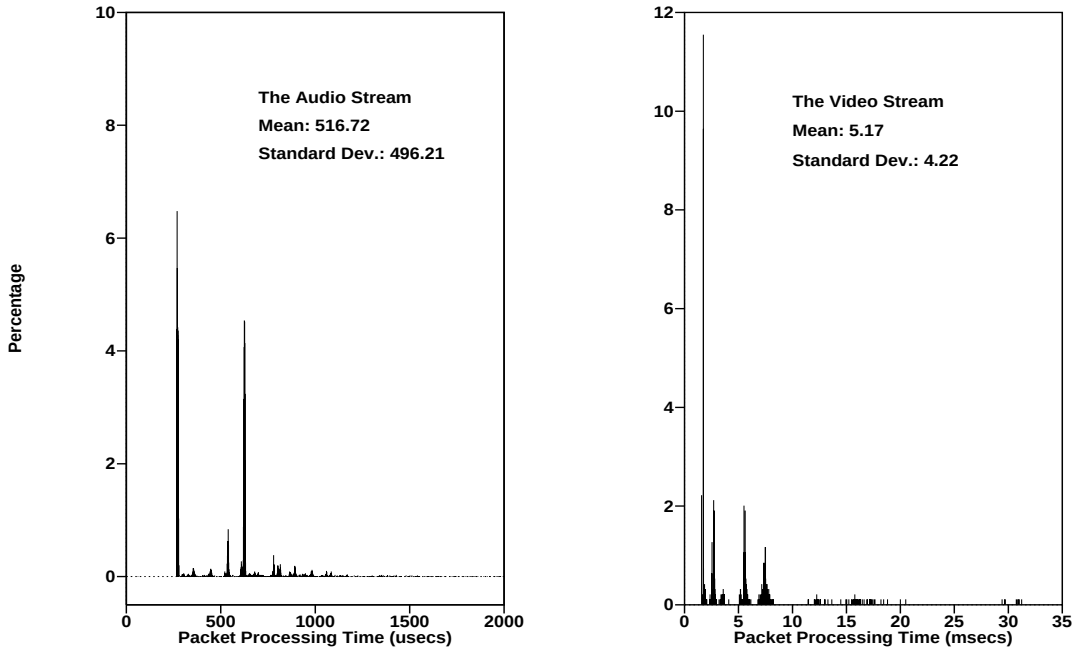


Figure 6.13: Audio and Video Processing Times

Audio and Video Processing

Figure 6.13 shows the processing times for the audio and video streams while recording a synchronised session. The average time for processing an audio packet is nearly twice that of the single audio stream which suggests that the CFR interrupt frequency of one every 276 μsecs significantly increases the elapsed time for an audio thread to process a packet due to the delay of 46 μsecs experienced by processing each CFR cell. However, the processing time is still much less than the inter-arrival time and very little packet loss results from the increase in jitter.

Similarly, contention by the audio threads and the CFR interrupt routine for the CPU increases the average processing time per video packet. In a few cases, the video thread is delayed substantially because the audio thread does not relinquish the CPU after processing each packet. If there are many audio packets queued in the protocol code, the audio thread will continue running which will delay the video thread further.

Figure 6.14 represents the jitter in processing times for an audio and a video stream while two concurrent audio/video sessions are active. Immediately noticeable is the significant delay experienced by the video thread because audio threads operate at a higher priority. The processing time for the audio thread is affected mostly by the increase in CFR processing.

