ABSTRACT

While emerging networking technology is capable of supporting the high data-rates and the real-time requirements of continuous media such as digital audio and video, file systems and storage technologies are unable to provide the necessary support.

The real-time nature and bandwidth requirements of multimedia objects fundamentally distinguish them from conventional file data. Conventional file servers are not designed to accommodate the real-time nature and access semantics of multimedia objects.

This thesis describes the design and implementation of a Continuous Media File Service (CMFS) suitable for the needs of multimedia computing. The central issues addressed by the design and implementation of CMFS are high storage bandwidth, synchronization of multimedia streams, admission control policies that guarantee continuous access to storage devices, and a user interface that supports multimedia abstractions.

High storage bandwidth is provided by interleaving multiple disks and processing nodes using synchronous disk and node striping. Synchronous striping is shown to be optimal for organizing data on multiple storage devices, given that the workload consists predominantly of sequential access to large files.

Admission control is based on the deadline driven algorithm for scheduling periodic tasks.

Synchronization of related data streams is facilitated by the inclusion of time stamps in every frame and the ability to reference frames by temporal file offsets.

Access paradigms provided by CMFS for multimedia objects include a file composition facility, directional, skip-frame, and broadcast file access, and access to frames by temporal offsets.

A prototype of CMFS was built using the Unix\textsuperscript{1} operating system and an FDDI-based local-area network. The performance results showed that the data-rates supported by CMFS for sequential reads scale linearly with the number of nodes. In similar configurations, CMFS performed as well as the local Ultrix file system and outperformed NFS by a factor of 1.5. Configured on 2 nodes with 2 disks on each node, CMFS was able to provide a read data-rate in excess of 3.5 Mbytes/s, outperforming the Unix file system by a factor of three and NFS by a factor of four.

A VCR-like application for managing audio and video was developed to evaluate the ability of CMFS to support continuous media. Subjective evaluation of the application has shown that the quality of the produced real-time video display was similar to that of an NTSC broadcast.

\textsuperscript{1}Unix is a registered trademark of Bell Laboratories
# Contents

1 Introduction ........................................................................................................... 7  
  1.1 Motivation ....................................................................................................... 7  

2 Analysis of Multimedia Objects ........................................................................... 9  
  2.1 Characteristics and Requirements of Multimedia Objects ......................... 9  
  2.2 Multimedia Objects versus Conventional Files ............................................. 12  
  2.3 Goals ............................................................................................................... 16  

3 Background and Related Work .......................................................................... 18  
  3.1 ATM Networks ............................................................................................... 18  
  3.2 Synchronization ............................................................................................. 18  
  3.3 Storage technology ......................................................................................... 19  
  3.4 File Systems for Digital Audio and motion Video ......................................... 20  
  3.5 Contributions to the CMFS design ................................................................. 22  

4 CMFS Model ....................................................................................................... 23  
  4.1 Functional Model ........................................................................................... 23  
  4.2 Design Model ................................................................................................ 25  

5 CMFS Interface .................................................................................................. 28  
  5.1 Client Interface Routines .............................................................................. 28  
  5.2 Discussion of the CMFS Services and Interface .......................................... 33  
  5.3 Protocol Design Alternatives ....................................................................... 35  

6 Administration Server ....................................................................................... 38  
  6.1 Directory Services .......................................................................................... 38  
  6.2 Admission Control ......................................................................................... 38  

7 Storage Server .................................................................................................. 44  
  7.1 Storage Servers as a Logical Storage System .............................................. 44  
  7.2 Initialization .................................................................................................... 46  
  7.3 Data Structures ................................................................................................ 46  
  7.4 Disk Striping .................................................................................................... 51  
  7.5 Processing of Requests ................................................................................ 59
7.6 Recovery ........................................................................ 64
7.7 Reliability and Fault Tolerance ...................................... 65

8 Prototype Implementation .................................................. 67

9 Experimental Results ....................................................... 73
  9.1 Performance of the System Components ..................... 73
  9.2 CMFS Performance .................................................... 79
  9.3 Subjective Evaluation .................................................. 82
  9.4 Implications of CMFS Performance Results .................. 84

10 Conclusions and Suggestions for Future Work ................ 86
  10.1 Summary and Conclusions ......................................... 86
  10.2 Suggestion for Future Work ....................................... 88

Acknowledgments .................................................................. 91

Figures

Figure 4-1 CMFS Functional Model ...................................... 23
Figure 4-2 CMFS Design Model .......................................... 25
Figure 5-1 C Declarations for the CMFS Interface Routines .... 29
Figure 7-1 Storage Domains ............................................... 44
Figure 7-2 Synchronous Disk and Node Striping ................. 45
Figure 7-3 Container Descriptors ........................................ 48
Figure 7-4 Physical Map .................................................... 49
Figure 7-5 Frame Map ....................................................... 49
Figure 7-6 Temporal map ................................................... 50
Figure 7-7 Uniform Striping ............................................... 52
Figure 7-8 3-Frame Striping ................................................ 53
Figure 7-9 Asynchronous Striping ....................................... 54
Figure 7-10 Striping Structures .......................................... 58
Figure 7-11 Container as a Log-Structured File .................. 63
Figure 8-1 Hardware Configuration ...................................... 68
Figure 9-1 System Components .......................................... 74
Figure 9-2 Control Panel for Playback .............................. 82
Figure 9-3 Control Panel for File Composition .................... 83

Tables

Table 2-1 Bandwidth Requirements for Digital Audio and Video .... 11
1 Introduction

This thesis describes the design and implementation of a high-bandwidth file service for continuous media. Continuous media are real-time multimedia objects such as digital audio and video. A Continuous Media File Service (CMFS) is distinguished from a conventional file service by its ability to support storage and access requirements associated with real-time continuous media. The design of CMFS focuses on the following issues: high-bandwidth to storage devices, synchronization of multimedia streams, performance guarantees, and access paradigms suitable for real-time multimedia objects.

1.1 Motivation

The advent of high-bandwidth networks prompted a research interest in distributed multimedia computing. While the emerging networking technology is capable of supporting the high data-rates and the real-time requirements of continuous media, file systems and storage technology are unable to provide the necessary support. The development of CMFS is motivated by the inability of conventional file systems to support access and storage requirements of the emerging distributed multimedia applications.

Advances in Communications

Recent advances in lightwave and fiber optic transmission technologies have resulted in dramatic increases in network bandwidth for local and long distance communications [HENRY88], [MAT89]. FDDI, a 100 Mb/s optical ring is already available as a commercial product [ROSS86]. Prototypes of networks that support an aggregate throughput in the gigabit range have been developed [MAT89],[CIDON88]. Commercial introduction of gigabit networks, driven by the needs of imaging and multimedia applications, are inevitable. Although one gigabit may seem to be a remarkably high network capacity, it is only a modest fraction of the full physical capacity of optical fiber, which is measured in tens of thousands of gigabits. The desire to realize the full bandwidth capacity of optical fiber will drive further research in optical devices, quite possibly resulting in networks carrying a total traffic of 50,000 gigabits [HENRY88].

The rapid evolution in telecommunications has lead to the emergence of new switching technologies capable of supporting a variety of communication services with a wide range of transmission rates [AHM89],[NEWM88]. The high bandwidth capacity of optical fiber and the new switching techniques form the core technologies for developing broadband multiservice networks. Broadband integrated networks will provide an efficient and flexible transport mechanism for conventional computer data as well as real-time traffic such as voice and video.

Multimedia Computing

High-bandwidth multiservice networks serve as the enabling technology for distributed multimedia applications. Multimedia computing is already attracting a great deal of academic and commercial interest [FOX89],[FOX91]. Researchers are optimistic that the use of media types such as audio and video in computer systems will greatly enhance the
communication bandwidth attainable between the computer and its users and between
users communicating via computer. The perceived benefits of multimedia computing
suggest two broad classes of applications: single user learning and training systems, and
systems for cooperative working. The majority of existing multimedia systems focus on
the first class of applications by providing interactive learning tools in a single user PC
environment. Broadband multiservice networks will provide a great impetus and the
technological basis for developing the cooperative applications for multiple users such as
conferencing, design, and authoring.

In addition to cooperative work, it is expected that the advent of integrated digital net-
works will create a demand for new multimedia applications such as video telephony,
retrieval and distribution of information [ARMB86]. Several projects are already investi-
gating the use of distributed multimedia systems in a packet switching environment
[HOP90], [LEU90].

File Systems
An important component of distributed multimedia applications is a file service. Unfor-
tunately, file systems and associated storage technologies have not kept pace with rapid
developments in communications and networking. Limited by secondary storage
throughput, main memory and I/O channel bandwidths, conventional file servers cannot
sustain the high data rates that are achievable on today’s networks, nor can they provide
support for real-time traffic. The temporal nature and bandwidth requirements of
multimedia objects, such as digital audio and video, fundamentally distinguish them
from conventional data. Conventional file systems are not designed to accommodate the
real-time nature and access semantics of multimedia objects. A file system capable of
matching the requirements of real-time multimedia objects is needed.

The rest of the thesis is organized as follows:

- Chapter 2 analyzes the characteristics and requirements of real-time multimedia
  objects and establishes the CMFS design goals
- Chapter 3 reviews technologies and research relevant to the design of CMFS
- Chapter 4 introduces CMFS functional and design models
- Chapters 5 through 7 describe the design of CMFS components in detail
- Chapters 8 describes the prototype implementation
- Chapter 9 presents the experimental results
- Chapter 10 presents the conclusions and suggestions for future work.
2 Analysis of Multimedia Objects

This chapter analyzes the characteristics and requirements of multimedia objects, emphasizing the fundamental differences between these media types and conventional files. The analysis reveals the inability of conventional file systems to provide support for continuous media and establishes a framework for the design of CMFS.

2.1 Characteristics and Requirements of Multimedia Objects

Multimedia objects are characterized in terms of their temporal behavior, storage requirements, and access patterns.

Temporal Nature

Real-time multimedia objects such as audio and motion video possess temporal characteristics that fundamentally distinguish these media types from conventional data. The temporal characteristics of voice and video are isochronous in nature; that is, finite sized samples must be generated at fixed time intervals.

In order to reconstruct continuous speech, digitized voice samples must be received and played back at the same frequency at which the original voice signal was sampled. Excessive delays in sample arrivals may cause audible sound distortions. The sampling frequency imposes a stringent upper bound on the time delay between successive samples.

A similar requirement exists for video streams. In order to render continuous motion, video frames must be received and displayed at a frequency of at least 24 frames per second. Lower frame rates will degrade the continuity of motion portrayal.

The temporal nature of voice and video imposes a stringent requirement for sustained bandwidth. The sustained bandwidth is described by two parameters: the size of each sample and the maximum time delay between successive samples. These parameters provide sufficient information to the sender and the receiver to allocate the resources necessary to sustain the isochronous flow of data.

The use of data compression may result in a stream with variable sample sizes. In this case, the instantaneous bandwidth requirement of the stream is a function of the sample sizes. The peak bandwidth requirement is imposed by the sample with the largest size.

Performance guarantees for sustained bandwidth must include the ability to support the peak bandwidth requirement.

Effects of Compression on Performance Guarantees

Compression of audio and video does not alter the temporal characteristics of these media types. Compression reduces the size of samples; however, the time delay between successive samples remains unaffected.
Two methods exist for compressing a data stream: fixed and variable bit-rate compression. Fixed bit-rate compression generates equal size samples and is typically employed in deterministic circuit switching environments. Variable bit-rate compression generates variable size samples and is typically employed in statistically multiplexed packet switching environments.

Peak bandwidth guarantees for streams with variable size samples may be provided deterministically or stochastically. The deterministic guarantees are based on the largest sample size of each stream. The stochastic guarantees are based on the statistical multiplexing of the instantaneous bandwidths requirements of multiple streams.

**Synchronization**

The isochronous properties capture one aspect of the temporal nature of voice and video: the continuity of the flow of data. Another factor that contributes to the temporal nature of multimedia objects is the need to synchronize related streams. A simple example of synchronizing related streams is "lip-synching", the synchronization of spoken voice with the movement of the speaker’s lips. More complex temporal relationships may exist among multiple media streams, requiring more sophisticated synchronization schemes.

**Storage Requirements**

Storage requirements for a video or an audio sequence are determined by the sampling rate, sample size, and the length of the sequence. Typical sizes of these media types at common sampling rates are shown in Table 2-1.

It is evident that even compressed short sequences of hi-fi audio and especially digital video impose high demands on storage and bandwidth. The storage requirements for digital video are particularly high, given that frame sizes are large and a logical scene lasts on average a few seconds [VER88].

**Access Patterns**

It is reasonable to assume that access to multimedia objects will consist mostly of sequential read requests. The majority of existing and proposed multimedia applications serve as evidence in support of this assumption. Asymmetric compression algorithms requiring a complicated encoder and a simple decoder have been developed for motion video [RIP89],[LIPP89]. These algorithms are optimized to fit the predominantly read access modes of multimedia applications. Recent experience with imaging file systems provides further evidence for read-mostly workloads [EDW89].

Sequential read access to multimedia objects is typically performed by real-time playback applications which require performance guarantees for sustained bandwidth.
A less frequently used workload consisting of random access to multimedia objects is likely to result from non real-time applications, such as editing or multimedia slide presentations. These applications, however, do not require guarantees of sustained bandwidth, since the operations that they perform may be fulfilled in "virtual-time" rather than real-time.

Thus, the workload consisting of sequential reads is the most common and also the most demanding in terms of the performance requirements. Therefore, while it is important to support random access to multimedia objects, the ability to support the sequential read access is critical.

<table>
<thead>
<tr>
<th>Media Type</th>
<th>Sampling Rate</th>
<th>Size per second</th>
<th>Frame Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>original</td>
<td>compressed</td>
</tr>
<tr>
<td>digitized voice</td>
<td>8KHz/8bit</td>
<td>8KB</td>
<td>2KB</td>
</tr>
<tr>
<td>CD audio</td>
<td>44.1KHz/16bit/2ch</td>
<td>176.4KB</td>
<td>44.1KB</td>
</tr>
<tr>
<td>NTSC color</td>
<td>13.5/6.25/6.25 MHz Y/Y-R/Y-B 8/8/8 bits/sample</td>
<td>27MB</td>
<td>1.1MB</td>
</tr>
<tr>
<td>NTSC monochrome</td>
<td>13.5MHz/8bit</td>
<td>13.5MB</td>
<td>550KB</td>
</tr>
<tr>
<td>HDTV</td>
<td>30 frames/sec</td>
<td>90MB</td>
<td>3.2MB</td>
</tr>
</tbody>
</table>

**Table 2-1 Bandwidth Requirements for Digital Audio and Video**

The characteristics of multimedia objects are summarized as follows:

- large sample sizes (video frames)
- large files (video and audio sequences)
- temporal in nature
  - require sustained bandwidth
  - require synchronization

---

1 Subband or Code Excited Linear Prediction (CELP) 4:1 compression ratio is assumed for audio, MPEG 24:1 compression ratio is assumed for video; B represents bytes, b represents bits
• accessed predominantly for sequential reading
  - require performance guarantees

2.2 Multimedia Objects versus Conventional Files

In the previous section we characterized multimedia objects in terms of their temporal behavior, bandwidth requirements, and access patterns. In this section we compare the requirements and characteristics of multimedia objects to the corresponding attributes of conventional files and show that they are fundamentally different. The attributes of conventional files are discussed in terms of their influence on the design of conventional file servers. The purpose of the discussion is to reveal that conventional file servers, designed to fit the attributes of conventional files, are unsuitable for real-time multimedia objects.

Conventional files differ fundamentally from multimedia objects in size, performance requirements, resource sharing, and scalability notions.

File Size and Locality

Conventional files are relatively small, on the order of 4 to 8K bytes [OUST85]. The inability of Unix file systems to span multiple disk devices and the fragmentation of blocks for disk space allocation in Berkeley’s FFS [MCK84] are examples of design choices that are influenced by the size attribute of conventional files.

Conventional file systems commonly use caching to exploit locality in file access patterns. Locality is the original motivation for implementing a file buffer cache. Since most files are accessed sequentially and read in their entirety¹ [OUST85], caching can only benefit performance if the size of the cache greatly exceeds the average size of a file. Thus, caching in traditional file systems works only because of the small size of an average conventional file. Furthermore, the small file size becomes a requirement for systems that cache entire files, such as Bullet [REN89] and the early versions of AFS [SAT85].

As shown in Table 2-1 of the previous section, audio and video objects tend to be extremely large. The enormous size of these media types precludes caching of an entire file in main memory.

Partial caching of a large multimedia file benefits performance of a file system only if access to the cached portion of the file exhibits temporal locality. It is reasonable to assume temporal locality in editing applications and in situations where multiple synchronized readers are accessing the same file simultaneously, as in broadcasting, for example. However, temporal locality of access to a portion of a file is unlikely in the general multimedia environment characterized by independent clients reading entire files sequentially. In this environment a read-ahead is more beneficial to performance than

¹This observation applies to Unix file system environments
caching. Thus, unlike a conventional file system, a multimedia file system must dynamically adapt its caching and buffering strategies to fit the characteristics of the application.

