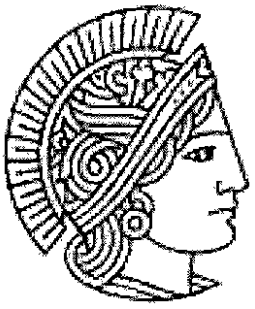
GSt89

Darmstadt University of Technology



Mediaservers

Carsten Griwodz, Ralf Steinmetz

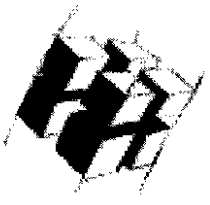
11 June 1998

Technical Report TR-KOM-1998-08

Industrial Process and System Communications (KOM)

Department of Electrical Engineering & Information Technology
Merckstraße 25 • D-64283 Darmstadt • Germany

Phone: +49 6151 166150
Fax: +49 6151 166152
Email: info@KOM.tu-darmstadt.de
URL: http://www.kom.e-technik.tu-darmstadt.de/



Mediaservers

Carsten Griwodz[1], *Ralf Steinmetz*[1,2]

1
Industrial Process and System Communications
Dept. of Electrical Engineering & Information Technology
Darmstadt University of Technology
Merckstr. 25 • D-64283 Darmstadt • Germany

2
GMD IPSI
German National Research Center
for Information Technology
Dolivostr. 15 • D-64293 Darmstadt • Germany

{Carsten.Griwodz, Ralf.Steinmetz}@kom.tu-darmstadt.de

1 Introduction

Media servers are a special variation of file servers with the requirement to deliver part of all of their services within a certain time-frame. This basic requirement can be addressed at a variety of hardware and software levels that comprise a media server. Consequently, the range of research issues that contribute to media server design is wide. While many research groups deal with multimedia servers as a database issue, this chapter of the book concentrates on multimedia servers' content storage and movement and does not consider its management.

A basic, application-specifc distinction is made in these design of such media servers: data retrieval can either be controlled strictly by the client, which requests and sends pieces of content files, or a client can tune in to a server-controlled sending of data, which might have been initiated by that client. Figure 1 demonstrates the request/response behaviour of both approaches.

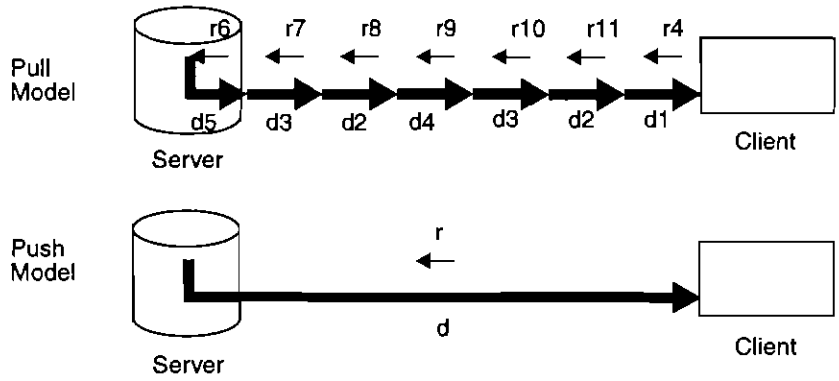


Figure 1: Pull and push server models

A media server that is operated in the first mode is called a *pull server*, a server that is operated in the second mode is call a *push server*. Another, frequently used expression for the push server is the term *data pump*, as this characterizes in a simple way its specialization in retrieving data from disk and delivering it to the network efficiently. Pull servers are surely the more appropriate choice for editing multimedia content in a LAN environment: linear retrieval is frequent but not the rule, pieces of content are rearranged, temporal and spatial cross-connections are introduced. Push servers are the obvious choice for broadcast or multicast distribution of content over wide areas, with no or infrequent user interaction. Applications that are not as clear-cut in there requirements may be solvable with either of the two approaches.

Pull and push servers are often considered competing concepts. Media server implementations, however, show that these worlds are not far apart from each other because major parts of a server can be

operated in modes that can be used in pull as well as push mode. A recently implemented mixture of these approaches is the definition of play lists, client-defined lists that refer to pieces of content which is stored on the server; these play lists are supposed to be sent to the client in a sequence [RTSP]. In the following, consider do not seperate these two approaches any more.