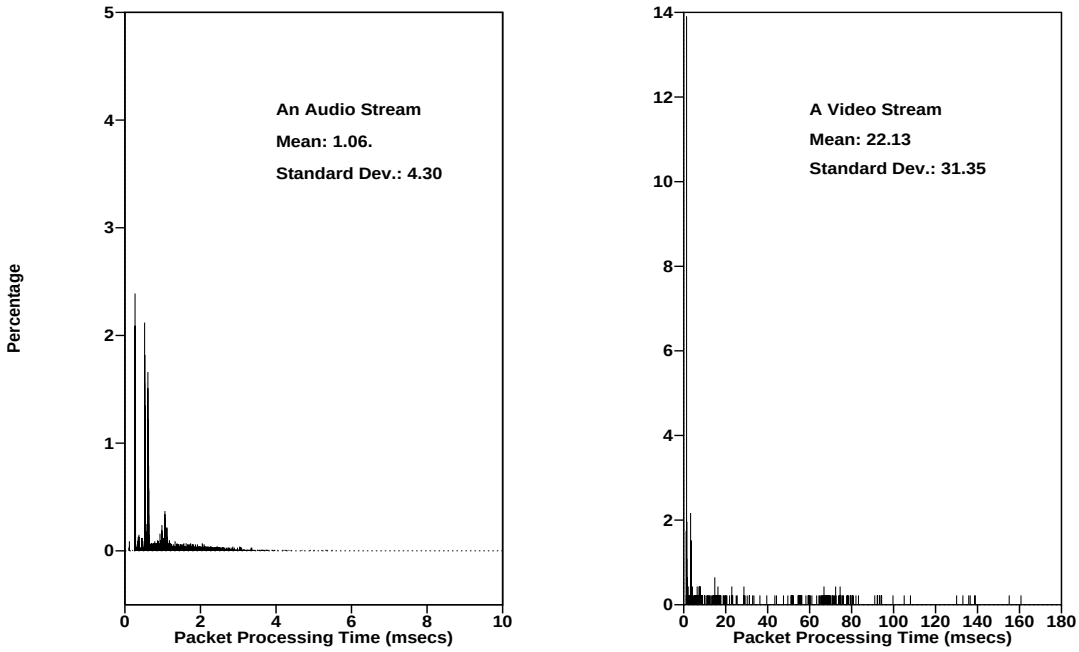


Figure 6.14: Audio and Video Processing Times With Two Combined Streams

6.5.1 Audio and Video Playback

As previously mentioned, there may be an advantage with audio and video playback in that it is possible to vary the frequency of the audio thread by sending more than one audio packet at a time given that there adequate buffering at the sink to handle the incoming data. The following experiment shows the effects of sending ten audio packets every 40 ms instead of one audio packet every 4 ms. The times shown include the amount of time taken by the thread to send each packet across the network.

Figure 6.15 shows the inter-packet read times for a synchronised audio and video streams. The graph is comparable to Figure 6.11 except that the audio stream experiences more jitter in the playback case because the CFR device uses a single FIFO cell queue and no priority is given to audio cells which may enter the queue, even when the audio thread is run at a higher priority. The transmission of video cells which are already on the CFR transmit queue introduces delay to the audio thread.

Figure 6.16 illustrates the effects of sending multiple audio packets during audio processing. This is possible because the Pandora Box contains a 4 *second* audio buffer and regulates the playback of audio more carefully than incoming video packets. Playing multiple audio packets decreases jitter for the majority of audio packets.