**I/O Request Sizes**

Many application programs request I/O in small units ranging from a few hundred bytes to a few kilobytes [OUST88]. As the result, conventional file systems set the disk block size to 4 or 8K bytes [LEF89]. When the disk transfer units are small, the dominant delay in fulfilling an I/O request is caused by the disk access time. In recognition of this fact, novel file system designs, such as log-structuring, attempt to increase the file system performance by reducing the disk seek time.

However, it is reasonable to expect, that multimedia applications, especially those that operate on digital video, will request I/O in large units; for example, a convenient I/O request unit is a video frame, which is much larger than a standard file block. When the disk transfer unit becomes large, the dominant delay factor in fulfilling an I/O request shifts from the disk access time to the disk data transfer time.

Thus, performance optimizations to a file service that supports large objects must focus on reducing the disk data transfer time as well as the access time.

**Performance Requirements**

Conventional files lack the temporal dimension that is present in real-time multimedia objects. I/O requests to conventional files do not need to be fulfilled within strict time constraints. As the result, conventional file systems provide "best effort" service, without quantitative performance guarantees.

Clients of conventional file systems typically judge performance subjectively, based on the average response time of an I/O request. The response time varies stochastically with the number of clients that are sharing the file system concurrently. Small variations in the response time do not normally degrade the quality of services provided by most applications. Thus, "best effort" file service is acceptable in conventional environments.

In contrast, I/O requests to real-time multimedia objects must be fulfilled within strict time constraints; variations in the response time may severely impair the quality of services provided by multimedia applications. Such applications need strict guarantees of sustained peak bandwidth.

**Resource Sharing**

A file system comprises a set of resources such as the CPU, memory, and disk bandwidth that are used to service the I/O requests. The maximum capacity of the available resources determines the maximum aggregate bandwidth and request service rate that can be supported by the file system.
A hypothetical single user file system may dedicate all its resources to one client. Such a hypothetical file system environment is completely deterministic, since the client's performance requirements and the maximum capacity of the file system are known.

In practice, however, a file system and its resources are shared by multiple concurrently active users.

In conventional file systems, resource sharing is based on the principle of statistical multiplexing; that is, the file system commits to processing all active requests, hoping that the aggregate load presented by concurrently active clients does not exceed the capacity of the available resources. The aggregate bandwidth requirement imposed on the file system is an instantaneous statistical sum of the individual data transfer rates of all active clients.

The process of statistical multiplexing is justified if the probability that the average aggregate load exceeds the maximum resource capacity is low. The probability of exceeding the maximum resource capacity is an increasing function of the average aggregate load presented to the file system; in turn, the average aggregate load is an increasing function of the number of concurrently active clients. We will illustrate the relationship between the number of active clients and the probability of exceeding the capacity of a file system by the following numerical example.

A study of the Unix 4.2 BSD file system observes that the average data transfer rate for a single user in a conventional environment is 2 Kbyte/s, with standard deviation of 4 Kbyte/s [OUST85]. For the sake of this example, suppose that the maximum aggregate data transfer rate supported by a file service is 12 Kbyte/s. Assume that the cumulative load imposed by multiple clients is normally distributed\(^1\). Table 2-2 shows the probabilities of exceeding the maximum capacity of 12 Kbyte/s for a varying number of concurrently active clients\(^2\).

In general, the probability of exceeding the maximum capacity is an increasing function of the ratio of the average aggregate load to the maximum capacity.

The process of statistical multiplexing works well for conventional files precisely because the ratio of the average aggregate load to the maximum capacity is extremely small. For example, Ousterhout observes that the average number of concurrently active users in the system is about 5 [OUST85].

Given the previous assumption of the normally distributed cumulative load, the average transfer rate per user of 2 Kbyte/s with standard deviation of 4 Kbyte/s, the probability that 5 active clients will exceed the 1 Mbytes/s capacity of a SCSI disk is less than \(8 \cdot 10^{-5}\), according to the Chebyshev's inequality.

---

\(^1\) The validity of this assumption is irrelevant, since we only aim to illustrate the relationship that exists between the number of active users and the probability of exceeding the maximum capacity of the file system. However, we note that this assumption is, in fact, reasonable, given that the number of active clients is sufficiently large to justify the central limit theorem.

\(^2\) We assume that each client transfers on average 2 Kbyte/s with standard deviation of 4 Kbyte/s.
<table>
<thead>
<tr>
<th>Number of Active Clients</th>
<th>Mean Transfer Rate</th>
<th>Standard Deviation</th>
<th>Probability of Exceeding 12 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>.01</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>.1</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>8</td>
<td>.31</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>10</td>
<td>.5</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>11</td>
<td>.62</td>
</tr>
</tbody>
</table>

Table 2-2 Probability of Exceeding Maximum Capacity

In the multimedia environment, the ratio of the average aggregate load to the maximum capacity is high. Given the current storage technology, it is unlikely that a file system can support a large number of users with the bandwidth requirements of continuous media such as digital video. For example, a single SCSI disk with the transfer rate of 1 Mbytes/second can support only 5 video streams, each compressed to the MPEG specified rate of 1.5 Mbits/second [LeGall91]. Assuming that 1.5 Mbits/s is the average rate, the probability that 5 clients will exceed the SCSI disk capacity is 50%, which is orders of magnitude higher than the probability calculated in a similar environment for conventional files.

Thus, resource sharing using the statistical multiplexing is inappropriate in the multimedia environment.

We showed earlier that the temporal nature of multimedia objects requires guaranteed sustained bandwidth. Since the best effort service based on the statistical multiplexing is inappropriate for real-time multimedia objects, resource sharing must be performed via a reservation scheme.

In order to make performance guarantees, a file system needs to reserve storage bandwidth deterministically, based on the peak requirements of all user, or ensure stochastically that the instantaneous bandwidth resulting from the statistical multiplexing does not exceed the system capacity.

\(^1\) Mean transfer rates and standard deviations are in units of Kbyte/second.
**Scalability**

Finally, the notion of scalability acquires a new dimension in the context of multimedia objects. The notion of server scalability, as espoused by traditional file systems, denotes the server's ability to provide the same level of performance as the number of users grows [HOWARD88]. Multimedia applications, however, may be more concerned with how much bandwidth a file server can provide to a single user than with how many users can be supported by the file server. As network speeds increase, it becomes reasonable to extend the notion of scalability to include the file server's ability to utilize increased network bandwidth. Table 2-3 summarizes the differences between multimedia objects and conventional file data.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Conventional Files</th>
<th>Multimedia Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>small, 4-8Kbytes</td>
<td>extremely large</td>
</tr>
<tr>
<td>I/O request unit</td>
<td>small (300-4000 bytes)</td>
<td>large (225-725 Kbytes)</td>
</tr>
<tr>
<td>performance</td>
<td>best effort service</td>
<td>guaranteed peak bandwidth</td>
</tr>
<tr>
<td>requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>resource sharing</td>
<td>statistical multiplexing</td>
<td>reservation (access control)</td>
</tr>
<tr>
<td>scalability</td>
<td>scale with the number of users</td>
<td>scale with network bandwidths</td>
</tr>
</tbody>
</table>

**Table 2-3 Multimedia Objects vs. Conventional Files**

**2.3 Goals**

The requirements of multimedia objects explored in the previous section determine the goals for the design of CMFS. The goals are summarized as follows:

- High bandwidth - provide high data transfer rate, relative to conventional file systems, by using multiple storage devices and processing nodes concurrently.
• Bandwidth scalability - provide the ability to utilize increased network speeds and data throughputs. The server should be able to grow modularly in order to sustain the maximum data rates that a given network offers.

• Large objects - provide support for large file and frame sizes

• Performance guarantees - provide guaranteed sustained peak bandwidth for continuous access to data.

• Synchronization - provide the ability to synchronize related media streams

• Multimedia abstractions - provide useful abstractions for developing multimedia and video based applications.

The file server described in this thesis deals only with those issues that are related to the stated goals. Many issues addressed in conventional file systems, such as atomicity, transactional facilities, availability and reliability [SVO84] are beyond the scope of our goals. Issues of server management and crash recovery are only discussed to the extent that they apply to the prototype implementation.
3 Background and Related Work

This chapter reviews the technology and previous research work that are relevant to the design of CMFS. The selection of technologies and research projects for this review is guided by the extent to which they relate to the goals stated in the previous chapter. The following areas in computing research have relevant contributions to the design of CMFS: networks, synchronization semantics, storage technology, and file systems for digital audio and video. The following sections briefly review recent developments in each area.

3.1 ATM Networks

Asynchronous Transfer Mode (ATM) is recommended by the CCITT as the basis of multiservice telecommunications [CCITT88]. In ATM environments, the small fixed size cell is the primary unit of transmission, switching, and multiplexing. Each cell, usually 32 to 48 bytes long, contains a short header used for routing, followed by user data [HAN89], [RIDER89]. Keeping the cell size small minimizes the jitter caused by the packetization and switching delays and reduces the impact of an occasional cell loss on performance.

ATM implementations rely heavily on switching and multiplexing technologies that induce very little delay and jitter. Typically, these systems utilize space division switching, also known as fast packet switching [AHM89],[NEW88]. In effect, ATM performs fast packet switching in hardware. ATM provides flexible and efficient transport for multiservice traffic by combining the rate adaptation and statistical multiplexing of packet switching with the low delay and jitter characteristics associated with circuit switching [TEN89].

It has been shown that the statistical multiplexing nature of ATM is well suited for a variable bit-rate video coding, thus making ATM a viable choice for the transport of packetized video [VER88],[MAG87],[KARL88].

In summary, the importance of ATM lies in its promise to provide a universal service. As a multiservice digital network, ATM will enable a variety of multimedia applications, both for commercial and research use.

3.2 Synchronization

The previous chapter described the need to synchronize related media streams. The most basic approach to synchronizing related streams is to include time stamps within each stream during recording. On playback, the time stamps may be used to achieve inter-stream synchronization. This approach is taken by the Etherphone voice server [TERR88] and the Pandora video file server [WIL91].

A higher level of abstraction may be provided by a synchronization protocol based on the time stamps within each stream. A multi-service flow synchronization protocol is proposed by Escobar, et al, in reference [ESCO91]. The protocol relies on the presence of a global clock in the network and the ability to synchronize local clocks to the global
clock within a few milliseconds. Given this assumption, the protocol achieves synchronization by equalizing the end-to-end delay.

Extensions to the OSI transport level protocol to support synchronization are proposed by Salmony and Shepherd in reference [SALM89]. The proposed synchronization scheme is based on the inclusion of synchronization markers within each stream and the addition of a synchronization channel. Simple synchronization, such as "lip-synching", may be achieved by aligning the stream markers. More complex temporal relationships among multiple streams may be encoded into a tree structure and transmitted on the synchronization channel. The tree structure specifies time alignment of related streams by referencing byte positions or packet sequence numbers of the corresponding streams.

A synchronization scheme that does not rely on the inclusion of time stamps within each stream is presented by Nicolau in reference [NIC90]. A receiving application detects the loss of synchronization if the average jitter per sample falls below an acceptable threshold. Jitter is defined as the difference between the time delays of two successive samples. Correcting action can take the form of modifying the Quality of Service (QOS) properties of the underlying communication subsystem.

Discussion

Most of the proposals for synchronizing related streams are based on the inclusion of time stamps within each stream. This approach seems to be fundamental in that it allows great flexibility in implementing higher level and more sophisticated synchronization schemes. Nicolau's proposal [NIC90] of synchronization based on the rate of change of the jitter per sample is limited in two ways:

1. it precludes more sophisticated synchronization schemes
2. it implies that the actual synchronization is performed by the sender; the receiver can only react to the loss of synchronization.

3.3 Storage technology

Advances in the CPU and memory designs are quickly outpacing the secondary storage technology. While CPU speeds are increasing dramatically, the speeds of disk drives are barely improving. For I/O bound applications, any improvements in the CPU will be canceled by the limited disk bandwidth. Disk interleaving and novel file structuring techniques have been suggested as a means for increasing bandwidth to disks.

Disk interleaving, also known as disk striping, refers to the technique of spreading a block of data across multiple disk devices. By interleaving data on multiple disks, the data may be accessed in parallel, reducing data transfer time by a factor of 1/n, where n is the number of disks. Multiple disks may be configured in a variety of ways to support the striping of data blocks. Effects of disk configurations and the granularity of striping on the performance of the system are studied in reference [REDY89]. Additional performance studies and statistical modeling of striping may be found in references [KIM86], [KIM87], [SALEM84]. We will discuss disk striping in great detail in later chapters.
The recently proposed concept of RAID [PAT88] enhances the technique of disk striping by providing fault tolerance at a moderate cost of redundant storage. Fault tolerance is achieved by the use of error correcting codes in a way that is similar to what is done for interleaved memory modules.

Ousterhout proposes log-structured files as a means for reducing disk seek time [OUST88]. The log-structuring technique converts all write requests into sequential disk writes, while hoping that most read requests will be satisfied from the cache. This technique exploits locality by caching commonly referenced files in main memory.

Muller and Pasquale propose a novel design for a multi-structured file system in reference [MUL91]. The design combines log-structuring and disk interleaving techniques to provide high bandwidth I/O and fast response. The storage hierarchy is divided into three levels, each implemented on an independent disk array and dedicated to serving file requests of particular attributes. High performance is achieved by optimizing each level for the corresponding set of file access characteristics.

**Discussion**

Three techniques for improving the file system performance may be discerned:

1. concurrent use of multiple storage devices, as in striping or RAID
2. exploiting large caches, as in log-structured files
3. adjusting the file system for a specific workload

It is likely that future file systems will rely on a combination of these techniques to provide high performance. We note that the log-structuring approach will be inappropriate for CMFS because it relies on caching entire files in main memory; this is infeasible for large continuous media files.

### 3.4 File Systems for Digital Audio and Motion Video

The majority of multimedia file systems that have been built or proposed focus on still images and/or audio [THO85],[GIB87],[CHRI86],[SUN89]. Most of these systems may be broadly classified as multimedia storage repositories designed for storage, retrieval and exchange of documents containing images. Such filing systems do not need to operate under stringent time constraints. We focus our attention on systems that provide storage for multimedia objects of temporal nature, such as audio and motion video.

Terry and Swinehart introduced a powerful voice storage system as part of the Etherphone project [TERR88]. The goal of the project is to provide facilities for storage, real-time recording and playback, and management of voice in a distributed computing environment.

Facilities for storage and access to voice are provided by a voice file server. The voice file server meets the real-time requirements of uncompressed voice streams and provides low-level access to disks for storing and retrieving digitized voice files.

Facilities for managing voice are provided by a voice manager. The voice manager implements high level abstractions for referencing and operating on voice segments.
Segments are represented as immutable voice ropes, an abstraction that allows string-like manipulation of sequences of stored voice samples. Operations that may be performed on voice ropes are PLAY, RECORD, STOP, CONCATENATE, etc. On record or playback, digitized voice samples are transferred between the voice file server and an Etherphone device using rate-based flow control. The transfer is initiated by the voice manager at the application's request. Since the Etherphone devices are decoupled from client applications, clients do not have the ability to control the timing and synchronization of audio streams.

The VOX audio server, described by Arons, et al., is a more recent example of a storage system for digitized voice [ARO88]. The VOX server allows more flexibility in handling voice streams than does the Etherphone system by introducing the logical audio device (LAUD) abstraction. A LAUD is a device independent abstraction of an audio related service, such as RECORD or PLAY. LAUDs are associated with physical audio devices. LAUDs may be combined to form composite LAUDs (CLAUDs). CLAUDs are used to synchronize activities of component LAUDs and provide a higher level of service. An example of a CLAUD is an answering machine service. The answering machine CLAUD may contain a PLAY LAUD for playing a prerecorded message and a RECORD LAUD for recording a caller's message. Input from component LAUDs are multiplexed into a single time-stamped stream. Unlike the Etherphone system which uses rate-based flow control, the VOX server allows prefetching and buffering of data by the LAUDs.

Vin and Rangan describe a design of a video file server inspired by the Etherphone project [VIN91]. The proposed video file server extends the concepts introduced by the Etherphone project and addresses the temporal requirements of multimedia objects for continuous access to disks and synchronization of related streams. An abstraction of a multimedia rope is adapted from Etherphone's voice ropes. A multimedia rope may contain media strands, such as strands of audio and video. Access to disks is regulated by admission control algorithms that ensure continuous storage and retrieval of data. Disk layout policies are determined by the continuity requirements of requests. Media strands of a multimedia rope are tied together by synchronization information. Admission control and disk layout policies guarantee continuous playback of multimedia ropes.

In a very recently published paper, Cabrera and Long describe a storage architecture, called Swift, that is based on the concept of distributed disk striping [CAB91]. The stated goal of Swift is to support continuous multimedia in a general purpose distributed system. Swift provides high data-rates by striping a file across a distributed collection of Storage Agents. Clients access the Storage Agents according to a transfer plan created by a centralized Storage Mediator. The Storage Mediator performs resource management and access control.