2 Architectures

Media servers are responsible for the timely delivery of content to an end-system. To achieve this goal, each component of the media server must conform to the bounds of time and space to fulfil its tasks. This attracts the research in a variety of areas: disk layout strategies, disk scheduling, file systems, data placement, memory management or CPU scheduling. Figure 2 shows the order in which media server components are involved in delivering the content. Some of the tasks that are seperated in that figure are historically implemented in a single system component.

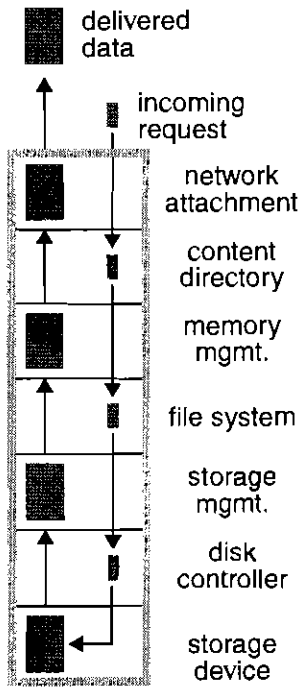


Figure 2: Media server architecture

The *network attachment* is typically a network adapter or a similar device that connects the media server to the customers. The *content directory* is the entity responsible for verifying whether content is available on the media server and whether the requesting client is allowed to access the data. The *memory management* is a separate entity because although a typical content file of multimedia applications is too large to be kept in the main memory for a long time, the caching of content data in main memory improves the performance considerably for some applications. The *file system* handles all information concerning the organization of the content on the media server. This includes such issues as the assignment of sufficient storage space during the upload phase, probably the transparent segmentation of the content file, the consistency of the data on disk, and the location of the elements of a segmented content file during retrieval operations. The *storage management* is the abstraction of driver implementations that communicate directly with the disk controller. The storage management is concerned with disk scheduling policies and the layout of files. The *disk controller* handles the access to data on the *storage device*. Research on the disk controller level includes the increase of head movement speed, I/O bandwidth, the largest and smallest units that can be read at a time and the granularity of addressing.

Of course, optimizing one of the components is not sufficient. The components must cooperate correctly even when the system grows. Such a growth means that the system or some of its components will be replaced or extended. In many cases an extensions means that a task is distributed onto multiple components, probably onto heterogeneous components, and that it may become necessary to replicate part of the data to access it from all components of the distributed system. [TF95] provides a formalization of the options for distributing parts of a video server. This formalization deviates from the reality with the generalization of the content directory's position in a distributed system. Obviously, the content directory must always can consistent and all-knowing in order to answer requests correctly. Figure 3 demonstrates the two alternative approaches to generalizing component distribution while a consistent content directory is maintained. Figure 3 (a) uses an internal content directory which, for consistancy reasons, can exist only once per media server. However, although the content directory appears consistent to all other components, it may still be distributed internally and achieve the appearance of a single component by presenting the same interface on all nodes of the media server. Figure 3 (b) shows all options for distributing components when the approach of an external content directory is adopted. A client of such a system contacts the external content server first to identify itself and to issue the request. After that initial request, two alternatives for proceeding with the retrieval operation are possible. If the response of the content server is returned to the client and the client is responsible for issuing the actual request for data in another call (Figure 4 (a)), additional security mechanisms must be applied because authentication of the client is checked by the content server. Alternatively, the content server can accept all requests directed to the media server, but instead of answering itself, it can immediately order the