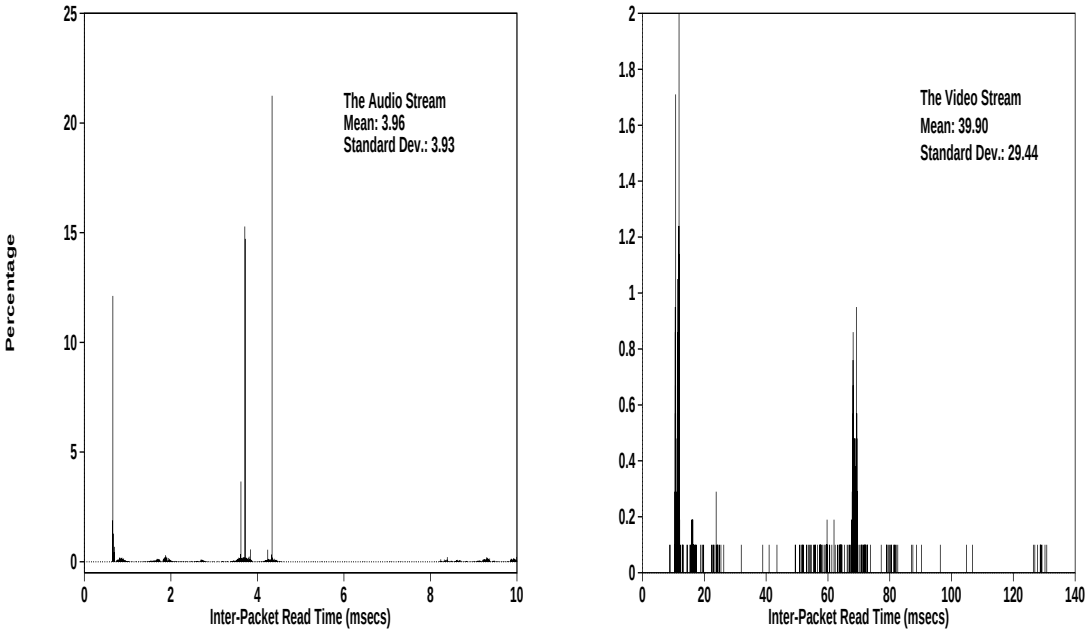


Figure 6.15: Audio and Video Playback Inter-Packet Read Times

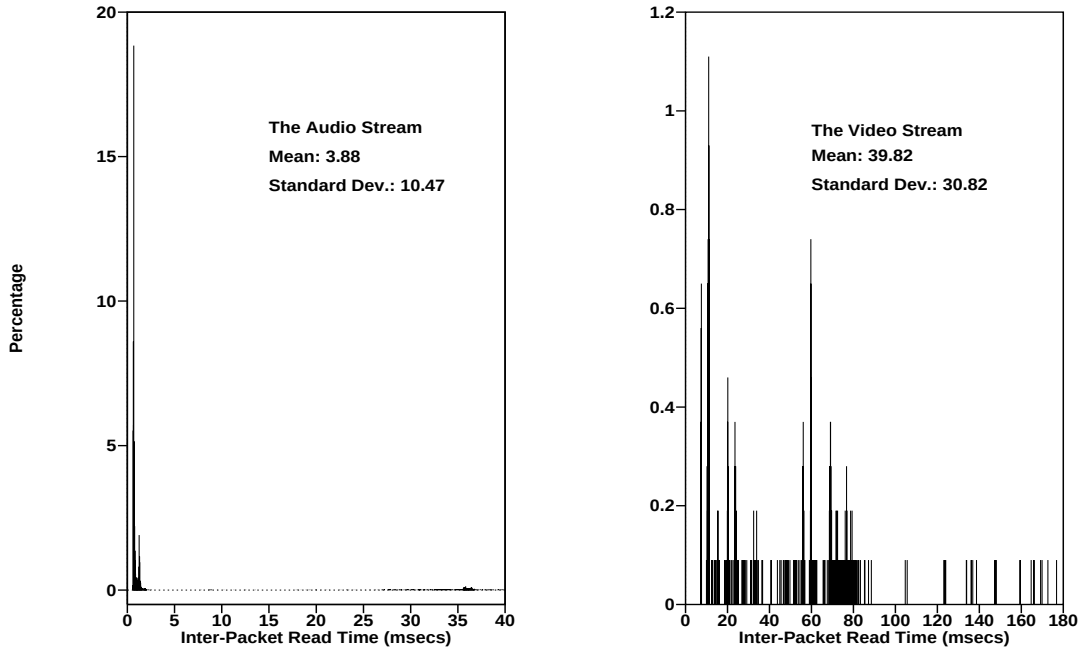


Figure 6.16: Audio and Video Playback With Lower Frequency Audio

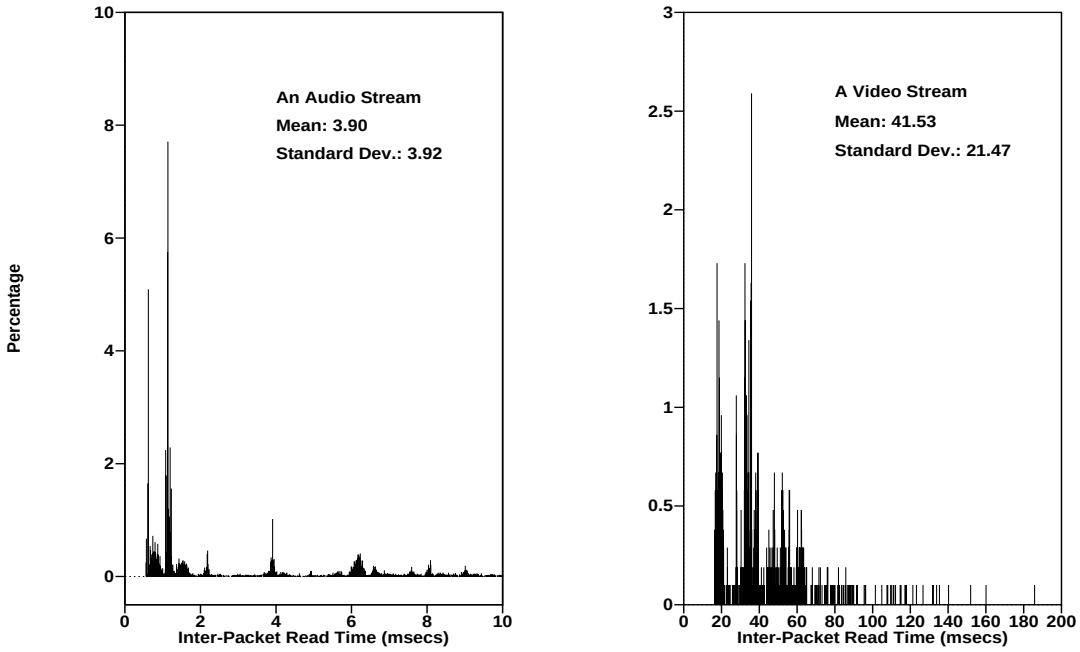


Figure 6.17: Audio and Video Playback With Two Combined Streams

Because the audio thread blocks for a large amount of time after each burst, a small percentage of the packets transmitted, experience a very large delay.

Comparing Figure 6.16 with Figure 6.15 it can be seen that increasing the number of packets transmitted by the audio thread causes greater maximum delay experienced by the video thread and more jitter is introduced in the video stream. The fact that audio timings were taken per packet accounts for the significant number of audio packets with low inter-packet read times.

Figure 6.17 illustrates the jitter introduced in one audio stream and one video streams while two combined streams are active. The amount of jitter experienced by the audio stream is similar to the audio stream in Figure 6.11 which illustrates the playback of one audio/video session. This is because audio streams are processed at a higher priority minimising contention from video streams. The distribution of jitter differs because of the additional CFR interrupts caused by the second audio/video session.

Surprisingly, a video stream experiences less average jitter than in the playback of a single combined session, but a greater maximum jitter value. Again, the distribution differs because of contention with the CFR.

6.6 CMFS Failure

There is a maximum number of streams which is supportable by the CMFS beyond which the quality of service degrades unacceptably. The number of combined synchronised audio and video streams is used since this represents the number of concurrent clients accessing the CMFS assuming video is always viewed with an associated soundtrack. The total capacity for the CMFS was found to be two.

An experiment with three streams resulted in video degradation beyond an acceptable level for continuous playback. Up to 85% packet loss on the video streams was observed. Audio reception for each set of streams remained acceptable. The following is a failure analysis for the hardware and software configuration used for experimentation.