Although developed independently, the designs of Swift and CMFS are very similar in their use of distributed storage. However, the scope of the design and the implemented prototypes of the two systems differ significantly. The initial design of Swift and its prototype implementation focus mainly on the viability of distributed disk striping as a high-bandwidth storage architecture. High-bandwidth is only one of the goals addressed by the design of CMFS.
Discussion

Previous experience with voice storage systems, particularly the Etherphone project, provided the ability to manipulate voice with ease and flexibility commonly associated with editing text. A similar functionality may be expected of a video storage system.

Initial work in file systems for continuous media identified the need for admission control policies.

The Swift project is a significant proof-of-concept for the viability of distributed striping as a high-bandwidth, scalable storage architecture.

3.5 Contributions to the CMFS design

The design of CMFS inherits and builds upon a number of techniques proposed by previous research work. In order to provide high bandwidth, CMFS incorporates the concept of disk striping and extends it to include node striping. Synchronization of multimedia streams is based on inclusion of time stamps within application frames. On-disk synchronization of related continuous media files is accomplished using a file container storage abstraction. In some respect, a container resembles an on-disk log-structured file. Access to CMFS is governed by admission control policies that are based on the efficient deadline driven scheduling algorithm.
4 CMFS Model

This chapter presents a high-level description of CMFS functional and design models. Specific details and the rationale behind some of the design choices are addressed in later chapters.

4.1 Functional Model

The CMFS functional model is described in terms of the following layers: interface, filing objects, file service, logical storage, and physical storage. The model is depicted in Figure 4-1.

![CMFS Functional Model Diagram]

Figure 4-1  CMFS Functional Model

Interface Layer

The interface layer provides access to the operations of the CMFS filing objects. Conceptually, the CMFS interface layers is similar to the standard I/O library provided by the Unix environment for accessing files.

Filing Objects Layer

The filing objects layer defines storage abstractions for the objects of the file system. The filing objects supported by CMFS are logical frames, files, file containers, and user directories.

A logical frame is a unit of information meaningful to the application.
A file is a collection of application related logical frames. In contrast to the byte stream access semantics of Unix files, the granularity of access to a CMFS file is a logical frame. A CMFS file may be comprised of variable length logical frames; the length of each frame is established by the frame's creator. Each frame is tagged with a unique logical frame number (LFN). Logical frame numbers are consecutive integers that correspond to the logical order of frames within a file.

A container is a typed storage repository that holds a group of application related files. The container's type determines the way in which files are stored on the physical media. A container's type is specified dynamically by the container's creator as a list of attributes. The attributes characterize a container in terms of its physical size, storage contiguity, extensibility, and performance sensitivity.

The file system uses containers for efficient disk space allocation and for physical clustering of files related by performance requirements. A container typed as performance sensitive is capable of satisfying the combined performance requirements of all the component file. This allows an application program to reserve bandwidth for future playback of related files.

Although not used by CMFS, the abstraction of a typed container may be used to provide different classes of file services based on the container's type.

CMFS user directories are similar to the user directories provided by the Unix File System [LEF89].

File Service Layer

The file service layer provides operations for manipulating and accessing the CMFS filing objects. In addition to the basic file system operations such as opening, closing, reading, writing and deleting files, CMFS provides the following services:

- **Synchronization.** This service facilitates stream synchronization by including time stamps in every logical frame.

- **Guaranteed Performance.** This service provides static performance guarantees to all files stored in a container that is typed as performance sensitive and dynamic performance guarantees to all active clients of the system.

- **File Composition.** This service provides the ability to compose new files from pieces of several existing files using a logical frame as the granularity of composition.

- **Broadcast.** This service allows multiple clients of the file system to read the same file simultaneously. All clients reading a file in broadcast mode view the same data at approximately the same time.

- **Directional file access.** This service allows a client to access a file sequentially in forward or reverse direction.

- **Skip-frame file access.** This service allows a client to access a file sequentially while skipping periodically over a specified number of frames.
Logical and Physical Storage Layers

The logical storage layer combines the available storage devices, possibly multiple disks and processing nodes, into a single logical storage entity. The physical storage layer organizes the filing objects on physical storage devices.

4.2 Design Model

The design of CMFS implements the described functional model in three components that may be distributed over a communication network: Logical Storage System, Administration Server, and CMFS Interface. The components are shown in Figure 4-2.

![CMFS Design Model Diagram](image-url)

Figure 4-2 CMFS Design Model
Logical Storage System

The Logical Storage System is a distributed collection of autonomous components called Storage Servers. The Storage Servers are distributed among multiple nodes. The model permits mapping of multiple Storage Servers onto a single node, as illustrated in Figure 4-2.

The collection of Storage Servers is viewed as a single logical entity that provides file services and access to storage devices. Application files are interleaved among the Storage Servers. Each Storage Server may further interleave its portion of the file among several disks.

Administration Server

The Administration Server provides directory services and regulates access to the Logical Storage System. Directory services implement the user directory storage abstraction and provide operations for accessing and manipulating it. The semantics of user directories and their operations are similar to those provided by the Unix File System.

Regulating access to the Logical Storage System ensures that performance requirements of all CMFS clients can be met by the Storage Servers. We refer to this process as admission control.

CMFS Interface

The CMFS Interface provides routines for accessing the CMFS filing objects and services.

The CMFS Interface implements node striping by spreading each application frame across all available Storage Servers. The node striping is invisible to application programs. Applications view the file system as a single logical entity.

Directory services and admission control are also performed transparently by the CMFS Interface.

Protocols

The components of CMFS communicate by exchanging messages across virtual sessions. A virtual session exists between the CMFS Interface and each Storage Server for every open file. Datagram streams are used to communicate with the Administration Server. The Administration Server participates only in opening and closing of a file, whereas, requests for reading and writing data are exchanged strictly between the CMFS Interface and Storage Servers.

Transmission of frames between the CMFS Interface and each Storage Server is client paced; that is, the CMFS Interface running on the client node dictates the rate at which frames are transmitted for storage or retrieval. Transmission of frames for retrieval is
paced by read requests. Transmission of frames for storage is paced by write requests. The write requests may be transmitted isochronously at the sampling rate of the device that generates the frames.
5 CMFS Interface

The CMFS Interface is a library of routines that support transparent access to the file services of CMFS. An instance of the CMFS Interface exists for every open file. Each instance represents a logical association between a client and an open file. The clients of the file system are application programs or threads within programs. State variables are maintained for each instance of the CMFS Interface. The state of an open file consists of virtual sessions to Storage Servers, the client's position within the file, and the current direction of file access. The state variables are volatile and may not be recovered after a node crash.

The file server appears to client programs as a mountable file system with a hierarchical directory space. The CMFS Interface is initialized and activated by mounting the file system on the client node. The mount procedure associates a file name prefix with an Administration Server that "administers" all files whose names begin with the specified prefix. The mount procedure sends a protocol message to the Administration Server and receives a reply which contains a list of network addresses of Storage Servers comprising the mounted Logical Storage System.

5.1 Client Interface Routines

In the previous chapter we described the services offered by CMFS. The following CMFS Interface routines provide access to those services:

- `CMallocate_container`
- `CMopen`
- `CMclose`
- `CMread`
- `CMwrite`
- `CMget_fid`
- `CMioctl`
- `CMlseek`
- `CMtseek`

The C declarations of the routines are shown in Table 5-1.
typedef int BOOLEAN;

int CMAllocate_container(fd, size, contig, stretch, perf_sensitive)
    int fd, size;
    BOOLEAN contig, stretch, perf_sensitive;

int CMopen(path, access_mode, cid, frame_size, frame_rate)
    char *path;
    int access_mode, cid, frame_size, frame_rate;

int CMseek(fd, offset, whence)
    int fd, offset, whence;

int CMtseek(fd, offset, whence)
    int fd, offset, whence;

struct CMSTAT {
    int byte_position;
    int time_position;
    int LFN;
    int frame_size;
};

int CMread (fd, buf, nbytes, stat)
    int fd, nbytes;
    char *buf;
    struct CMSTAT *stat;

int CMwrite (fd, buf, nbytes, time_stamp)
    int fd, nbytes, time_stamp;
    char *buf;

int CMclose(fd)
    int fd;

int CMget_fid(path)
    char *path;

int CMioctl (fd, request, argp)
    int fd, request;

Figure 5-1  C Declarations for the CMFS Interface Routines
Each CMFS Interface routine, except the `CMallocate_container` and `CMget_fid`, has a functional counterpart in the Unix File System. The CMFS routines are syntactically distinguished from their Unix counterparts by the prefix "CM".

The following description of each routine focuses on the features and functionality that differentiate it from the equivalent operation on an ordinary Unix file.

**CMallocate_container**

Clients invoke the `CMallocate_container` routine to create a file container with specific attributes. Parameters supplied to the routine specify the desired container attributes such as size, contiguity, extensibility, and performance sensitivity. The routine returns a unique container identification.

Containers are used by applications programs to group files that are related by performance requirements such as video and associated audio. Containers, in effect, provide a mechanism for reserving disk bandwidth. CMFS ensures that a playback of all files stored in a container does not exceed the disk transfer rate capacity.

**CMopen**

Applications invoke the CMopen routine to access an existing file or to create a new one and associate it with a file container. The CMopen routine initiates an instance of the CMFS Interface associated with the current request.

An existing file may be accessed in one of the following modes: read, update, broadcast, or composition. The `access_mode` parameter of the routine selects the desired mode.

Associated with every open file is a variable called the current frame pointer. The current frame pointer contains a logical frame number of the frame that will be returned for the next read request or updated by the next write request. The `CMopen` routine initializes the current frame pointer according to the requested access mode: for read access the pointer locates the first frame of the file, for update access the pointer locates the end of the file.

For file creations, the client application needs to specify a container for holding the file. The container is specified in the `CID` argument of the routine.

The client application specifies its performance requirements for the request in two parameters: frame size and frame rate. The use of these parameters by CMFS is described in detail in chapter 5.

The `CMopen` routine sends a protocol message to the Administration Server and receives a reply that contains an internal file identifier corresponding to the file name and an indication as to whether the request was admitted or rejected. If the request was admitted, the `CMopen` routine establishes a virtual session with each Storage Server of the Logical Storage System. The same file identifier is used to identify the file to all Storage Servers.

- 30 -
**CMclose**

Clients invoke the **CMclose** routine to indicate the completion of a request. The **CMclose** routine sends a protocol message to all Storage Servers and to the Administration Server causing them to release all resources associated with the request. Virtual sessions and state variables of the corresponding instance of the CMFS Interface are destroyed.

**CMlseek**

Clients invoke the **CMlseek** routine to change a logical frame position within a file. Logical frame positions are expressed in terms of logical frame numbers. The **CMlseek** routine modifies the current frame pointer associated with the file to reflect the new logical position. The current frame pointer is used by the **CMread** and **CMwrite** routines to identify the next frame for retrieval or update.

**CMtseek**

Clients invoke the **CMtseek** routine to change a temporal position within a file. Temporal positions are expressed in seconds. Positioning a file on the n-th second is equivalent to establishing a logical frame position using the logical frame number of the first frame recorded in the n-th second. The association between logical frame numbers and recording times is established by the **CMwrite** routine and is maintained by Storage Servers.

**CMioctl**

Clients invoke the **CMioctl** routine to perform the following functions:

- set the frame skip parameter
- change the direction of file access

The **CMioctl** routine modifies the state variables associated with the file to reflect the specified skip parameter and the established direction of file access. These parameters remain in effect until the next invocation of the **CMioctl** routine sets the new values.

**CMget_fid**

Clients invoke the **CMget_fid** routine to convert a file name into an internal file identifier (FID). The FIDs are used in composition records to reference component files. Composition records are described in the context of the **CMwrite** routine.

The **CMget_fid** routine transmits a protocol message to the Administration Server and receives a reply containing the file’s FID. The FID is returned to the calling client.
**CMwrite**

Clients invoke the *CMwrite* routine to store a logical frame in a file or to supply a sequence of frames for file composition. The file access mode specified by the *CMopen* routine determines whether the application intends to update an existing file or compose a new one.

In the case of updating a file, the *CMwrite* routine implements node striping by uniformly spreading the frame across all Storage Servers. If $n$ Storage Servers are available as part of the Logical Storage System, the *CMwrite* routines splits the frame into $n$ portions of approximately equal lengths and transmits each portion to a different Storage Server. The portions of the frame are stored concurrently by the corresponding Storage Servers.

In the case of composing a file from existing files, the data buffer supplied by the client contains a stylized composition record. The composition record contains a list of descriptors. Each descriptor identifies a component file, a starting location within the file, and the length of the sequence. The location and length of a sequence may be specified using logical frame numbers or temporal attributes.

The *CMwrite* routine transmits a copy of the composition record to each Storage Server.

**CMread**

Clients invoke the *CMread* routine to retrieve a frame from an open file. Each instance of the CMFS Interface performs frame read-ahead. The purpose of the read-ahead is to smooth out the jitter of frame interarrival periods. The jitter may be introduced by the network or by variations in disk access times on Storage Servers. Frame read-ahead, in effect, averages delay variations between successive frames.

If a complete frame already exists in the read-ahead buffer, the *CMread* routine copies the frame to the application supplied buffer and initiates a read-ahead of the next frame. If the frame has not been received yet, the *CMread* routine waits for the previous read-ahead to complete before sending the next read requests. A read request contains the following information relevant for frame retrievals:

- current frame request
- prefetch hint for the next frame

The current frame request identifies the frame requested by the application. The prefetch hint identifies a frame that the application is most likely to request next. Both the current frame request and the prefetch hint are computed based on the current frame pointer, the frame skip parameter, and the established direction of file access. An algorithm for this computation is described symbolically as follows:
current_request = current_frame_pointer ± frame_skip.
current_frame_pointer = current_request.
prefetch_hint = current_frame_pointer ± frame_skip.

For the forward direction the sign of the frame_skip term is positive, for the backward direction it is negative.

The current frame request and the prefetch hint may be expressed in any combination of logical frame numbers and temporal positions. Each Storage Server responds to the read request by transmitting its portion of the frame identified by the current frame request and locally prefetching the frame identified by the prefetch hint. The READ routine receives a portion of the requested frame from the corresponding Storage Server and assembles a complete frame in its read-ahead buffer. The prefetched frame is not transmitted by the Storage Servers until it is explicitly requested.

The READ routine returns a status block that contains the following information:

- logical frame number of the current frame
- frame size
- temporal offset of the current frame

5.2 Discussion of the CMFS Services and Interface

The CMFS filing objects and services are designed to support applications that handle real-time continuous media. This section briefly discusses the utility and possible usage of these objects and services in a multimedia application.

Variable Length Logical Frames

Variable length logical frames are primarily intended for video based applications that employ variable bit-rate image compression such as JPEG or MPEG. A JPEG application handling still images may choose to map a single compressed video frame onto a single logical frame. In contrast to that, an MPEG application handling motion video may choose to map a sequence of compressed images enclosed between key MPEG frames onto a single logical frame.

Directional and Frame-Skip Access

Directional and frame-skip access services are intended to facilitate the development of applications that provide motion video playback features. The directional access may be used to implement forward and reverse playback. The frame-skip access, in combination with the directional access, may be used to implement the "fast forward" or "fast re-wind" feature of a motion video playback.
The ability to skip frames may also be utilized in providing a scalable access to a video file. For example, a video file could be stored in such a way that a full resolution video is delivered by reading every frame, half the resolution is delivered by reading every other frame, and successively lower resolutions are delivered by reading every \( n \)th frame.

**Temporal Positioning**

The temporal positioning enabled by the \textit{CMseek} routine and the time stamps embedded in each frame are intended primarily to facilitate synchronization of related streams. For example, an application may use time stamps returned by the \textit{CMread} to maintain relative synchronization of audio and video streams. Relative synchronization may be achieved by "slaving" the video stream to the audio stream by either dropping or repeating video frames depending on the progress of the audio playback. The \textit{CMseek} call may be used to align the video and audio streams, if a resynchronization is required.

**File Composition**

The file composition facility has obvious applications in video editing for pasting a sequence of separately recorded scenes. The composition facility may also be used for editing immutable files. For example, instead of updating a sequence of frames in place, permanently altering the original file, an application may choose to create an "update" file containing a sequence of frames that serve as updates, and then compose a file using frame sequences from the original and update files. The composed file in this case is identical to the one that would result from updating the original file in place.

**File Containers**

Applications may use a container to hold files that are intended for simultaneous playback such as video and associated audio, for example. The performance sensitivity attribute of a container allows an application to record related files separately, while ensuring in advance the performance guarantees necessary for the simultaneous playback.

**Broadcast**

The primary purpose of the CMFS broadcast facility is to reduce disk traffic in a situation where multiple clients are reading the same file simultaneously. The reduction of disk traffic is the result of partial file caching and locality of access in the case of simultaneous readers. Some applications may exploit the broadcast mode in order to benefit the overall file system performance. As an example, consider a number of distributed workstations, each running the same, display intensive, application. In this environment each instance of the application can access files in broadcast mode.