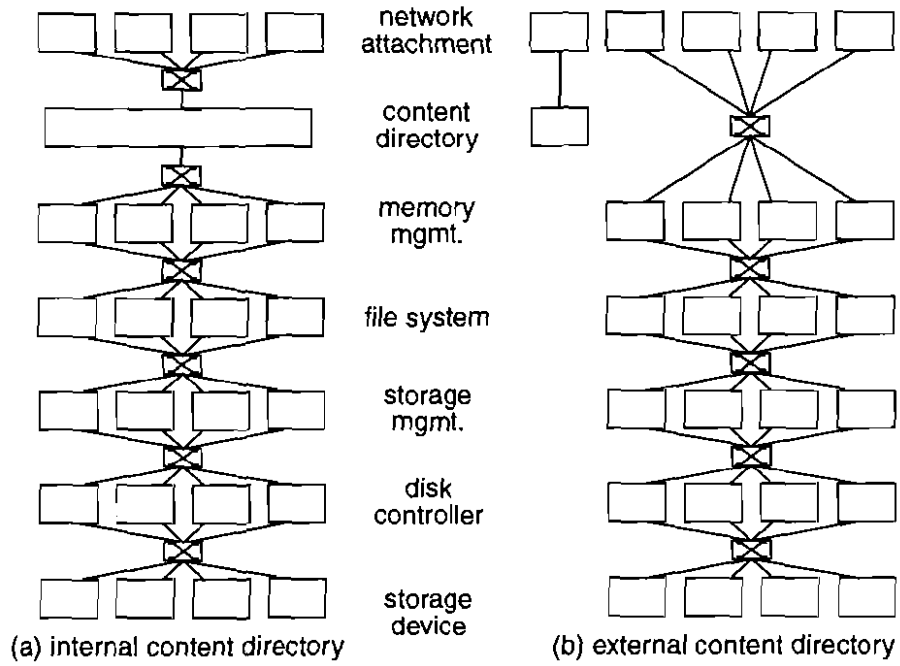


Figure 3: Media server's distribution options

appropriate nodes of the media server to deliver the content data (Figure 4 (b)). This approach is restricted because it requires one of two things: either the client must be able to receive the content data from a different server than the target of its request, or each server node must deliver the content using the address of the content server.

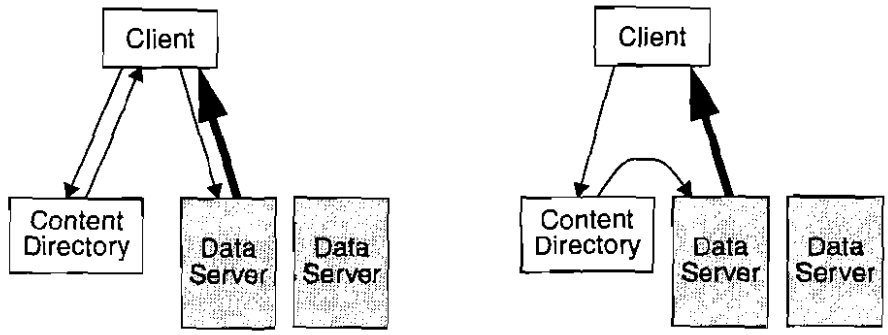


Figure 4: External content retrieval options

Content directories are typically handled by means of multimedia databases which are not subject of this paper.

3 Storage Devices

The storage subsystem is a major component of any information system. Due to the immense storage space requirements of continuous media, conventional magnetic storage devices are often not sufficient. Tapes, still in use in some traditional systems, are inadequate for multimedia systems because they cannot provide independently accessible streams, and random access is slow and expensive.

3.1 Disk Layout

The *layout* of disks determines the way in which content on a disk is addressed, how much storage space on the media is actually addressable and usable und the density of stored content on the media. This has a major influence on the speed of read and write operations on that disk as well as on the capacity of the disk.

Since disks are typically used as random-access media, it would be inefficient to organize data in a single track -a single spiral of data- as it has been the case for CD-Roms until just recently. The single-track technique requires all accessible information to be recorded in terms of distance from the track start, and an access mechanism requires a translation of this number to the position that the read head of the disk has to assume, which must be expressed as a combination of distance from the center or the edge of the disk, to which the head must be moved, and the angular distance, which requires a partial rotation of the disk below the head before the data can be accessed. An additional drawback is the complex handling of operations on files such as delete or append operations. Due to the serial nature of the single-track approach, deletion of files leaves empty space in the track that can hardly be filled with an identically-sized new file. Similarly, data written in append operations is probably located far apart from the original pieces of the file. Segmentation of the medium and a continuous degradation of read- and write-performance are the result.