During the reception of 6 concurrent streams (3 audio and 3 video), the total data rate into the CFR interface is 2.8 Mbits/s which is within the measured capacity of 3.2 Mbits/s. However, CFR interrupts occur at 92 μsec intervals and require 46 μsecs to process. Thus, 50% of the total available CPU bandwidth is required for CFR interrupt processing.

Furthermore, a complete audio packet arrives every 1.33 ms. The WANDA kernel, in a single address space (SAS) configuration, imposes a 130 μsec overhead to wake up a user semaphore and context switch to an audio processing thread. From Figure 6.6 the average audio packet processing time is 285 μsecs which includes one data copy from the network buffer to a disk buffer. Thus, the total audio processing time is 415 μsecs.

For three concurrent audio threads, a total of 1.24 ms is needed to complete the processing of one audio packets for each of the three threads, leaving 88 μsecs to schedule another thread in the system. This is not enough time to schedule a video thread at the rate CFR interrupts are occurring. The result is that the video reception buffer overflows and packets are dropped by the network protocol code.

In general, the CMFS fails whenever the CFR interrupt frequency exceeds the maximum user space context switch time of 130 μsecs. This may be understood by considering the following sequence of events. When the network protocol code signals a user thread after a complete packet arrives, the operating system proceeds to schedule and run the signaled thread. However, before the thread has a chance to execute, another interrupt occurs and transfers control back into the kernel, starving the thread of CPU. The result is that the user thread can not process a packets before another is built and queued in the kernel network code.

6.7 Recommendations

From the results discussed in the previous sections a number of recommendations for system support of a continuous media file service can be made. These include:

- Do not perform packet segmentation and reassembly using the host processor.
- Be sure stream prioritisation is effective at all levels.