A benevolent side effect of the broadcast facility is the ability to synchronize multiple clients to a common service. For example, geographically distributed participants of a
live conference or listeners of a lecture may need to view a file simultaneously, in order to stay in phase with the speaker, whose commentary is synchronized with the playback of the file.

5.3 Protocol Design Alternatives

This section reviews possible alternatives to the design of the protocol for frame transmission and discusses their advantages and disadvantages in comparison with the CMFS approach.

Transmission for Retrieval

In the CMFS design, the transmission of frames from the Storage Servers is paced by the client; that is, a frame is transmitted only in response to a read request. An alternative approach is to allow the Storage Servers to provide pacing for the transmission of frames automatically, based on the client requested frame rate. The fundamental difference between the client paced and the server paced transmission lies in the source of timing necessary for stream synchronization. In the client paced transmission the timing is derived from the prerecorded time stamps in each frame. In the server paced transmission the timing is derived implicitly from the instances of frame arrivals.

The client paced transmission offers several advantages over the server paced transmission method.

First, it allows greater flexibility and precision necessary for synchronization. Both the flexibility and precision result from the client’s ability to prefetch data on related streams early. Early prefetching of data helps in dejittering the streams and allows longer time intervals for equalizing the delays. Prefetching of data is appropriate for synchronizing continuous streams as well as discrete events, such as different portions of a slide in a multimedia presentation.

Second, the client paced transmission allows multiple Storage Servers to operate autonomously; the server paced transmission would require a potentially complicated inter-server synchronization scheme to ensure that Storage Servers remain in phase while transmitting their corresponding portions of each frame.

Finally, the client paced transmission enables a simpler implementation of the Storage Servers. The server paced transmission method would require an operating system with a real-time flavor and a network that reserves bandwidth deterministically based on the peak performance requirements.

The client paced transmission has several disadvantages, all associated with the overhead caused by the transmission of the read requests.

First, each read request consumes some amount of network bandwidth. Second, in the case of a single frame read-ahead, the frame inter-arrival time is increased by the network round-trip delay. Third, the operating system incurs some overhead caused by the work performed to send the read requests.
All these disadvantages, however, are minor. The amount of network bandwidth consumed by a single read request is insignificant compared to the bandwidth needed for a single frame. The round-trip delay, which occurs only if a single frame read-ahead is used, is also insignificant compared to the frame transmission time. For example, the round-trip delay on a 20 km FDDI local-area network is about .2 ms., which is only .7% of the 26 ms that is required to transmit one monochrome NTSC frame. Finally, by the same logic, the software overhead of preparing and sending a short read request is a small fraction of the time and resources required to process a single received frame.

The advantages of the increased flexibility, precision, and Storage Server autonomy that are offered by the client paced transmission, far outweigh the small cost of transmitting the read requests.

Transmission for Storage

The CMFS design specifies the client paced transmission of frames from the client to Storage Servers. An alternative approach is to allow the Storage Servers to pace the clients. For example, the Storage Servers could pace the clients using frame acknowledgments.

In the previous case of transmission for display we argued in favor of the client paced transmission based on its flexibility and precision necessary for stream synchronization. In the case of transmitting frames for storage the issue of synchronization is irrelevant; the Storage Servers do not need to synchronize the received data streams.

The fundamental difference between the client paced and the server paced transmission, in the case of transmission for storage, is the location of temporary buffering. Presumably, the benefit of the server paced transmission is that the server will ask for a frame only when it has sufficient resources to process it; until that time the frame is buffered by the client. In the client paced transmission the server may need to buffer the frame, assuming that buffer space is available, until it has sufficient resources to process it.

In the presence of performance guarantees the client paced transmission is preferable, since the server is guaranteed by the admission control to have sufficient resources for all requests. In a statistically multiplexed environment the server paced transmission involving frame acknowledgments may be required in order to cope with temporary overload conditions on the server.

Multiplexed Streams

The CMFS design treats every data stream independently by mapping each stream onto a separate virtual session, regardless of the temporal relationships among the streams. An alternative approach is to multiplex related streams onto the same session and possibly interleave the multiplexed data on a storage device.

The main advantage of the multiplexing method is that it allows more precise synchronization of related streams than does the original approach. A network packet multiplexes the data samples from related stream. Since the packet travels in the network as a unit,
all related samples experience the same delay and arrive at the destination at the same
time; hence a greater synchronization precision can be achieved than in the case of in-
dependent sessions with possibly different sample arrival times.

However, the multiplexing method has several major disadvantages. First, it requires
that related streams originate from a single source and terminate in a single destination.
Second, even though related streams such as audio and video have different characteris-
tics, they must be transmitted over a single network connection, precluding the possibil-
ity of selecting the most appropriate connection for each stream [NIC90]. Finally, since
the multiplexed session combines samples generated at possibly disparate sampling
rates, the task of dejittering individual streams may be difficult [NIC90].
6 Administration Server

The Administration Server is a process that "administers" a Logical Storage System. The administrative functions are directory services and admission control. For simplicity, our design assumes a one to one correspondence between the Administration Server and the Logical Storage System. The design may be easily extended to allow an Administration Server to administer multiple Logical Storage Systems. In this case, the administrative functions may also be extended to include load sharing of file creations among all the available Logical Storage Systems.

Initialization

Upon activation, the Administration Server is supplied with initialization parameters. The initialization parameters include a list of network addresses of Storage Servers comprising the Logical Storage System. The list of addresses is transmitted to the CMFS Interface in response to a mount request.

The Administration Server sends a protocol message to each Storage Server requesting its operational state. The operational state of a Storage Server is described by static and dynamic parameters. The static state describes the Storage Server's capacity in terms of:

- the maximum number of open requests that may be supported simultaneously
- maximum storage bandwidth

The dynamic state consists of the Storage Server's current load and is described by a list of currently active files with associated performance requirements.

The operational state of each Storage Server is used in admission control.

6.1 Directory Services

Files are organized in a hierarchical directory space in the same way as in traditional Unix-based file systems. Applications reference files by names, which are arbitrary ascii strings. Storage Servers reference files by internal file identifiers (FIDs). Directory services provide a mapping between the file names and the FIDs. An FID is assigned to a new file at the request of the CMopen routine. Directory services maintain the correspondence between a file name and the assigned FID as long as the file exists. The FID may be reused after the file has been deleted.

Directory services provide file access control, based on ownership and protection, in a similar way that is done in the Unix file system.

6.2 Admission Control

Admission control is a process that regulates access to the Logical Storage System according to the performance requirements specified by file open requests. The purpose of
admission control is to ensure that the Storage Servers of the Logical Storage System are capable of providing guaranteed service to all the admitted requests.

Admission control is performed dynamically and statically. Dynamic admission control ensures guaranteed service to all files that are currently open. Static admission control ensures in advance the feasibility of guaranteed service to a set of requests.

Dynamic admission control is performed for all open requests. A request passes the dynamic admission test if its performance requirements can be satisfied without violating any of the performance guarantees given to the previously admitted active requests.

Static admission control is performed only for those open requests that are creating a new file in a container typed as performance sensitive. A request passes the static admission test, if it is feasible to provide performance guarantees necessary for a simultaneous playback of all the files stored in the associated container, including the file that is created by the new request. The feasibility of performance guarantees for future service is computed on the assumption that the playback requests for the files in the container are the only active requests in the system. Furthermore, it is assumed that the performance requirements used in playback are the same as the ones used in the original recording of the files.

A new request is accepted only if it passes both the dynamic and static admission tests. Both tests employ the same algorithm for deciding admission. The admission algorithm is a function that operates on a set of performance requirements and, based on certain scheduling and processing assumptions, computes the feasibility of the guaranteed service. Dynamic tests invoke the algorithm with the set of performance requirements of all active requests, including the new request. Static tests invoke the algorithm with the set of recording performance requirements for all files stored in the container, including the new request as well.

Selecting an Algorithm for Scheduling and Admission Control

Any algorithm that determines admission of requests for guaranteed service is fundamentally related to the scheduling policies used in servicing the requests. The relationship between the two is formulated as follows. A new request may be admitted into the system if a schedule exists such that the new request and all the previously admitted ones can be serviced within their specified time constraints. A schedule with the above properties is called a feasible schedule. The problem of computing a feasible schedule for time-constrained requests has been studied extensively in the context of hard real-time systems.

A number of scheduling algorithms for hard real-time systems have been proposed in the literature [CHENG87]. The algorithms are distinguished by the model of the system, the nature of tasks to be scheduled, and the rules used in computing a feasible schedule.

The system model defines a target environment for executing the schedule. We will consider only those algorithms that are designed for uniprocessors and allow preemption. The reason for requiring preemption is that many non-preemptive scheduling problems have been shown to be NP-hard [CHENG87]. Specifically, the problem of scheduling
requests with arbitrary arrival times non-preemptively, which is germane to the CMFS environment, is NP-hard.

A task is a schedulable unit in the system. For the purposes of scheduling file access requests, it is convenient to view a task as a function that is invoked by a request to perform a certain service. Tasks may be characterized in terms of their periodicity, time constraints, and precedence constraints. A periodic task is invoked once per time period. A task that receives requests at a constant rate is periodic. A time constrained task is a task that must finish executing a request before a specified deadline. Precedence constraints refer to the partial order in which all tasks must be executed. We are interested in an algorithm for periodic time-constrained tasks. Such tasks are ideally suited for the isochronous nature of real-time continuous media.

Real-time scheduling algorithms typically assign execution priorities to tasks to reflect their importance and time criticality. Two priority assignment schemes are possible: fixed and dynamic. In a fixed priority assignment scheme, priorities assigned to tasks do not change with time. In a dynamic priority scheme, the priority assigned to tasks may vary.

For the purpose of scheduling file access requests we propose to use the preemptive, deadline driven scheduling algorithm of Liu and Layland, developed specifically for periodic time-constrained tasks [LIU73]. The deadline driven algorithm assigns tasks priorities dynamically: the highest priority is assigned to the task with the nearest deadline, and the lowest priority is assigned to the task with the furthest deadline. At any instant, the task with the highest priority and yet unfulfilled request is scheduled for execution, possibly preempting a currently running task with a lower priority.

The deadline driven algorithm is chosen on the basis of its optimality, efficient use of the processing resources, and its particular suitability to the periodic time-constrained nature of the CMFS read and write requests. The deadline driven algorithm is optimum in the sense that if a set of periodic tasks can be scheduled by any algorithm, it can also be scheduled by the deadline driven algorithm. Liu and Layland show that the deadline driven algorithm achieves the processor utilization factor of 100%. The utilization factor is defined as the fraction of processor time spent in the execution of the task set.

The remainder of this chapter describes the use of the deadline driven algorithm in scheduling and admission control of the CMFS file access requests.

**Deadline Driven Scheduling**

For the sake of clarity, we assume that the Logical Storage System consists of only one Storage Server with one disk. The admission control algorithm generalizes easily for the case of multiple Storage Servers with multiple disks. The scheduling algorithm is independent of the number of Storage Servers, since each Storage Server executes the same algorithm.

A CMFS client specifies its performance requirements in two parameters supplied to the *CMopen* routine:
• $S$, frame size
• $f$, frame rate

If the client is admitted into the system, it will periodically request a service from the Storage Server; the request period of the client is $1/f$.

For every request from the client the Storage Server accesses the disk to store or retrieve a frame of size $S$. Frames are stored on the disk contiguously. The Storage Server transfers every frame in blocks of size $B$, where $B < S$ in most cases.

The frame transfer initiated by the current request must be completed before the arrival of the next request from the same client. Since the client’s request period is $1/f$, the next request will arrive at time $D$: $D = t + 1/f$, where $t$ is the time of arrival of the current request. $D$ is known as the deadline of the request; it is the absolute time by which the request must be completed.

The deadline of a request determines its scheduling priority: the highest priority is assigned to the request with the nearest deadline. Scheduling a request with the highest priority involves preemption of the frame transfer initiated earlier by a lower priority request. The preemption occurs on the block transfer boundaries. The Storage Server waits for the current block transfer to complete and then initiates the frame transfer for the new request. Thus the maximum cost of preempting a lower priority request is the time required to complete a disk block transfer: $B$, where $B$ is the block size and $R$ is the disk transfer rate. The maximum cost of resuming the preempted frame transfer is the maximum disk access time$^1$.

**Deadline Driven Admission Control**

An important result derived by Liu and Layland is the schedulability criteria for a set of periodic time-constrained tasks by the deadline driven algorithm. Liu and Layland proved that a set of $n$ periodic tasks is schedulable by the deadline driven algorithm if and only if the following condition holds: where $C_i$ is the worst-case execution time of a request for the $i$th task and $f_i$ is the request rate for the $i$th task. The above schedulability criteria is the basis for the admission control algorithm. A new CMFS client is admitted into the system if and only if the set consisting of all previously admitted requests and the new one satisfies (1).

\[
\left( \sum_{i=1}^{n} C_i \cdot f_i \right) \leq 1 \tag{1}
\]

$^1$ disk access time is composed of seek and rotational latency.
In order to complete the description of the admission control algorithm, we have to show how to compute $C_i$, the worst-case execution time of a client’s request. The time required to execute a single request for the $i$th client is:

$$C_i = t_{access} + \frac{S_i}{R} + P_i$$  \hspace{1cm} (2)

where

- $t_{access}$ is the disk access time
- $S_i$ is the frame size specified by the $i$th client
- $P_i$ is the execution delay caused by preemptions
- $R$ is the disk transfer rate

$P_i$ includes a one time cost of preempting a lower priority request and multiple costs associated with resuming execution of the request each time it is preempted by a higher priority request.

We mentioned earlier that the maximum cost of preemting a request is $B$, the time required to complete a disk block transfer, and that the maximum cost of resuming execution is the maximum disk access time. Thus, where $m_i$ is the maximum number of re-

$$P_i = \frac{B}{R} + (m_i \cdot t_{\text{max}, access})$$  \hspace{1cm} (3)

sumptions during the execution of a request from the $i$th client.

In order to find $m_i$, consider the set of $n$ clients with the following frame rates arranged in ascending order: $f_1, f_2, \cdots, f_n$, where $f_i$ is the frame rate specified by the $i$th client, and $f_i \leq f_{i+1}$. Note that the clients are enumerated in the order of increasing frame rates, rather than the order of chronological arrival times. Such enumeration scheme does not result in any loss of generality.

Suppose the Storage Server receives a request from client $j$, while currently executing a request from client $i$. The currently executing request will not be preempted by the new request if $j < i$, since the deadline of the $i$th request is closer. The currently executing request may, however, be preempted by the new request if $i < j$. In the worst case, a request from the $i$th client will be interrupted $\frac{f_{i+1}}{f_i}$ times by client $i+1$, $\frac{f_{i+2}}{f_i}$ times by
client \(i+2\), and so on until the \(n\)th client\(^1\). Thus the maximum number of interruptions and consequent resumptions of a request from the \(i\)th client is:

\[
m_i = \sum_{j=i}^{n-1} \lceil \frac{f_{j+1}}{f_i} \rceil
\]  

(4)

Back substituting (4) into (3) and (3) into (2) provides a way of computing the worst-case execution time, enabling the use of (1) as the admission criteria.

**Best Effort Service**

The scheduling and admission control algorithms are intended for clients that require guaranteed service for access to continuous media. In practice, clients that perform editing operations or access conventional files do not need strict performance guarantees and would be willing to accept performance based on the "best effort" from the file system. The algorithms could be easily extended to accommodate clients whose performance requirements are specified as best effort. The admission control process could permanently "reserve" a slot for a client with a low frame rate. The time that would normally belong to the reserved client may be spent on multiplexing among all the "best effort" requests in the system.

\(^1\) \(\lceil x \rceil\) denotes the largest integer smaller than or equal to \(x\).
7 Storage Server

In the description of the CMFS design model, presented in Chapter 4, we introduced the notion of a Logical Storage System as a distributed collection of Storage Servers. Storage Servers comprise the major component of CMFS. The design of Storage Servers as autonomous components of a single Logical Storage System is the topic of this chapter.

7.1 Storage Servers as a Logical Storage System

Storage Servers provide low-level access to storage devices. The design of CMFS assumes that storage devices are magnetic disks. Each Storage Server operates within an independent environment called a storage domain. A storage domain comprises a single instance of a Storage Server, a processing node, and a set of disks called the stripe set. The stripe set contained in the storage domain is accessible only by the instance of the Storage Server that operates within that domain. Multiple storage domains may exist on the same processing node. The notion of storage domains is illustrated in Figure 7-1.

![Diagram of Storage Domains](image)

**Figure 7-1 Storage Domains**

Conventional file servers normally make use of a single storage domain; that is, every file is stored entirely on the set of disks that belongs to the storage domain. A conventional file server may access any part of any file stored within its domain.

Unlike conventional file servers, CMFS makes use of multiple storage domains, each controlled by an autonomous Storage Server. User data stored in a file is interleaved among all storage domains that are defined for the Logical Storage System. Each Storage Server may access only that portion of a file which is stored in its own domain. The
file service abstraction masks the distribution of data across multiple domains and presents a single unified view of CMFS to the clients of the system.