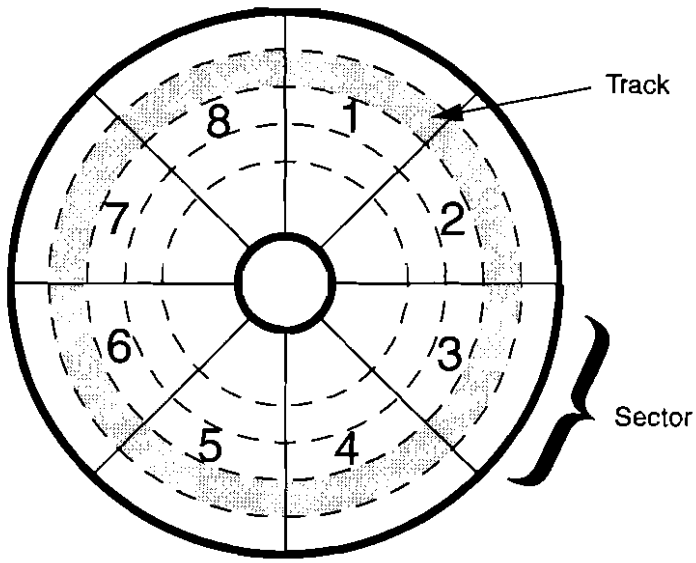


Figure 5: Tracks and Sectors

The location is more easily expressed by partitioning the medium in tracks and sector as depicted in Figure 5. The granularity of disk access is restricted to one sector on one track. The advantage of this scheme is the easy mapping of location information to head movement and disk rotation, and with this scheme the disk can also hide defective parts of the media by reassigning tracks to spare regions of the medium. This scheme has a disadvantage as it looses storage space. When files are generally much smaller than the chosen sector size, large parts of the sector remain unused unless data is appended to the file. If such an append operation is executed, however, it can be handled efficiently.

Additionally, constant rotation speed as well as constant recording and reading speed is typical for both single-track and multi-track disk layout schemes. This does not take into account the fact that the storage capacities of the medium are the same for identically-sized areas in the inner and the outer regions of the disk. Since both rotation and recording speed are kept constant, a sector in track 1 holds the same amount of data as a sector in track 200 although the capacity of the area covered by the sector in track 200 is twice the capacity of the area covered by track 1. Early microcomputers' floppy disk drives addressed this issue by varying sector sizes or by variable rotation speed. For disk drives, neither approach is followed.

3.2 Zone Bit Recording

The current approach to overcome this is Zone Bit Recording [REF]. It makes an approach on the recovery of some of the lost space while both sector sizes and rotation speed are kept constant. Figure 6 shows a sketch of the distribution of sectors on a disk when ZBR is used. The fact that the rotation speed remains constant while more equally-sized sectors are present in the outer tracks of the disk are addressed by a variable reading and writing speed of the disk. The figure is slightly misleading because it hides the fact that zoned disks do not have different numbers of sectors for each track, but only a small number of zones with different layouts.

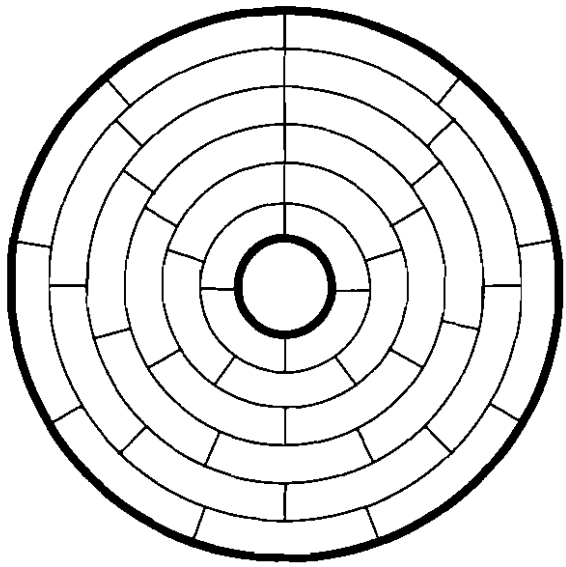


Figure 6: Sector arrangement on a Zoned disk

For the typical use of disks for the storage of discrete content, ZBR disks have the advantage of using the physical medium on the disk more efficiently. Access to data on the outer tracks remains equally fast as access to content on the inner tracks, and since sector sizes remain constant, no additional complexity is visible. When the disks are used for the delivery of continuous media streams such as video, additional considerations are necessary.

Assuming that the disk holds videos of various popularities [BGW97], the movement of popular video files to the outer tracks can reduce the average seek time when multiple users retrieve videos and thus, increase the number of video streams that can be delivered concurrently. The reason for this is that an outer track is read as fast as an inner track but that the amount of sectors of an outer track is higher than that of an inner track.