The experiments and failure analysis of the CMFS clearly show the detrimental effects of interrupt generation on every cell received from or transmitted to the network. In some cases, over 50% of the CPU was spent for network interrupt processing, starving processing threads of needed CPU bandwidth. For this reason, packet segmentation and reassembly should be performed in the network interface or by a separate processor in the system. Some suggestions for improving network interface design are discussed in [**Dixon92**] including DMA of block level data. This poses some problems since the network interface must know about the transmission protocol used, thus limiting the functionality of the interface. This not generally considered a good idea because supporting multiple protocols becomes more difficult and costly. There exists a tradeoff between the generality of the interface and the optimal performance characteristics. However, it is clear that the host processor should not be responsible for network protocol processing.

It is useful to discuss how much the above recommendation will improve the performance characteristics of the CMFS. Assume that the CMFS is connected to the network via an interface which supports block level DMA access. At the maximum utilisation of the CMFS (3 streams), 50% of the CPU is allocated to segmentation and reassembly. At present, a total of 290 cells (230 video cells and 60 audio cells) per stream arrive at the interface every 80 ms, each generating a processor interrupt requiring 46 μsecs to service. If block level interrupts were possible, the number of interrupts per stream would be reduced to 2 for video and 20 for audio in the same time frame; 7.6% of the original interrupt rate. This would reduce the CPU utilisation for interrupt processing from 50% to 3.8%. If the other 50% of the CPU is responsible for user space processing of the streams, then each stream requires 17% of the CPU for data copying to (or from) a disk buffer. This suggests that with block level interrupts, five streams are supportable. The way to increase this figure is to reduce the amount of CPU necessary for user space processing of packets. Presently, 51% of the CPU processing per video packet is needed for data copying. A network interface which would DMA directly to a disk block buffer would eliminate the need for the data copy. This might further increase the total supportable streams in the CMFS to as much as ten.

In section 6.5.1 a source of jitter in different priority streams was the single CFR FIFO used for transmission. Thus, during large video block transmission, small and higher priority audio packets are delayed at the network interface. The use of multiple prioritised FIFOs would improve this situation.

A reduction in thread jitter may be achieved by making disk allocations asynchronous. An experiment was carried out in which only two disk extent allocations were performed at the time of file creation and incoming data was placed in these extents to simulate an environment where allocations are asynchronous. The result did not affect the overall performance characteristics of the system because this was not a performance bottleneck.

A further reduction in stream jitter might be possible with judicious use of priority based semaphores. Using this scheme, low priority threads inside a critical region inherit the priority of the highest priority thread which is blocked on the semaphore for that region. For example, a video thread which is performing a disk allocation would execute at the higher audio priority if an audio thread required an allocation from the same disk. The effect is to reduce the amount of time the audio thread waits for the disk. However, the naive approach to priority inheritance may lead to chained blocking and deadlocks. A priority ceiling protocol proposed in [**Sha90**], discusses a solution to this problem such that a task T is allowed to enter a critical region only if its assigned priority is higher than the priority ceilings of all the semaphores locked by tasks other than T. Further research is required to determine the effectiveness of this scheme in a real implementation.

6.8 Summary

Experimental results of the CMFS were discussed. From the initial experiments with the concurrent disk and network activity, it has been shown that the choice of disk extent size in not important to overall throughput of the network. Instead, this affects the scheduling jitter for high frequency threads in the system. More immediate considerations for the choice of disk extent size are the amount of buffering available to each stream and the frequency of disk access. Reducing the frequency of disk access by using large extents reduces the need for disk scheduling in a system.

Sections 6.3 - 6.5.1 show the effects of interrupt service contention and thread contention on independent stream processing threads. Several recommendations for operating system and network interface development were outlined.

The CMFS can support two concurrent combined audio/video streams. The bottleneck is identified as the CPU as a result of servicing high frequency CFR interrupts and WANDA kernel overhead imposed by context switching threads.

Chapter 7

Conclusion

The previous chapters discuss the design of network based file systems with attention to the differences in low level issues required to support the data rates associated with continuous media. The main thesis of this dissertation is that continuous media file services demand special design consideration because current popular file systems do not support the demands of continuous media. Many classes of file services exist and have been optimised for particular applications. For example, the Unix file system has been optimised for program development in a primarily academic environment. Traditional mainframe file systems cater mostly to database systems. This dissertation has defined a network file system optimised specifically for continuous media under the assumption that specialised distributed services are practical, affordable and desirable.