In the previous chapters we referred to the interleaving of user data among multiple storage domains as node striping. Node striping is conceptually similar to disk striping: a block of data is spread across multiple nodes. In Chapter 3 we noted that the benefit of disk striping is a reduction of data transfer time by a factor of \(1/n\), where \(n\) is the number of striped disks. Node striping extends the benefit of disk striping by further increasing the aggregate data throughput to and from the file system.

In addition to the limitation of the disk data transfer rate, the maximum data throughput on a single node is limited by the bandwidths of the network adapter, main memory, and the internal bus. By spreading a block of data across \(n\) nodes, the throughput may be increased by a factor of \(n\). Node striping is a facility that allows scaling of the file server to support higher network speeds.

As we discussed in the previous chapter, the node striping policy is implemented by the \textit{CMwrite} routine of the CMFS Interface. The \textit{CMwrite} routine splits every frame uniformly across all Storage Servers. Each Storage Server stores an approximately equal portion of a frame. A Storage Server spreads its portion of the frame uniformly across all the disks in its storage domain. The uniform node and disk striping policy is illustrated in Figure 7-2.

![Diagram showing node striping](image)

**Figure 7-2  Synchronous Disk and Node Striping**

The reason for choosing this particular form of striping is discussed in detail in later sections.

The algorithms executed by each Storage Server for storage and retrieval of frames is independent of the total number of servers in the Logical Storage System. It is the responsibility of the \textit{CMread} routine of the CMFS Interface to request each portion of the frame from all Storage Servers and to reassemble a complete frame for the client. By treating a portion of a frame as if it were a complete frame, Storage Servers are capable of functioning independently from each other. Independence of Storage Servers implies
that they never need to exchange messages or synchronize their activities; each Storage Server functions in complete autonomy.

Since Storage Servers are autonomous, it is sufficient to describe the functionality and the design of a single Storage Server. The remainder of this chapter describes the design of a Storage Server in terms of the initialization process, data structures, striping policy, request handling, and restart and recovery procedures.

### 7.2 Initialization

Upon activation, a Storage Server is supplied with the name of a parameter file. The file contains the following initialization parameters:

- a set of disks that defines the initialized storage domain
- maximum storage bandwidth
- network address of the node supporting the storage domain
- maximum memory available for buffer pools

After having read the parameter file, the Storage Server allocates and initializes the following data structures:

- file headers
- file maps
- container descriptors
- free list
- context blocks buffer pool.

In order to initialize the allocated data structures with appropriate information, the Storage Server performs a recovery process. The purpose of the recovery process is to restore the state information of the file system as it existed before the storage server became inactive. The data structures and the recovery process are described in detail in later sections.

After having completed the recovery process, the Storage Server enters its main work loop of servicing requests. The Storage Server continues to service requests until it is terminated.

### 7.3 Data Structures

The main data structures used by the Storage Server may be classified into two categories: recoverable and volatile. Recoverable data structures contain the file meta-data which must persist across node failures. Volatile data structures contain contextual and transient data that need not survive the node failures.

The recoverable data structures include:

- file headers
• file maps
• container descriptors
• free list.

The volatile data structures include: context blocks and the buffer pool.

All the recoverable data structures, with the exception of the file headers, describe the location of data or free blocks on disk. The Storage Server views each disk in its storage domain as an array of logical blocks indexed by Logical Block Numbers (LBNs). For the sake of clarifying the purpose of the disk related data structures, we will assume, in this section only, that the storage domain contains a single disk. In the next section we will show that the usage of these data structures remains the same in the presence of multiple disks and uniform disk striping.

File Headers

A File Header is a structure that contains global information pertaining to an existing file. An array of File Headers is allocated by the Storage Server during initialization. A unique file identifier (FID) assigned by the Administration Server is used to index into the File Header array. The File Header contains the following information:

• pointers to file maps
• the time of creation and last access
• the size of the file
• number of frames in the file
• identification of the container holding the file.

Note that the header does not contain the file ownership or protection information. The file access control is performed by the directory services provided by the Administration Server.

Container Descriptors

Containers are portions of the disk reserved for storage of user files. A container is composed of one or more contiguous disk fragments. A container descriptor exists for each contiguous fragment of a container. Associated with each container is a linked list of container descriptors anchored in the Container Id Table. Containers are assigned unique identifiers (CIDs) which serve as an index into the Container Id Table. Each entry of the Container Id Table serves as a container base. A container base records the following information:

• a pointer to the list of container descriptors
• the original size and disk location of the container
• the size and disk location of each extension of the container
Each container descriptor identifies a contiguous fragment of a container in terms of its disk location, size, and the current write pointer. These structures are illustrated in Figure 7-3.

Management of data within a container is described in later sections.

![Container Descriptors Diagram](image)

**Figure 7-3 Container Descriptors**

**File Maps**

File maps are conceptually similar to inodes in the Unix-based file systems. Maps are used to locate file data on the disk. For each user file, the Storage Server maintains three maps: physical, frame, and temporal.

The physical map is an array of data block descriptors. Each descriptor contains the starting location and the size of a contiguous file data block. If the entire file is contiguous, the physical map contains a single block descriptor. The order of block descriptors within the physical map corresponds to the logical order of data blocks within the file. The physical map structure is illustrated in Figure 7-4.
Figure 7-4 Physical Map

The frame map is an array of frame offsets into a file. The array is indexed by a Logical Frame Number. Each entry of the frame map contains an offset of the corresponding frame into the file. The last entry of the frame map contains the size of the entire file. The frame map is used for two purposes:

- determine the size of each frame
- locate an individual frame on disk.

The size of the $i$th frame, $S_i$, is determined as follows: where $O_i, O_{i+1}$ are the offsets of

$$S_i = O_{i+1} - O_i$$

the $i$th and $i+1$ frame, respectively. It is possible to determine the disk location of a frame by using the physical map and the frame’s offset into the file. Figure 7-5 depicts an example illustrating the relationship between the maps and the frame’s location on disk.

Figure 7-5 Frame Map
The temporal map maintains temporal offsets of frames relative to the beginning of a file. The first frame of a file is recorded at time zero. The temporal offset of a frame reflects the time at which the frame was recorded relative to the first frame. The resolution of temporal offsets is one second.

The temporal map is an array of logical frame numbers indexed by the temporal offset. Each entry of the temporal map contains a logical frame number of the first frame recorded within the time of the corresponding temporal offset. The temporal map is used to establish a temporal position within a file. The relationship among the maps is illustrated in Figure 7-6.

It is possible to determine an LFN corresponding to a requested temporal position based on the frame rate used in recording, thus bypassing the need to maintain temporal maps. However, decoupling the frame rate from temporal positioning allows client applications more flexibility in establishing temporal relationships among frames.

![Temporal Map Diagram]

**Figure 7-6 Temporal map**

**Free List**

The free list is a linked list of disk space descriptors. Each descriptor locates a contiguous free area on the disk in terms of the area's size and starting LBN. Disk space is used for allocating file containers. Initially, the free list consists of a single descriptor that specifies the entire disk as free space. The allocation and subsequent deletion of containers may result in the fragmentation of free space. A descriptor is created on the free list for every contiguous fragment comprising the free disk space. The process of deleting a container may cause several disjoint free space areas to form a single contiguous disk block. In this case, the descriptors corresponding to the disjoint areas are coalesced into a single descriptor that specifies the combined area as contiguous free space.


Context Blocks

In order to allow multiple clients to access the same file simultaneously, the Storage Server needs to maintain separate contexts for each client. During the initialization process the Storage Server allocates a pool of free context blocks. An unused context block is allocated from the pool during the file open process. The index of the allocated block serves as a unique context identification, and is used to distinguish multiple instances of access to the same file.

The context of a file access contains the following information:

- the client’s network address
- the FID of the associated file
- pointers to receive and transmit buffers
- current position in the file.

Buffer Pool

The Storage Server statically allocates a portion of main memory as a buffer pool. The size of the pool is determined by the maximum memory parameter specified in the initialization file. The buffer pool is used for allocating receive, transmit, and broadcast buffers.

For each instance of an open file the Storage Server allocates two transmit and two receive buffers. Double-buffering facilitates the pipelining of network transmission and disk access.

A portion of the buffer pool is reserved for allocating broadcast buffers. The broadcast buffers are used to cache frames of a file that is read in broadcast mode. The size of the subpool used for allocating the receive and transmit buffers places a limit on the maximum number of concurrent clients. The size of the broadcast subpool determines the extent to which simultaneous readers of a file may be out of phase with each other in viewing the file data.

7.4 Disk Striping

In the previous section we discussed the disk related data structures on the assumption that a storage domain contained a single disk. In this section we extend the discussion to a more general case in which a storage domain contains multiple disks. The goal of this section is to explore a variety of methods available for storing data on multiple disks and to select the one that is most suitable for CMFS. Note that the following discussion applies directly to node striping as well.
Several alternatives exist for organizing data on multiple disks: traditional load sharing, synchronous striping, and asynchronous striping.

In the traditional approach all disks are organized as independent units. Each file is located entirely on one disk, and files are distributed across the disks according to a load sharing scheme. Load sharing benefits the aggregate data throughput; however, the throughput of a single file remains the same as in the single disk configuration.

Multiple disks may be configured to increase the data throughput to a single file by using a technique known as disk striping. We briefly discussed the concept of striping in chapter 3. Striping refers to the interleaving of data among multiple disks. Interleaved data may be accessed in parallel, thus reducing disk transfer time and increasing the data throughput.

Two alternative methods exist for striping the file data across multiple disks: synchronous and asynchronous. Conceptually, synchronous striping refers to byte-wise interleaving of data among multiple disks. The disks are synchronized such that each disk head is positioned at the same place. All the disks function together as a single large disk.

Synchronous striping is equivalent to splitting a block of data into $n$ equal portions, where $n$ is the number of disks, and writing each portion concurrently to a corresponding disk of the stripe set. The data block, before it is split into equal portions, is known as a stripe. A portion of the stripe written to a single disk is known as a striping unit.

A stripe may consist of one or more frames. Synchronous striping in which a stripe contains one frame is called uniform, since each frame is uniformly distributed across all disks. For convenience, we will refer to synchronous striping in which a stripe contains $n$ frames as $n$-frame striping.

Uniform and $n$-frame striping are illustrated in Figures 7-7 and 7-8 respectively.

![Figure 7-7 Uniform Striping](image-url)
Conceptually, asynchronous striping refers to block-wise interleaving of data among all disks, where block sizes may vary across the disks. In terms of frames, asynchronous striping is equivalent to writing frame $i$ entirely to disk $i \mod n$, where $n$ is the number of disks. Asynchronous disk striping is illustrated in Figure 7-9.
Figure 7-9  Asynchronous Striping

Most of the remaining discussion in this section focuses on selecting a storage method that is most suitable for CMFS.

A Case for Uniform Disk Striping

Performance of the three methods for organizing data on multiple disks depends on the workload characteristics of the file system. Simulations studies have shown that a synchronous system outperforms other disk organizations at low request rates, whereas, load sharing and asynchronous striping systems perform better at high request rates [REDY89]. The simulation studies address conventional workloads that are characterized by bursty access to small blocks, as in programming development and transaction processing environments.

Multimedia Workload

Multimedia applications, however, present a quite different workload to the file system. Based on our discussion in Chapter 2 concerning the characteristics of multimedia objects, it is reasonable to assume that, unlike conventional workloads, a multimedia workload will consist predominantly of continuous, sequential access to large blocks of data. In addition to the workload characteristics, the performance of a storage system de-
pends on the way in which requests are handled by the file server. The performance of conventional file servers is sensitive to the variations in request rates, since the requests are statistically multiplexed. In contrast to conventional file systems, CMFS is largely insensitive to the variations in the request rate, since the process of statistical multiplexing is regulated by the admission control policies.

The best storage method for CMFS is the one that enables the majority of the server’s functional modes to provide the highest data throughput for the multimedia workload. We will show that synchronous uniform striping is the best method in the majority of cases. The optimality of the uniform striping is fundamentally derived from its ability to provide the maximum available concurrency for accessing every frame of every file. The following discussion assumes that a storage domain contains \( n \) disks, and that each Storage Server performs the deadline driven scheduling algorithm described in Chapter 6.

**Load Sharing**

We already mentioned that striping outperforms load sharing in accessing a single file, since in load sharing each file resides entirely on one disk. The load sharing method provides the highest throughput in the situation where \( n \) active requests are accessing \( n \) files residing on \( n \) different disks, since this situation leads to the maximum possible parallelism. In this case, the throughput is equivalent to that which may be obtained by asynchronous striping. In order to obtain the same throughput with asynchronous striping, the Storage Server would have to service \( n \) frames per client request. It is clear then that asynchronous striping outperforms load sharing in situations where clients read entire files sequentially. Therefore, the selection of methods for organizing data on multiple disks reduces to comparing synchronous and asynchronous striping.

**Synchronous versus Asynchronous Striping**

Synchronous \( n \)-frame striping outperforms asynchronous striping. In comparing the two methods we assume that in asynchronous striping the Storage Server would access \( n \) frames in parallel, each frame residing on a different disk; whereas in synchronous \( n \)-frame striping the Storage Server would access \( n \) frames by accessing equal portions of the \( n \)-frame stripe in parallel. Let \( T_s \) denote the time required to read \( n \) successive frames striped synchronously, as in \( n \)-frame striping. Similarly, let \( T_a \) denote the time required to access \( n \) successive frames striped asynchronously.
Then

\[ T_s = t_{avg\text{-}access} + \sum_{i=1}^{n} \frac{S_i}{n \cdot D} \]  

(1)

where

- \( t_{avg\text{-}access} \) is the average disk access time
- \( S \) is the frame size
- \( D \) is the disk transfer rate

and

\[ T_a = \max_{i=1}^{n} \left( t_{avg\text{-}access} + \frac{S_i}{D} \right) \]  

(2)

Noting that \( \sum_{i=1}^{n} S_i \) is the average frame size over \( n \) successive frames, equation (1) may be rewritten as Since multimedia frames are large, it is reasonable to assume that disk transfer time of a frame exceeds the disk access delay. With this assumption, we may rewrite (2) as where \( S_{\text{max}} \) is the size of the largest frame among \( n \) successive frames.

\[ T_s = t_{avg\text{-}access} + \frac{S_{\text{avg}}}{D} \]

Since it follows that

\[ S_{\text{avg}} \leq S_{\text{max}} \]

\[ T_s \leq T_a \]

Thus, synchronous \( n \)-frame striping is the most optimal storage method for the multimedia workload.

**Uniform Striping**

A stripe containing \( n \) frames, however, impairs the performance of composite files. Recall that in file composition a new file is composed from frames of existing files. Composition is accomplished by creating new maps, without having to replicate data on the disk. Thus it is possible to compose an "unbalanced" file whose frames reside entirely on one disk. Data throughput to such file would be limited by the transfer rate of a single
disk. When sufficient number of unbalanced files are created, the situation becomes similar to the suboptimal case of load sharing.

In order to avoid the creation of unbalanced files, the stripe is modified to contain a single frame. The resulting uniform striping ensures that every frame of every file is balanced across all disks.

**Performance Penalty of Uniform Striping**

If a file is stored on the disk contiguously, uniform striping provides the same performance as synchronous $n$-frame striping. If the file is not contiguous, uniform striping becomes less efficient due to disk seek times between frames. Let $T_n$ denote the time required to access $n$ frames, each striped uniformly across $n$ disks. In the worst case, involving a disk seek for every frame, $T_n$ is computed as follows:

$$T_n = n \cdot t_{avg\cdot access} + \sum_{i=1}^{n} \frac{S_i}{n \cdot D}$$

Thus, in the worst case, $T_n$ differs from $T_s$ and $T_a$ computed in (1) and (2) by at most

$$\boxed{(n-1) \cdot t_{avg\cdot access}}$$

For a small number of disks, and given that for the most part files are stored in file containers contiguously, this difference is insignificant. The small performance penalty of uniform striping may well be justified by frequent use of the file composition facility.

**Data Structures With Multiple Disks**

The disk related data structures described in the previous section extend straightforwardly to support uniform striping. Since uniform striping leads to symmetric disk layouts, provided that every disk offers the same storage and bandwidth capacity, it is sufficient to record the structure of a single disk; the same structure applies to all the disks in the stripe set. Figure 7-10 shows an example of the physical and frame maps that record a single 500 byte frame striped uniformly on two disks.
Figure 7-10 Striping Structures

Stripe Sets with Disparate Disks

As we mentioned earlier, the optimality of the uniform striping is derived from its ability to provide the maximum available concurrency for access to each frame. The discussion of the uniform striping made an implicit assumption that the transfer rates supported by all disks in the stripe set are equal. Given this assumption, the maximum concurrency is, in fact, achieved by spreading each frame evenly across all disks by using striping units that are equal in size.

In practice, however, a stripe set may comprise disparate disks that support different data transfer rates.