Fundamental differences in the design follow from the temporal nature of continuous media. The increase in data rates and data volume demand more from a file service than conventional file systems can deliver. There is a need for an admission control policy which constitutes a contract between a client and a server such that the quality of service provided to the client does not degrade beyond that which was negotiated with the server. This policy should take into account each resource in the system to ensure that the algorithm is accurate when any one bottleneck is removed. Using this scheme, a QOS manager may access server state information so that admission control can be effectively administered. This is an extension of the notion of Quality of Service in the network applied to an operating system which manages many other devices. Further research is necessary to decide on an implementation of the QOS manager. However, suggestions on the update of server state information include a device query mechanism using system calls or an RPC mechanism and an event based update of shared state variables.

A file server with an admission control policy differs from conventional file systems where *best effort* service is acceptable. The need for a rate based flow control mech-

anism also reflects the temporal nature of continuous media and explicit controls are necessary to manipulate stream characteristics in a deterministic manner. Finally, intra-stream and inter-stream synchronisation is a consideration when a server based flow control mechanism is used. Both are achieved using an accurate system clock with a resolution on the order of the highest frequency thread expected in the system.

The above issues represent the differences in operation and interface from a conventional file service. These are higher level issues when considering the implementation details of a file system supporting large files. Lower level issues involve the disk layout strategies and CPU scheduling algorithms used. In Chapter 3, a continuous media file system was discussed as characteristic of a soft real-time system. A lightweight micro kernel was used to implement periodic threads which calculate their own deadlines. The kernel is only responsible for scheduling threads on a timer queue which results in a rate monotonic earliest deadline first algorithm. Such a scheme should also be used to schedule interrupt service to reduce the risk of missing thread deadlines, but was not implemented in the prototype. The micro kernel differs from traditional real-time kernels in that no explicit interface for specifying thread periodicity exists.

The criteria for the use of disk striping and disk scheduling was discussed in Chapter 4. The conclusion is that disk scheduling is not necessary unless the disk request rate exceeds a particular threshold which may vary with the disk configuration used. With continuous media data, the disk request rate may be reduced by transferring large contiguous disk blocks. This is well suited to the sequential access nature of the media. An extent based system provides flexibility in choosing the amount of contiguous storage for each disk allocation. A tradeoff between the extent size and disk request rate affects the amount of scheduling jitter experienced by high frequency threads in the server. The use of disk interleaving and separate I/O channels remain an option for increasing disk data throughput, but are currently not warranted when considering the maximum data rates of current network interfaces. Disk interleaving is an advantage when the time to read (or write) an extent of continuous media data approaches the duration of its playback.

A traditional caching strategy is not appropriate for continuous media data, since these data can not be relied upon to exhibit temporal locality. In conventional file servers, re-use of files such as system libraries or application databases is commonplace. Editing of document or program files results in frequent access to these files. Neither of these sources of temporal locality exist to the same extent for applications such as video mail and recorded conferences. This is a very significant difference from a conventional file service where data caching is employed as the primary source of performance improvement.

Chapter 1 outlined four specific aims of the dissertation. The first objective was to

identify parameters that affect continuous media file system performance and Chapters 3 , 4 and 5 discussed these parameters. The second objective was to measure and point out important performance criteria. Measurement of a prototype implementation of the CMFS shows the sources of stream jitter to be contention from network interrupt processing and thread switch overhead associated with concurrent access to the file server. Recommendations for the improvement of network interfaces include segmentation and reassembly performed in the interface and DMA of block level data. Multiple prioritised FIFOs may also improve the jitter characteristics of high priority streams. Further research is necessary to determine the next bottleneck in the system when using this configuration. The third objective was to extend the notion of Quality of Service (QOS) from the network to the operating system. One way of accomplishing this goal is through the use of an admission control algorithm which takes into account resources other than the network. Finally, a general distributed architecture, into which the CMFS may be integrated, is reviewed.

Bibliography

[**Ades87**] Stephen Ades. *An Architecture for Integrated Services on the Local Area Network.* PhD thesis, Cambridge University Computer Laboratory, September 1987. (p 7)

[**Anderson91**] D. P. Anderson, Yoshitomo Osawa, and Ramesh Govindan. *Real-Time Disk Storage and Retrieval of Digital Audio/Video data.* Technical Report, Computer Science Division. University of California, Berkeley, August 1991. (pp 15, 32)

[**Ang92**] Chu Suan Ang. *Continuous Media in Fast Networks.* PhD thesis, Cambridge University Computer Laboratory, January 1992. (pp 5, 23)

[**ANSA89**] Architecture Projects Management. *The ANSA Reference Manual*, March 1989. (pp 4, 52)

[**Bacon91**] Jean Bacon, Ken Moody, Sue Thomson, and Tim Wilson. *A Multi-Service Storage Architecture.* Operating Systems Review, 25(4):47–65, October 1991. (p 51)

[**Borghoff92**] Uwe M. Borghoff. *Catalogue of Distributed File/Operating Systems.* Springer-Verlag, 1992. (p 1)

[**Burrows88**] Michael Burrows. *Efficient Data Sharing.* PhD thesis, Cambridge University Computer Laboratory, December 1988. (pp 2, 38)

[**Calnan88**] Roger Calnan. *The Integration of Voice within a Digital Network.* PhD thesis, Cambridge University Computer Laboratory, September 1988. (p 8)

[**Chen88**] Peter Chen, Garth Gibson, Randy Katz, David Patterson, and Martin Schulze. *Two Papers on RAIDs.* Technical Report UCB/CSD 88/479, University of California at Berkeley, December 1988. (p 46)

[**Dion81**] J. Dion. *Reliable Storage in a Local Network.* PhD thesis, Cambridge University Computer Laboratory, 1981. (p 8)

[**Dixon92**] Michael Joseph Dixon. *System Support for Multi-Service Traffic.* PhD thesis, Cambridge University Computer Laboratory, January 1992. (pp 4, 22, 27, 30, 70, 90)

[**English92**] Robert M. English and Alexander A. Stepanov. *Loge: A Self Organizing Disk Controller.* In USENIX, pages 237–251, Winter 1992. (p 47)

[**Flatman86**] A. V. Flatman. *FDDI Standards for High Speed LANs.* In SPIE Fibre Optics, pages 76–82, 1986. (p 12)

[**Gifford88**] D. Gifford, R.M. Needham, and M. D. Schroeder. *The Cedar File System.* Comm. of the ACM, 31(3), March 1988. (p 7)

[**Gong90**] Li Gong. *Cryptographic Protocols for Distributed Systems.* PhD thesis, Cambridge University Computer Laboratory, April 1990. (p 37)

[**Hopper88**] A. Hopper and R. M. Needham. *The Cambridge Fast Ring Networking System.* IEEE Trans. Computers, 37(10), October 1988. (p 3)

[**Hopper90**] A. Hopper. *Pandora - An experimental system for multimedia applications.* ACM Operating Systems Review, 24(2), April 1990. (pp 4, 14, 28, 64)

[**Karger88**] Paul Karger. *Improving Security and Performance for Capability Systems.* PhD thesis, Cambridge University Computer Laboratory, March 1988. (p 37)

[**Kim86**] Michelle Y. Kim. *Synchronized Disk Interleaving.* IEEE Transactions on Computers, C-35(11):978–988, November 1986. (p 46)

[**Lampson86**] Butler Lampson. *Designing a Global Name Service.* In Proc. 5th ACM Symposium on Principles of Distributed Computing, August 1986. (p 36)

[**Lazar87**] A. Lazar and J. White. *Packetized video on MAGNET.* Optical Engineering, 26(7), July 1987. (p 18)

[**Liu73**] C. L. Liu and James Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment.* Journal of the Association for Computing Machinery, 20(1):46–61, January 1973. (p 29)

[**McAuley90**] D. R. McAuley. *Protocol Design For High Speed Networks.* Technical Report 186, Cambridge University Computer Laboratory, January 1990. Ph.D. dissertation. (pp 4, 13, 23, 71)

[**Metcalfe76**] R. M. Metcalfe and D. R. Boggs. *Ethernet: Distributed packet switching for local computer networks.* Comm. of the ACM, 19(6), July 1976. (p 8)

[**Nelson88**] Michael Nelson, Brent Welch, and John Ousterhout. *Caching in the Sprite Network File System.* ACM Transactions on Computer Systems, 6(1), February 1988. (p 38)

[**Nicolaou91**] C. Nicolaou. *A Distributed Architecture for Multimedia Communication Systems.* Technical Report 220, Cambridge University Computer Laboratory, May 1991. Ph.D. dissertation. (pp 21, 33)

[**Ousterhout84**] John Ousterhout and Fred Douglis. *Beating the I/O Bottleneck: A Case for Log-Structured File Systems.* Technical Report UCB/CSD 88/467, University of California at Berkeley, October 1984. (p 41)