The technique of uniform striping may be extended easily to support stripe sets that contain disks with varying transfer rates. In the case of disparate disks, the maximum concurrency is achieved if the sizes of the striping units are proportional to the transfer rates of the corresponding disks.

We will illustrate this by an example that shows how to subdivide a frame into striping units in order to achieve the maximum concurrency in the case of disparate disks.

Suppose that a stripe set contains \( n \) disparate disks. Let \( T_i, \ i \in [1..n] \) denote the maximum data transfer rate supported by the \( i \)th disk. Let \( L \) denote the size of a frame. The maximum concurrency is achieved if the frame is split into \( n \) striping units, each having the size \( l_i \), computed as follows:

\[
l_i = \frac{L}{\sum_{j=1}^{n} T_j} \cdot T_i
\]
7.5 Processing of Requests

The Storage Server receives two types of requests: administrative and file access.

The following administrative requests are supported:

- operational state
- shutdown.

The operational state request is initiated by the Administration Server. The purpose of the request was discussed in Chapter 6.

The shutdown request is initiated by a management entity. The purpose of the request is to inactivate the Storage Server. As part of the shutdown process, the Storage Server commits all critical in-memory data-structures to stable storage. The data structures are restored by the recovery process during restart.

The following file access requests are supported:

- Allocate Container
- OPEN
- CLOSE
- READ
- WRITE
- DELETE

The Storage Server services the Read and Write requests according to the deadline driven algorithm described in Chapter 6. The time of arrival of the client’s first request is used as the basis for computing the deadlines of the client’s subsequent requests. If the first request arrives at time \( t \), the deadline of the \( n \)th request from the same client is \( t + (n-1)f \), where \( f \) is the frame rate parameter specified by the client.

Calculating the deadlines as offsets from an absolute time reference, as opposed to the time of a request’s arrival, ensures that performance guarantees given to all the clients will be met even if some requests arrive early. This scheme of computing the deadlines serves as an automatic defense mechanism against malicious or malfunctioning clients who transmit requests at a rate exceeding their specified frame rate parameters. The scheme, however, does not preclude the early arrival of requests, allowing the clients to perform frame read-ahead.

A late request is treated as if it were the very first request; that is, the time of arrival of the late request is used as the new basis for computing the deadlines of the client’s subsequent requests.

The remainder of this section provides a functional description of procedures performed by the Storage Server for each file access request.
Allocate_Container

In order to allocate a file container, the Storage Server searches the free list to find a contiguous disk area of sufficient size to accommodate the container. A container descriptor is created and anchored in a free slot of the Container Id Table. The index of the slot used for anchoring the descriptor is assigned to the allocated container as a unique container identification. The size and location of the allocated disk area is recorded in the container base.

The Container Id Table is checkpointed to disk. The saved structure is used to recover the in-memory free list, in case of a Storage Server crash.

Upon successful completion of the Allocate_Container procedure, the Storage Server replies to the client with a protocol message containing the container identification.

Open

An OPEN request initiates a virtual session between a client and the Storage Server. Processing of the OPEN request consists mainly of allocating and initializing the context block for the new session.

The Storage Server allocates two transmit buffers, if the file is opened for non-broadcast read access, and two receive buffers, if the file is opened for write access. The Storage Server uses a broadcast buffer cache for files that are accessed in broadcast mode. Buffer pointers, the client’s network address, and the client’s performance parameters are stored in the context block.

The Storage Server maintains the client’s position within the file in a context variable called the current frame pointer. The current frame pointer contains a logical frame number of the frame that will be returned for the next read request or updated by the next write request. If the file is accessed for read only, the current frame pointer is initialized to point to the first frame of the file. If the file is accessed for write, the current frame pointer is initialized to point past the last frame of the file.

Upon successful completion of the OPEN procedure, the Storage Server responds to the client with a protocol message containing the context identification for the current session.

READ

A client sends a READ request to the Storage Server for each frame that it wishes to retrieve from the file. If a file contains \( n \) frames, the client will transmit \( n \) READ requests in order to read the entire file.

The Storage Server distinguishes between the broadcast and non-broadcast READ requests, depending on the file access mode.
Non-Broadcast READS

A READ request identifies the frame that the client wishes to retrieve and provides a prefetch hint for the frame that is most likely to be requested next. The prefetch hint is specified as an offset from the requested frame. The requested frame may be specified in one of the following terms:

- a logical frame number
- a temporal offset
- an offset from the current frame pointer.

If the current frame is specified as a temporal offset, the storage server consults the temporal map of the file in order to convert the temporal offset to a logical frame number.

Once the logical frame number of the requested frame is determined, the Storage Server checks to see whether the frame is already present in its prefetch buffer. One of the two transmit buffers allocated for the session alternately serves as the prefetch buffer. If the frame exists, the Storage Server transmits it to the client and issues a read-ahead of the frame identified by the prefetch hint. The current frame pointer is updated to point to the prefetched frame. If the frame does not exist, the Storage Server initiates a disk transfer for the requested frame. The processing of the current READ request is delayed until the disk transfer completes.

Broadcast READS

Files accessed in broadcast mode may only be read sequentially. For every file opened for broadcast, the Storage Server caches a sequence of most recently read frames. The broadcast cache is shared equally by all files that are accessed for broadcast. Thus, the length of the cached sequence of frames may vary dynamically depending on the number of files that are currently opened for broadcast.

A broadcast READ request identifies the desired frame in terms of an offset from the current frame pointer. The relationship of the desired Logical Frame Number to the range of LFNs of the cached frames determines the way in which the request is processed. If the desired LFN falls within the range of the cached frames, the desired frame is transmitted from the cache. If the desired LFN falls below the range, then the oldest frame in the cache, i.e. the one with the smallest LFN, is transmitted to the client. In this case, the client misses all the frames in the LFN range bounded by the desired LFN and the oldest LFN. If the desired LFN falls above the range, then the desired frame is brought from disk to the broadcast cache and transmitted to the client. In this case, if the length of the cached sequence is at its maximum, the new frame replaces the oldest frame of the sequence.
Write

The Storage Server distinguishes between composition and non-composition WRITE requests, depending on the file access mode.

Non-Composition Writes

A non-composition WRITE request contains data of a new logical frame. The new frame is assigned an LFN based on the value of the current frame pointer. The assigned LFN determines the logical placement of the new frame in the file. If another frame with the same LFN already exists, the new frame replaces the existing one; otherwise the new frame is logically appended to the file.

The Storage Server uses the new frame to construct a stripe. The stripe consists of the frame data surrounded by a disk header and trailer. The disk header contains information that describes the frame in terms of its size, LFN, temporal offset, and the file and container identifications. This information is used to restore the in-memory data structures that may be lost during node failures. The disk trailer contains a checksum computed by the Storage Server for the frame data. The checksum is used to detect incomplete or corrupted frames caused by partial disk writes.

The strategy for determining the physical placement of the new frame on disk depends on the logical position of the frame in the file. If the frame is appended to the file, the Storage Server places the frame in the free space of the file container. If the new frame replaces an existing frame, the Storage Server attempts to reuse the space occupied by the old frame.

In the case of appending a new frame to the file, the Storage Server searches the list of container descriptors to locate a contiguous container fragment of sufficient size to store the frame. If a fragment of sufficient size exists, the Storage Server splits the stripe into striping units of equal size and writes unit $i$ to disk $i$, where $i$ ranges from 1 to the number of disks. The stripe is partitioned in such a way that the disk header of every frame is always mapped to the first disk in the stripe set. The container descriptor corresponding to the contiguous fragment is updated to reflect the change in the container space as follows: the write pointer is advanced by the size of the frame, and the free space of the fragment is reduced by the size of the frame.

In terms of space allocation, containers behave as wrap-around log-structured files. New frames are appended at the end of the log. The logical boundaries of the log vary dynamically as files are created and deleted. The current state of the log is completely described by the list of container descriptors. The log-structured view of containers, in terms of space allocation, is illustrated in Figure 7-11.
Figure 7-11 Container as a Log-Structured File

If the file container does not have sufficient contiguous space to fit the appended frame, the Storage Server attempts to extend the container. Attributes specified at the time of container creation indicate whether it is permissible to extend the container. Extending a container involves allocating a contiguous disk block from the free list, updating the container base, and checkpointing the Container Id Table to disk. The reason for checkpointing is the same as in the case of allocating a new container.

Replacing an existing frame with the new frame results in two cases depending on the relative sizes of the frames. If the new frame is smaller than the existing one, then the Storage Server rewrites the existing frame in place. If the new frame is larger than the existing one, then the Storage Server writes the new frame as if it were appending it to the file. In this case, a new container descriptor is created to point to the space occupied by the existing frame and linked to the container as a new fragment.

Composition Writes

A composition WRITE request contains a list of descriptors specifying sequences of existing frames to be included in the composed file. A file may be composed with one or more composition requests. A single composition request with \( n \) descriptors is functionally equivalent to \( n \) composition requests, each containing a single descriptor.
The Storage Server composes a new file by creating file maps that point to the existing frames. File composition does not involve any movement or replication of data on the disk.

For the purposes of recovering the new maps in the case of node failures, the Storage Server creates a dummy file corresponding to the composed file. The dummy file contains all the original composition requests as frames. During the recovery process, the maps are "recomposed" from the original requests.

Delete

The DELETE request identifies a file or a container that the client wishes to remove from the file system. A container must be empty before it can be deleted. A file cannot be deleted if it contributes frames to a composed file. Interest lists and reference counts, as implemented by the voice file server of the Etherphone project, may be used to distinguish files that participate in a composition from those that do not.

Deleting a file involves releasing the space held by the file to the associated file container. The Storage Server converts the physical map of the deleted file into container descriptors and links the descriptors to the container as new fragments. The file identification of the deleted file and the deletion time stamp are checkpointed to disk. The checkpointed FID is used to remind the Storage Server not to restore the deleted file during the recovery process. The FID of the deleted file may be reassigned to a new file by the Administration Server. The deletion time stamp is used to distinguish between live and deleted files that have the same FID.

Deleting a container involves clearing the container base in the Container Id Table, checkpointing the modified portion of the Container Id Table to disk, and releasing the disk space previously held by the container to the free list.

7.6 Recovery

While the Storage Server is active, the file meta-data is cached in main memory. The Storage Server relies on the meta-data for accessing the existing files and creating new ones. The meta-data comprises file headers and maps, the Container Id Table, container descriptors, and the free list. During a node failure these structures may be lost or corrupted, preventing the Storage Server from accessing files. The purpose of the recovery process is to restore the in-memory data structures based on the data committed to disk.

The recovery process distinguishes between recovering after a node failure and restarting after a clean shutdown. In the case of restarting after a clean shutdown, the meta-data structures are restored from the checkpoint performed by the shutdown procedure.

The process of crash recovery, performed after a node failure, is more complicated. The Storage Server begins the crash recovery process by restoring the checkpointed Container Id Table. Every non empty entry of the Container Id Table contains a container base. A container base records the original allocation and all subsequent extensions of the corresponding container. This information is used to restore the free list.
The second step of the crash recovery process is to restore the file headers and maps. The Storage Server restores these structures based on the information recorded in the disk header of every frame. It is possible to recover the file headers and maps in one sequential scan of the disk that contains the header portion of the original stripe. During the sequential scan of the disk the Storage Server bypasses headers that belong to deleted files.

The next step of the crash recovery process is to restore the maps of composed files. The Storage Server scans the array of file headers to identify the composed file. For each composed file, the corresponding dummy file is read in order to retrieve the original composition requests. The maps are recomposed based on the original composition requests.

The final step of the crash recovery process is to recreate the list of container descriptors for each container. The spatial state of containers is restored based on the recovered file maps. The space occupied by a file is described completely by the file’s physical map. The Storage Server scans every physical map, subtracting the allocated blocks from the corresponding container. This process creates a list of container descriptors reflecting the spatial state of each container.

Note that the meta-data structures restored by the crash recovery process may be inconsistent across multiple Storage Servers. The reason for the inconsistency is that the Storage Servers may have stored different amounts of frames for the same file before the failure. The inconsistency manifests itself in the CMread routine of the CMFS Interface, when a read request results in data from some Storage Servers and the end-of-file indication from others. The CMread routine masks the discrepancy by treating this situation as the end-of-file condition, even though some Storage Servers may have additional portions of the frame.

7.7 Reliability and Fault Tolerance

The recovery process described in the previous section does not address the problem of permanent disk failures. The failure of a single disk in any Storage Server makes the entire Logical Storage System unavailable. Since the probability of a permanent failure increases with the number of components, the reliability becomes a critical concern in the design of parallel storage systems, such as CMFS. The ability to provide fault tolerance is a prerequisite for the viability of disk and node striping as a practical high-bandwidth storage architecture.

Reliability is commonly achieved by redundant storage that may be used to recover the data of a failed component. Redundant storage may contain a copy of the original data, as in systems with mirrored disks, or it may contain error correction codes (ECC) which can be used to regenerate the contents of a failed device, as in RAID [PAT88].

The mirrored disks approach may be appropriate for a configuration with a small number of nodes and a small number of disks in a stripe set on each node. For a large configuration, however, this approach may become prohibitively expensive, since all disks must be duplicated.
Error correcting codes reduce the amount of redundant storage at the expense of additional processing required for the ECC computation. Parity striping, for example, needs only one ECC disk per stripe set to provide fault tolerance for a single disk failure, assuming that the position of the failed disk in the stripe set is known [PARK86]. The parity disk contains one parity bit for a group of interleaved bits of a stripe. A missing striping unit in a stripe may be regenerated based on the parity information and the remaining units of the stripe.

The performance of parity striping, in terms of the required disk transfers, is optimal in situations where the entire stripe is accessed as a unit [PAT88], which is the case in synchronous striping. Since we advocated synchronous striping for multimedia workloads, the parity striping based redundancy provides the most effective way of achieving fault tolerance in the CMFS environment.

Parity striping may be implemented on the node or disk striping level. On the node striping level, the CMwrite routine would compute the parity per frame and store it on the redundant parity Storage Server. On the disk striping level, each Storage Server would compute the parity per stripe and store it on the redundant parity disk. Computing the parity on the disk striping level is more efficient, since the computation is performed on a smaller data unit.
8 Prototype Implementation

This chapter describes the prototype implementation of the CMFS design. All CMFS components specified by the design have been implemented.

Software Environment

The prototype is implemented in the Ultrix\textsuperscript{1} environment using the standard UDP/IP socket library for communicating among the components. All of the prototype code runs in user mode. We did, however, modify the standard socket layer of Ultrix to provide greater efficiency in transmission of messages.

Hardware Environment

The hardware platform for the prototype is a DECstation 5000, a RISC workstation based on the 25MHz R3000 CPU chip [KANE89]. External devices attach to the CPU via a high performance I/O bus called the Turbochannel. The Turbochannel provides three options slots for external adapter cards. In addition to the three options slots, a DECstation 5000 provides two internal communication ports: SCSI and Ethernet. A complete technical description of a DECstation 5000 may be found in reference [DEC90].

The hardware available for configuring the prototype included three DECstations 5000, each equipped with a local swapping device, an FDDI LAN, and 4 RZ56 SCSI disks. One of the workstations is designated to run the CMFS Interface software. Each of the remaining two workstations runs the Storage Server software and serves as a storage domain containing two RZ56 disks. The Administration Server may run on any of the three workstations; we arbitrarily ran it on the client node. The hardware configuration of the prototype is illustrated in figure 8-1.

\textsuperscript{1} Ultrix is Digital's implementation of Unix based on 4.3 BSD [LEF89].
Figure 8-1  Hardware Configuration
CMFS Interface

The CMFS Interface is implemented as a library of callable routines. All the routines described in Chapter 5 have been implemented. The following description of the CMFS Interface routines emphasizes the implementation of node striping.

CMopen

The CMopen routine allocates a context block for the file in a shared memory segment. The pointer to the allocated block is used as the file descriptor in all subsequent references to the file.

The CMopen routine creates a separate UDP socket for each Storage Server of the Logical Storage System and stores the created socket descriptors in the context block for the file. The reason for creating an individual socket for each Storage Server, instead of creating a single multiplexed socket for all Storage Servers, is to reduce the number of copy operations involved in reassembling the frame by the CMread routine.

CMwrite

The CMwrite routine splits the data buffer supplied by the client into $n$ node striping units of approximately equal lengths, where $n$ is the number of Storage Servers. Each node striping unit is transmitted to the corresponding Storage Server in 32K UDP packets.

Each fragment contains a small protocol header followed by the client’s data. The Scatter-gather I/O is used to concatenate the header and data, avoiding a possible copy operation. The protocol header specifies the size of the node striping unit destined for the Storage Server, the size of the current fragment, the logical fragment number, and the fragment’s relative offset into the frame. This information is used by the CMread routine during retrievals to reassemble the original frame.

CMread

The CMread routine retrieves a frame striped across multiple Storage Servers by requesting a node striping unit from each Storage Server. Each striping unit is received in 32K fragments via a socket associated with the corresponding Storage Server. The CMread routine uses the fragment’s position information, which was originally recorded by the CMwrite routine, to assemble a complete frame from the node striping units.