[**Ousterhout85**] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunza, M.Kupfer, and J. Thompson. *A Trace-Driven Analysis of the Unix 4.2 BSD File System.* In In Proc. of the 10th Symposium Principles, December 1985. (p 12)

[**Patterson88**] David A. Patterson, Garth Gibson, and Randy H. Katz. *A Case for Redundant Arrays of Inexpensive Disks (RAID).* In ACM SIGMOD Conference '88, pages 109–116, June 1988. (p 46)

[**Rajunas86**] S.A. Rajunas, N. Hardy, A.C. Bomberger, W.S. Frantz, and C.R. Landau. *Security in KeyKOS.* In Proc. 1986 IEEE Symposium on Security and Privacy, April 1986. (p 37)

[**Richards79**] M. Richards, A. Aylward, P. Bond, R. Evans, and B. Knight. *TRIPOS: A Portable Operating System for Mini-Computers.* Software Proactive and Experience, 9(7), 1979. (p 8)

[**Roth85**] J. Roth, G. Kendall, and S. Decker. *A Network sound System for Unix.* In International Computer Music Conference, August 1985. (p 13)

[**Salem84**] Kenneth Salem. *Disk Striping.* Technical Report EECS-TR-332-84, Computer Science Dept., Princeton University, December 1984. (p 46)

[**Satyanarayanan89**] M. Satyanarayanan. *A Survey of Distributed File Systems.* Technical Report CMU-CS-89-116, Carnegie Melon University, February 1989. (p 2)

[**Schmandt88**] Chris Schmandt and Michael Mckenna. *An Audio and Telephone server for Multi-Media Workstations.* In Proc. Second IEEE Conference on Computer Workstations, 1988. (p 18)

[**Seaborne89**] Andrew Franklin Seaborne. *Filing in a Heterogeneous Network.* PhD thesis, Cambridge University Computer Laboratory, April 1989. (p 51)

[**Sha90**] L. Sha, R. Rajkumar, and J. P. Lehoczky. *Priority Inheritence Protocols: An Approach to Real-Time Synchronization.* IEEE Transactions on Computers, 39(9):1175–1185, September 1990. (p 91)

[**Shultze88**] Martin E. Shultze. *Considerations in the Design of a RAID Prototype.* In Proc. COMPCON 89, March 1988. (p 46)

[**Sreenan92**] Cormac Sreenan. *Synchronisation Services for Digital Continuous Media*, 1992. Cambridge University Computer Laboratory, Ph.D. thesis in preparation. (p 32)

[**Steinberg89**] D. Steinberg and T Learmont. *The MultiMedia File System.* In International Computer Music Conference, November 1989. (p 18)

[**Swinehart83**] D. C. Swinehart, L. C. Stewart, and S. M. Ornstein. *Adding Voice to an Office Computer Network.* In Proc. IEEE Global Telecomm. Conf., November 1983. (p 7)

[**Temple84**] Steve Temple. *The Design of a Ring Communication Network.* PhD thesis, Cambridge University Computer Laboratory, January 1984. (pp 3, 9, 54)

[**Teorey72**] Toby J. Teorey and Tad B. Pinkerton. *A Comparative Analysis of Disk Scheduling Policies.* Comm. of the ACM, 15(3), March 1972. (pp 45, 59)

[**Terry88**] D. Terry and D. Swinehart. *Managing Stored Voice in the Etherphone System.* ACM Transactions on Computer Systems, 6(1), February 1988. (p 7)

[**Vaitzblit91**] Lev Vaitzblit. *The Design and Implementation of a High-Bandwidth File Service for Continuous Media.* Master of Science Thesis, Massachussets Institute of Technology, September 1991. (p 12)

[**Vin91**] Harrick M. Vin and P. Venkat Rangan. *Designing File Systems for Digital Video and Audio.* Technical Report CS91-191, University of California, San Diego, February 1991. (p 10)

[**Wilkes79**] M. V. Wilkes and D. J. Wheeler. *The Cambridge Digital Communication Ring.* In Proc. Local Area Communication Network Symposium, May 1979. (p 8)

[**Wilson92**] Tim D. Wilson. *Increasing the Performace of Storage Services for Local Area Networks.* PhD thesis, Cambridge University Computer Laboratory, April 1992. (pp 24, 63)