In our implementation, the process of assembling a frame requires only one copy operation per fragment. The reason for this is that the relative offset of the node striping unit into the frame is implicitly determined by the socket from which the unit is received.

---

1 The architectural limit for the maximum size of a UDP datagram is 64K bytes. However, both the 4.3 BSD and Ultrix operating systems limit the maximum UDP datagram size to 32K bytes.
given that the fragments comprising the striping unit are ordered correctly. In the case of a single multiplexed socket, it would be necessary to examine the fragment’s protocol header in order to determine the fragment’s position within the frame. The examination of the header would require an extra copy operation per fragment.

During the testing of the prototype, we discovered that the CMread routine was not able to keep up with the rate of fragment arrivals on multiple sockets, resulting in data overruns and subsequent fragment losses. We solved this problem by modifying Ultrix to allocate more buffer space to a socket. Since the CMread routine implements a single frame read-ahead, it was sufficient to make the internal size of the socket buffer large enough to hold one node striping unit.

**Administration Server**

The Administration Server process uses the standard Unix file system to implement directory services. For each CMFS file, the Administration Server creates a "witness" file of the same name in the Unix file system. The witness file contains the FID of the corresponding CMFS file.

The admission control algorithm, as described in Chapter 6, was not implemented.

**Storage Server**

The Storage Server communicates with all clients via a single multiplexed UDP socket, called the request socket. File access requests are demultiplexed based on the context identifier field of the protocol header. Multithreading of requests is accomplished by asynchronous I/O to disks.

The Storage Server accesses disks as raw devices using a character driver. The disk character driver transfers data directly between a user’s buffer and the disk adapter, bypassing the file buffer cache. The Ultrix operating system extends the functionality of a standard BSD 4.3 character driver by including the n-buffered I/O feature. The feature allows asynchronous access to a raw device through the character driver. The Storage Server uses asynchronous I/O to access each disk in its storage domain.

Two methods are available to the Storage Server process for determining the completion of asynchronous I/O:

- the process may elect to be notified asynchronously via the SIGIO signal
- the process may poll the driver by the IOCTL system call.

Our implementation uses the polling method to check for the completion of asynchronous I/O. It is not possible to use signals reliably in the environment where multiple I/O requests are issued to multiple disks for the following reasons:

- the signals are not queued by the kernel; this may prevent a process from catching every signal associated with asynchronous I/O completion
● Neither Ultrix nor the BSD 4.3 operating system provides a way of associating a signal with the corresponding I/O request; this makes it impossible to distinguish which disk of the stripe set completed its I/O request.

We mentioned earlier that the CMwrite routine of the CMFS Interface software transmits a portion of a logical frame in 32K fragments. The Storage Server converts each fragment into a stripe by overlaying the fragment’s protocol header with a disk header and appending a disk trailer at the end of the fragment’s data. The stripe is then divided into n striping units, where n is the number of disks in the stripe set, and each unit is written asynchronously to a corresponding disk of the stripe set.

A single read request from a client causes the Storage Server to retrieve and transmit all the fragments comprising the stored portion of the frame. The Storage Server assembles a complete fragment in its prefetch buffer by reading asynchronously a striping unit from each disk in the stripe set.

Upon the receipt of the client’s read request, the Storage Server checks its prefetch buffer for the existence of the first fragment of the requested frame. If the fragment exists, the Storage Server transmits it to the client and initiates an asynchronous retrieval of the second fragment. Read requests for the second and all subsequent fragments are generated internally by the Storage Server. The internal read requests are written to the request socket, as though they were generated and transmitted externally by the client. If a read request for a fragment arrives before all disk I/Os to the stripe set complete, the Storage Server delays the read request by writing it back to the request socket. The resulting effect is that of polling for the I/O completion. After the last fragment of the frame has been transmitted, the Storage Server prefetches the first fragment of the next frame.

We did not implement the deadline driven scheduling as described in Chapter 6. However, the facilities provided by the multiplexed request socket and by the internally generated requests for frame fragments may be easily extended to include the scheduler. For example, preemption may be implemented by placing the internally generated fragment request associated with the preempted frame transfer on a temporary queue and writing it back to the request socket when its priority becomes the highest.

**Double Mapped Buffer**

Transmitting messages to the network through the standard socket layer of Ultrix entails two data copy operations: the first copy involves moving data from user space to kernel space, and the second copy involves moving the data from kernel space to the network adapter. In an effort to increase the data throughput, we modified Ultrix to eliminate the first copy operation. In the modified, version data is transferred directly from a user’s buffer to the network adapter.

The modification is implemented by introducing a new system service, MAP_TO_KERNL, and altering the SOSEND routine of the socket layer. The MAP_TO_KERNL system service takes two arguments: a pointer to a buffer, and the buffer’s size. The implementation of the service requires the buffer to reside in shared
memory. The \texttt{MAP\_TO\_KERNEL} system service locks the buffer in physical memory and maps it into the kernel by allocating system PTEs pointing to the physical pages of the buffer. The double mapped buffer is now accessible by any user process that is attached to the shared memory segment containing the buffer and by any system routine.

The standard \texttt{SOSEND} routine copies data from a user's buffers into a kernel buffer and creates a chain of mbufs that point to the data in the kernel buffer. The network driver references the data through the mbuf chain. The \texttt{SOSEND} routine was modified to create a chain of mbufs pointing to the data in the user's buffer. Since the user's buffer is locked in memory and mapped into the system space, the network driver can still reference the data through the chain of mbufs.

We did not implement the elimination of data movement between the kernel and user space for receiving messages from the network through a socket. Implementing this feature would considerably extend the prototype development time, since it involves non trivial changes to the FDDI driver and the IP, UDP, and socket layers. Changes to the driver would be required to ensure that a received message is mapped on physical page boundaries. This would require a special treatment of IP and UDP headers, such as the trailer protocol [LEF89], for example, resulting in the need to alter the IP and the UDP layers. The socket layer would need to be modified to map the physical pages containing the received data into the user's virtual address space.
9 Experimental Results

This chapter presents performance measurements of the CMFS prototype. The evaluation of the prototype performance is conducted in four parts.

First, we describe the predicted and the measured performance of the system components, independently from CMFS. The purpose of these measurements is to determine the upper bound on the performance of CMFS and to compare the actual performance attained by the prototype to the system maximum.

Second, we present the measured performance of the CMFS prototype in several different configurations.

Third, we present subjective evaluation of the quality of services provided by CMFS to a sample video application, which implements a VCR-like functionality for managing video and sound.

Finally, we discuss the implications of the performance attained by the prototype. The purpose of this discussion is to examine the ability of CMFS to support some practical applications.

9.1 Performance of the System Components

This section presents performance measurements of the system components that are of interest to CMFS. System components relevant to the performance of CMFS are those that are involved in moving data between disks and the network.

System Components

The following system components determine the ultimate performance of CMFS:

- disk I/O subsystem
  - SCSI disks
  - synchronous SCSI bus
  - SCSI adapter
- memory subsystem
- network subsystem
  - socket, UDP, and IP kernel code
  - FDDI driver code
  - FDDI adapter

The system components are shown in Figure 9-1.
Figure 9-1 System Components

The following sections describe the predicted\textsuperscript{1} and the measured performance of the system components. Our predictions and measurements emphasize the retrieval path; that is, the movement of data from the disk to the network, which is expected to be heavily utilized by the multimedia workload.

Disk to I/O Buffer Performance

Predicted:
The path between the disk and the I/O buffer consists of the synchronous SCSI bus, the SRAM buffer of the SCSI adapter, the Turbochannel, and the R3000 CPU. The predicted transfer rates supported by each component along this path are summarized in Table 9-1.

\textsuperscript{1}For hardware components, predictions are based on the cycle times found in the design specifications; for software components, predictions are based on the estimated number of instructions in the code path
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RZ56</td>
<td>1.66 Mbyte/s</td>
</tr>
<tr>
<td>SCSI bus</td>
<td>4 Mbyte/s</td>
</tr>
<tr>
<td>SCSI SRAM</td>
<td>7.1 Mbyte/s</td>
</tr>
<tr>
<td>Turbochannel/R3000 programmed I/O</td>
<td>12 Mbyte/s</td>
</tr>
</tbody>
</table>

**Table 9-1 Disk I/O Subsystem Predicted Transfer Rates**

Given these predicted rates, the R3000 CPU requires about 1.66/7.1 seconds, which is about .23, to transfer 1.66 Mbytes of data from the SCSI SRAM. However, it takes the RZ56 1 second to fill up the SRAM. Since the CPU is busy only for .23 seconds, the predicted CPU utilization for the data transfer is about 23%.

**Measured:**
The maximum rate at which a user process could transfer data from an RZ56 disk to an I/O buffer was measured to be 1.66 Mbyte/s. The R3000 CPU utilization for this transfer was measured to be about 30%.

The maximum rate at which a user process could transfer data from the I/O buffer to RZ56 was measured to be 900 Kbyte/s.

**Discussion:**
The predicted disk transfer rate and the CPU utilization agree with the empirically obtained measurement of 1.66 Mbyte/s and 30% respectively.

This test indicates that transferring data from 3 RZ56 disks concurrently will saturate the system, since the CPU consumption in that case would approach 100%.

The predicted and measured transfer rates of the I/O subsystem components clearly illustrate the current bottlenecks. Although multiple disks and adapters may be attached to the system, the aggregate transfer rate of 7.1 Mbyte/s will saturate the CPU, which is the limitation imposed by the speed of the SCSI SRAM. In fact, independent tests have shown that the system saturates at about 6.2 Mbyte/s [DEC90A].

Note that in all the tests involving the disk subsystem, the SCSI adapter is accessed using programmed I/O and the disk is accessed as a raw device using the character I/O driver.

---

^1 In all tests, the CPU utilization was measured using the standard utilities provided by Ulitrix and confirmed by the cycle soaker process.
Memory Performance

Predicted:
The predicted performance of the copy operation is 16 Mbyte/s.

Measured:
The rate at which the R3000 could perform a memory-to-memory copy operation was measured to be 15 Mbyte/s, which is consistent with the prediction. This rate was measured for the worst-case in which a cache miss occurs for every memory access.

I/O Buffer to the Network, Using UDP

This section reports measurements of data transfer rates from the I/O buffer to the FDDI adapter.

The Ultrix networking code that computes the UDP checksum was not optimized for the RISC architecture of the R3000 CPU, resulting in a severe performance degradation. We disabled the UDP checksum processing for all the tests described in this chapter. The measurements and results reported in this chapter do not include the overhead of computing the UDP checksum.

Predicted:
Transferring data from the I/O buffer to the FDDI adapter involves the following steps:

- a copy between user space and kernel space by the socket layer
- UDP and IP processing overhead
- FDDI driver processing overhead
- FDDI programmed I/O copy from kernel space to the FDDI adapter

The predicted performance numbers for the above steps are summarized in Table 9-2.
<table>
<thead>
<tr>
<th>Component</th>
<th>Predicted Performance</th>
<th>Processing Time for one FDDI Packet (4500 bytes)</th>
</tr>
</thead>
</table>
| copy                   | 15 Mbyte/s            | \[
\frac{4500}{(15 \cdot 10^6)} = .3ms
\]          |
| UDP/IP overhead        |                       | .15 ms                                           |
| FDDI driver overhead   |                       | .17 ms                                           |
| programmed I/O to FDDI| 12 Mbyte/s            | \[
\frac{4500}{(12 \cdot 10^6)} \approx .38ms
\]          |

**Table 9-2 Predicted Packet Transmission Time**

Table 9-2 shows that the predicted CPU time required to transmit a 4500 byte FDDI packet is:

\[ .3 + .15 + .17 + .38 = 1ms \]

The rate of 4500 bytes per 1ms translates to the maximum message transmission rate of 4.5 Mbyte/s, consuming 100% of the CPU.

If we eliminate the copy from user space to kernel space, the predicted CPU time required to transmit a 4500 FDDI packet becomes:

\[ .15 + .17 + .38 \approx .7ms \]

The rate of 4500 bytes per .7ms translates to the maximum transmission rate of 6.4 Mbyte/s.

**Measured:**

We used both the standard and the modified versions of Ultrix to measure the attainable data transfer rate from the I/O buffer to the FDDI adapter. The modified version of Ultrix eliminates the data copy operation from user space to kernel space, which is present in the standard version.

The transfer rate supported by the standard version was measured to be 4.5 Mbyte/s.

The transfer rate supported by the modified version was 6.4 Mbyte/s.

The CPU utilization in both cases was 100%.
DMA is used in the reverse direction, i.e. from the FDDI adapter to the I/O buffer. The transfer rate in this case was measured to be 5 Mbyte/s.

**Discussion:**
We note that the predicted results agree with empirical measurements both for the modified and the standard versions of the Ultrix operating system. We also note that eliminating one data copy operation results in about 40% increase in the transmission throughput.

**Complete Retrieval Path**
Given the above results, we now discuss the complete retrieval path; that is, the movement of data from the disk to the network.

**Predicted:**
The maximum SCSI adapter throughput is 7.1 Mbytes/s consuming 100% of the CPU and the maximum FDDI transmission throughput of 6.4 Mbytes/s consuming 100% of the CPU suggest that the maximum rate at which data can be transferred from SCSI disks to the FDDI adapter is about 3.4 Mbytes/s.

**Measured:**
We verified this hypothesis by running a simple process that issues asynchronous disk reads to 2 RZ56 disks, connected to separate SCSI adapters, and transmits the data to the FDDI. The test showed that the system saturated at about 2.4 Mbyte/s.

**Discussion:**
The above results indicate that the maximum retrieval throughput supported by a single node is between 2.4 and 3.4 Mbyte/s; this suggests the need for a distributed storage architecture as a means of increasing the aggregate data throughput.

**Summary of the Component Performance**
The performance measurements of the system components are summarized in Table 9-3.
<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>RATE in Mbytes/s</th>
<th>%CPU used</th>
</tr>
</thead>
<tbody>
<tr>
<td>RZ56 to I/O buffer</td>
<td>1.66</td>
<td>30</td>
</tr>
<tr>
<td>I/O buffer to RZ56</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>R3000 memory-to-memory copy</td>
<td>15</td>
<td>100</td>
</tr>
<tr>
<td>SCSI SRAM to I/O buffer</td>
<td>7.1</td>
<td>100</td>
</tr>
<tr>
<td>I/O buffer to FDDI adapter</td>
<td>6.4 (modified Ultrix)</td>
<td>100</td>
</tr>
<tr>
<td>FDDI adapter to I/O buffer</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>2 RZ56 to FDDI adapter</td>
<td>2.4</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 9-3 Component Performance Summary

9.2 CMFS Performance

We measured the performance of CMFS in terms of the maximum frame rate obtainable by a client of the system during a sequential file access. The size of all frames is fixed at 7680 bytes, representing an image of 240x320 8-bit pixels. We measured the attainable rate in several Logical Storage Server configurations using the available hardware of 3 DECstations 5000 and 4 RZ56 disks. In all tests the client ran on a separate DECstation 5000. The client program used for the performance tests was counting the number of frames received per second; no application processing was performed on the received frames.

**Configuration 1:** one node and one disk.

The client measured about 16 frames/s, corresponding to a throughput of 1.3 Mbyte/s.

For this test we measured the jitter of the frame inter-arrival times. The jitter is defined as the difference in the arrival times between 2 successive frames. The measurements showed that the jitter varied from 0 to 15ms with the average of about 6ms. We attribute the jitter to the variations in the rotational latency of the disk.

Note that the average jitter in this case is smaller than the frame period of 62ms; in fact, it is within 10% of the frame period.

**Configuration 2:** one node and two disks on separate SCSI adapters.

Although the Storage Server is implemented as a multithreaded process capable of handling multiple disks in a stripe set, we had to perform this test with two instances of the Storage Server process, each handling the stripe set of 1 disk. The reason for this was a bug in the disk SCSI driver which caused the driver to single thread asynchronous disk
requests originating from the same process.\footnote{Due to this bug, we failed to obtain valid jitter measurements for configurations with more than 1 disk. We expect, however, that in the case of 2 disks the jitter will be about $8ms$ and in the case of 4 disks, it will be about $11ms$, based on the statistical modeling reported in [KIM87].}
The client measured over 25 frames/s, corresponding to a throughput of 2 Mbyte/s.

**Configuration 3:** 2 nodes with 1 disk on each node.
The client measured about 32 frames/sec, corresponding to a throughput of 2.6 Mbyte/s.

**Configuration 4:** 2 nodes with 2 disks on each node.
The client measured about 45 frames/s, corresponding to a throughput of 3.5 Mbyte/s.

**Discussion and Enhancements**
While performing the tests, we discovered that the Storage Server implemented the processing of receive requests suboptimally. In response to a read request, the implemented prototype was coded to transmit a prefetched fragment first, followed by an asynchronous read of the stripe set for the next fragment. Reversing the order of these operations results in better pipelining of the network transmission and disk reads. We verified that the second case results in a higher transmission throughput by experimenting with a simple process. The experiments show that the rate of transmission increases by about 100K bytes per disk if an asynchronous disk read is issued prior to the network transmission. With this enhancement, we expect the prototype running as in Configuration 4 above to deliver the throughput of about 4 Mbyte/s.

**Comparison with UFS and NFS**
We measured read data rates supported by the Ultrix File System (UFS) and NFS in configurations similar to those in which the performance tests were conducted for the CMFS prototype.
The UFS and NFS performance was measured by a program that read a 6 Mbyte file in 32K blocks.

UFS was configured on a local RZ56 disk.
The NFS served file system was also configured on RZ56; the client and the server machines were DECstations 5000 connected by FDDI.

We ensured that both UFS and NFS ran with cold caches.
The read data-rate provided by UFS was 1.3M byte/s, at best. The read data-rate provided by NFS was 900 Kbyte/s, at best.
Summary of CMFS Performance

Performance measurements for CMFS, NFS and UFS are summarized in Table 9-4. The column labeled "achieved" reports the data-rates that CMFS was able to provide to clients for read access. The column labeled "system max" is the maximum transmit throughput that can be achieved in a given configuration; the maximum transmit throughput was computed using the performance measurements of the system components and verified by running simple test programs.

It can be seen that the data-rates provided by CMFS for sequential read access scale linearly with the number of nodes. We expect the scalability in the number of nodes to remain linear until the network is saturated. Three nodes with two disks on each node will saturate the 100 Mbit/s FDDI ring.

The data-rates do not scale quite linearly with the number of disks due to the early saturation system saturation on each node. However, the effective utilization of the storage bandwidth on each node is high. In the configuration with one disk, CMFS achieves the data-rate that is 93% of the system maximum, and in the configuration with two disks, CMFS achieves 83% of the system maximum.

In the same hardware configuration, CMFS performs as well as the local Ultrix file system, and outperforms NFS by 50%.

<table>
<thead>
<tr>
<th>File System</th>
<th>1 node/1 disk</th>
<th>1 node/2 disks</th>
<th>2 nodes/2 disk</th>
<th>2 node/4 disks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>achieved</td>
<td>achieved</td>
<td>achieved</td>
<td>achieved</td>
</tr>
<tr>
<td></td>
<td>system max</td>
<td>system max</td>
<td>system max</td>
<td>system max</td>
</tr>
<tr>
<td>CMFS</td>
<td>1.3</td>
<td>1.4</td>
<td>2</td>
<td>2.4</td>
</tr>
<tr>
<td>UFS</td>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFS</td>
<td>.9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9-4  CMFS Performance Summary

All measurements are in megabytes per second (Mbyte/s)
9.3 Subjective Evaluation

In order to experiment with the ability of CMFS to support continuous media, we implemented a simple VCR-like application for managing video and sound. The application supports viewing of individual video frames, composition of files, and playing video and audio files separately or simultaneously. Reverse playback and frame-skip options are also supported by the application. Figures 9-2 and 9-3 show the control panels used for playback and file composition.

Figure 9-2 Control Panel for Playback
In order to support audio, we equipped the client DECstation 5000 with a Lo-Fi board, an 8KHz speech digitizer designed for the Turbochannel bus. Lo-Fi allows real-time recording and playback of sound.

Unfortunately, at the time of the prototype development, no video digitizers were available for the Turbochannel. We digitized a 4 second video sequence off-line using the frame grabbing hardware of the Silicon Graphics IRIS workstation. The digitizer converts an NTSC image into three 480x640 pixel planes corresponding to the YIQ components. Each pixel is represented by 8 bits. We extracted the luminance component and reduced it down to 240x320 by an averaging 2x2 square filter. The resulting sequence of 240x320 images was transferred off-line to CMFS. The sound track associated with the video was recorded using the Lo-Fi board. Several unrelated audio and video sequences were also recorded to experiment with the file composition facility.

Video images were displayed by the X11 server enhanced with MIT's shared memory extension. When CMFS was configured to provide the highest bandwidth, the rate of display exceeded 30 frames per second, each frame of the size 240x320 bytes. Subjectively, the perceived quality of the display was similar to that of an NTSC broadcast. Each frame, however, when viewed individually, showed some visible artifacts caused by digitizing an interlaced image.

Displaying a composed file also resulted in a smooth motion portrayal, quite similar in quality to an NTSC broadcast. Longer disk seek times required to switch among the original sequences did not produce any visible "jerkiness" in the playback of the composed sequence.
When two client processes running on the same node were displaying a video sequence in two separate windows, a degradation in motion portrayal became visible, since each client process was displaying about 16 frames a second.

A broadcast mode display of a file in two different windows by separate client processes running on the same node appeared synchronized and in phase. Each client process, however, was able to display frames at the rate of 18 per second, a limitation caused by the FDDI adapter receive throughput.

We did not attempt to synchronize audio and video streams due to a short 4 second video sequence and the absence of time stamps that would have been inserted into both streams had the real-time video recording facilities been available.

9.4 Implications of CMFS Performance Results

Although the CMFS performance meets the objectives of high-bandwidth and scalability, it is important to consider whether the achieved throughput of 3.5 Mbyte/s allows CMFS to support some practical applications.

Certainly, the ability of CMFS to support an application environment depends on the workload characteristics specific to that environment. In this section we consider a hypothetical workload that may be presented by a multimedia mail system, and the ability of CMFS to meet the requirements of this hypothetical environment.

A 4.2 BSD Unix file system study observes that a typical administrative environment, in which users exchange mail and documents, is characterized by 5 active users in a ten-second period. A user is considered active if he or she performs any file system activity in a ten-second interval.

In order to estimate the average number of active users in the hypothetical multimedia mail system, we need to consider the duration of access to a multimedia mail message. Assume that it takes 30 seconds to playback a multimedia document. In the administrative environment, described by the Unix file system study, 15 users would have an opportunity to be active in a thirty-second interval. We assume then, hypothetically, of course, that a multimedia mail system is characterized by 15 active users.

CMFS, configured for its maximum bandwidth of 3.5 Mbyte/s can support about 17 MPEG compressed video streams\(^1\). Thus, with admission control, CMFS can support the workload of the hypothetical multimedia mail system. The admission control is still needed because the maximum number of clients supported by CMFS does not exceed greatly the average number of active clients, a condition required by statistical multiplexing.

It is difficult to determine the ratio of active users to maximum supported user at which the process of statistical multiplexing can be used instead of admission control. We may gain some insights, however, by looking at typical file system installations. A small NFS server, for example, can normally support 20 users [HOWARD88]. Assuming an administrative environment of 5 active users, the ratio is 4 to 1. In order to achieve the

\(^1\) MPEG compression is specified at 1.5 Mbits/second.
same ratio, CMFS needs to be able to support at least 60 simultaneous users. Assuming MPEG compression, the requirement of 60 simultaneous users translates to the throughput of 12 Mbyte/s.

We may use the rate of 12 Mbyte/s as a tentative guideline for sufficient bandwidth in a small environment. It is possible to achieve this rate using 3 DECstations 5000 and 2 RZ56 disks on each node.
10 Conclusions and Suggestions for Future Work

We conclude this thesis by summarizing the results obtained by our work and suggesting improvements for the design of CMFS.

10.1 Summary and Conclusions

This thesis described the design and implementation of CMFS, a file service for continuous media. The need for CMFS was motivated by the emergence of distributed multimedia applications that require a file service capable of supporting continuous media and the inability of conventional file systems to provide the necessary support.

Analysis of Continuous Media

We characterized continuous media in terms of their temporal nature, large frame sizes, high frame rates, and access patterns. The analysis of the characteristics showed that continuous media require support for sustained bandwidth, synchronization, large frames, and sequential time-constrained access. We compared the characteristics and requirements of continuous media with those of conventional files and showed that they differ fundamentally in terms of size, I/O request units, performance sensitivity, resource sharing, and scalability notions.

Goals

Based on the characteristics and requirements of continuous media, the following goals were established for the design of CMFS: high-bandwidth scalable storage architecture, performance guarantees, support for stream synchronization, and access paradigms suitable for continuous media.

Node Striping

High-bandwidth and scalability were achieved by a distributed storage architecture based on the concept of node striping. The technique of node striping increases the data transfer rate by spreading each frame across multiple nodes. We analyzed several methods for distributing data across multiple nodes, including uniform striping, asynchronous striping, and load sharing, and showed that uniform striping is the best method for the multimedia workload.

The performance of the prototype proved the feasibility of node striping as a high-bandwidth scalable storage architecture, a result obtained independently by the designers of the Swift project [CAB91]. Node striping generalizes the concept of RAID by supporting a distributed array of storage devices. Unlike RAID, the distributed nature of node striping allows load sharing, growth, and the flexibility of integrating diverse storage devices such as magnetic, optical, and solid state disks, RAID itself, and any future storage technologies. Furthermore, the similarity between the node striping and disk
striping software suggests that, once the code for the general case of node striping is developed, it may be easily adapted to provide the functionality of RAID in a local environment, as evidenced by the CMFS implementation.

Parity striping, as used in synchronous RAID systems, was suggested as an effective method for providing fault tolerance in the CMFS environment. Parity striping, requiring only an exclusive OR operation, is attractive because of its simplicity, ease of implementation in software, and its particular suitability for the multimedia environment, in which access is performed to the whole stripe containing a large frame.

**Admission Control**

We argued that the best effort file service, based on the statistical multiplexing, cannot provide performance guarantees required by continuous media. We justified the need for admission control by showing that the average aggregate bandwidth requirement of a few multimedia clients has a high probability of exceeding the capacity of the current storage technology.

Performance guarantees were provided by the admission control process using the efficient deadline driven scheduling algorithm. The algorithm was adapted for scheduling the file access requests.

**Synchronization**

Synchronization of related data streams was facilitated by the inclusion of time stamps in every frame and the ability to reference frames by temporal file offsets. We showed that the time stamps combined with client-paced transmission of frames provides an effective and flexible synchronization mechanism.

The ability to stream data using client-paced transmission is an interesting result. To the best of our knowledge, existing video file systems use server-paced transmission by employing rate control for stream synchronization.

**Access paradigms and Storage Abstractions**

Access paradigms provided by CMFS for multimedia objects included a file composition facility, directional, skip-frame, and broadcast file access, and frame reference by temporal offsets.

Storage abstractions included logical frames and file containers.

Containers are used by CMFS for efficient disk space allocation strategies and for physical clustering of files related by performance requirements. Applications use containers to reserve disk bandwidth for simultaneous playback of related files such as video and associated audio.
Experimental Results based on the Prototype

A prototype of CMFS was built using the Unix operating system and an FDDI-based local-area network. The performance results showed that the data-rates supported by CMFS for sequential reads scale linearly with the number of nodes. CMFS achieves high efficiency by providing data-rates that are within 7 to 17% of the system maximum.

In similar configurations, CMFS performed as well as the local Unix file system and outperformed NFS by a factor of 1.5. Configured on 2 nodes with 2 disks on each node, CMFS was able to provide the read data-rate in excess of 3.5 Mbytes/s, outperforming UFS by a factor of 3 and NFS by a factor of 4.

A VCR-like application for managing audio and video was developed to evaluate the ability of CMFS to support continuous media. Subjective evaluation of this application showed that the quality of the real-time video display was similar to that of an NTSC signal.

Summary

In summary, this thesis made the following contributions:

- Analysis of the characteristics and requirements of continuous media.
- Empirical proof for the feasibility of streaming data continuously at the rate required by video applications. It is interesting to note that the streaming of data was enabled by frame read-ahead rather than caching.
- Empirical proof for the feasibility of node striping as a high-bandwidth scalable storage architecture.
- Analysis of striping methods suitable for large frames and multimedia workloads.

10.2 Suggestion for Future Work

The following issues related to the design and implementation of CMFS require further attention:

- operating system support for continuous media
- stochastic admission control policies
- management of requests for admission
- extended CMFS functionality.
Operating System Support for Continuous Media

We implemented the CMFS prototype in a general purpose Unix environment. A time-sharing operating system, such as Unix, is poorly suited for the high-data rates, synchronization, scheduling and performance guarantees requirements of continuous media. Current work in operating systems is addressing the need to provide facilities that support continuous media. The proposed operating system extensions enable deadline scheduling of light-weight processes, minimal context switching, user-kernel shared memory [ANDER91], DMA transfers directly between device adapters[PAS91], asynchronous event handling and user-mode device drivers [NAK91].

The design and implementation of CMFS needs to be integrated with the proposed extensions. Incorporating the described extensions into CMFS would result in higher performance and more graceful implementation, compared with the prototype implemented in standard Unix.

Stochastic Admission Control

The admission control algorithm proposed in the design of CMFS gives performance guarantees based on the fixed request execution time. The request execution time for a client is derived from the frame size parameter specified by the client on the open call. It is likely, however, that each client will use JPEG or MPEG image compression resulting in varying frame sizes and consequently varying request execution times. CMFS could support a larger number of simultaneous clients if the scheduling and admission control algorithms were based on the stochastic knowledge of the request execution time. In order to gain this knowledge, statistical modeling of the frame size distribution resulting from image compression is needed. A probability density function of the frame size distribution could be used as the basis for an admission control algorithm that provides performance guarantees both deterministically and stochastically.

Management of Requests for Admission

Presently, the CMFS admission control process does not provide any assistance to the clients that are denied access to the file system. A client that is rejected receives no information on which to base its future actions, such as retrying or abandoning the open request.

The admission control process could assist the rejected client by providing the following facilities:

- an estimate for the time when it may be reasonable for the client to retry the request
- a call-back mechanism that is used to inform the client about an admission opening
- queuing of the request until an opening is created
- negotiation of the performance requirements.
If admission control were enhanced with some of these facilities, client applications would have more flexibility and control in using the file system.

**Extended CMFS Functionality**

One of the goals for the design of CMFS was the ability to provide useful abstractions for multimedia and video based application. The VCR application, described earlier in the thesis, was developed primarily to gain initial experience with CMFS and to test the performance of the prototype; it was not intended as a comprehensive test for the utility of services provided by CMFS.

Sophisticated video applications are needed for further experimentation with CMFS and its ability to support the set of features commonly associated with digital video [LIP91]. Specifically, it is necessary to explore the requirements of scalable video objects and editing of compressed frames. It is likely that the examination of these requirements will uncover the need for new services and functionality that are not currently supported by CMFS.
Acknowledgments

I would like to express my sincere gratitude to the following people:

To Andrew Black, my company advisor, for undertaking the responsibility of supervising my thesis project in addition to his regular workload at Digital, for providing guidance throughout the project, and for his careful reviewing of this thesis which greatly enhanced both its style and content.

To professor David Tennenhose, my MIT advisor, for suggesting the problem, for taking the risk of supervising an off-campus thesis, for challenging me to look deeper at relevant issues and for substantially improving the structure of the thesis.

To Victor Vyssotsky, director of CRL, for sponsoring the project, and for providing insightful comments on the statistical analysis section of the thesis.

To Rick Szeliski for his help in digitizing video sequences, for discussions on digital image processing, and the willingness to contribute the resources of his workstation to this project.

Special thanks to Ted Wojcik for his interest in the project and outstanding technical support. It would be hard to imagine the success of this project without Ted’s resourcefulness.

To Alec Wolman, Win Treese, and Neil Fishman for their help during the prototype implementation.

To Larry Stewart and Tom Livergood for help with the Lo-Fi and for numerous discussions on real-time and synchronization.

To Digital Equipment Corporation for sponsoring my graduate study.

And, of course, to my family.

To my wife Cynthia for loving me even in my less amicable moments and for always having faith in me.

To my parents, who chose a life of immigration to give me the opportunities that only America could provide.
Bibliography


ARMB86. Heinrich Armbruster, "Applications of Future Broad-Band Services in the Office and Home", IEEE JSAC vol. sac-4, #4 July 1986


CCITT88. "Draft Recommendation I.121", CCITT, SG XVIII TD#49, 1988


DEC90. "DECstation and DECSYSTEM 5000 model 200 Technical Overview", Digital Equipment Corp. Order number: EC-N0854 51


KIM86. Michelle Y. Kim, "Synchronous Disk Interleaving", IEEE Trans. on Comp. vol C-35 #11, Nov. 1986

KIM87. Michelle Y. Kim, "Asynchronous Disk Interleaving", IBM technical report, RC 12497 (#56190) 219187


MAG87. B. Maglaris, etal, "Performance Analysis of Statistical Multiplexing for Packet Video Sources", in Proceed. IEEE Globecom 87, Tokyo, Nov. 1987


MUL91. K. Muller and J. Pasquale, "A High-Performance Multi-Structured File System Design", accepted for publication by SOSF.


SVO84. Liba Svobodova, "File Servers for Network Based Distributed Systems", ACM Comp. Surv. 16,4 Dec. 1984


VIN91. Harrick M. Vin, P. Venkat Rangan, "Designing File Systems for Digital Video and Audio", to be published by SOSP,

WIL91. Ian Wilson, Personal Communication