In [KLC97], the allocation of complete tracks to videos in this way is proposed. This allows for the fast transfer of frequently requested data from the outer zones to main memory buffers and it is consistent with the approach of storing video contiguously on disk to reduce seek times. However, from these buffers, a continuous playout must be guaranteed and thus, this video data must be kept in memory until

it is delivered to the network. This raises the question whether this approach wastes buffers space. The more relevant effect of storing blocks of popular videos in outer tracks is gained in conjunction with algorithms such as SCAN or SCAN-EDF (see Section 6). Since most data of popular videos is stored in outer tracks, the probability that data must be retrieve from inner tracks as well as the average distance of disk head movements is reduced.

Alternatively, *track pairing* is proposed in [BY95]. In this approach, two consecutive parts of a video data are stored first on an outer and than an inner track. A pair of outer and inner tracks forms a logical track, where all logical tracks on a disk have identical average throughput. In [CT97], this approach is modified to *segment group pairing* scheme, in which identically-sized I/O units of a video are stored in outer and inner zones of the ZBR disk in such a way that the average throughput of a pair of groups remains constant.

The variable block size scheme, or *VARB* scheme introduced in [GKS95] is another scheme to exploit zoned disks. The main difference is the ignorance of disk sector size in favour of self-defined blocks. Blocks of a content file are stored on disk in a round robin manner in such a way that the time for reading a block is always the same. This implies that blocks are larger in the outer zones, where segments are read faster than in the inner zones. As a result of this arrangement, assuming that data has to be delivered to the client in a constant bit-rate manner, the time between retrieval operations on a single stream become variable. The fixed block size scheme, or *FIXB*, was also introduced in [GKS95]. It is a scheme for use with zoned disks but in this case, blocks of the same size are stored in all zones.

3.3 File Structure

We commonly distinguish two methods of file organization. In sequential storage, each file is organized as a simple sequence of bytes or records. Files are stored consecutively on the secondary storage media as shown in Figure 7. They are separated from each other by a well defined "end of file" bit pattern,

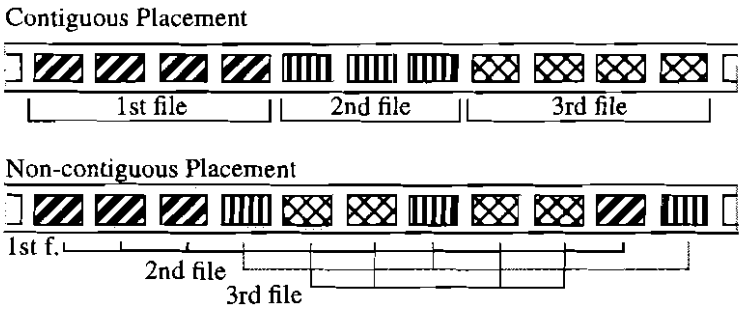


Figure 7: Contiguous and non-contiguous storage

character or character sequence. A file descriptor is usually placed at the beginning of the file and is, in some systems, repeated at the end of the file. Sequential storage is the only possible way to organize the storage on tape, but it can also be used on disks. The main advantage is its efficiency for sequential access, as well as for direct access [Kra88]. Disk access time for reading and writing is minimized.

With multimedia data, neither contiguous placement nor random placement of disk blocks is an optimal solution. Contiguous placement can be implemented easily but has the drawbacks of creating large empty arrays and thus, fragmentation on the disk. Insertion and deletion operations are extremely costly when data is moved to keep continuity and defragment the disk. This makes contiguous placement unaffordable for media servers that are also used in editing or frequent upload operations. Random placement, in contrast, implies that random seeks from one file block to the next must be made very often, even when only a small amount of data is required.

A few approaches address this problem. One approach is the selection of large block sizes. Since continuous media content is typically large, the percentage of lost space due to partially unused blocks at the end of the file is acceptable. The media server takes advantage of this by the reduced management for all operations because less addressing information needs to be kept, and because big amounts of data can be transferred to the main memory without seek penalties. In [RW94], Reddy and Wyllie introduce *constrained placement* (at the disk controller or storage management level), which introduces the technique of placing blocks on disk in such a way that they are in a reasonable distance from each other, meaning that the seek-time between a block and its consecutive block is within acceptable bounds.

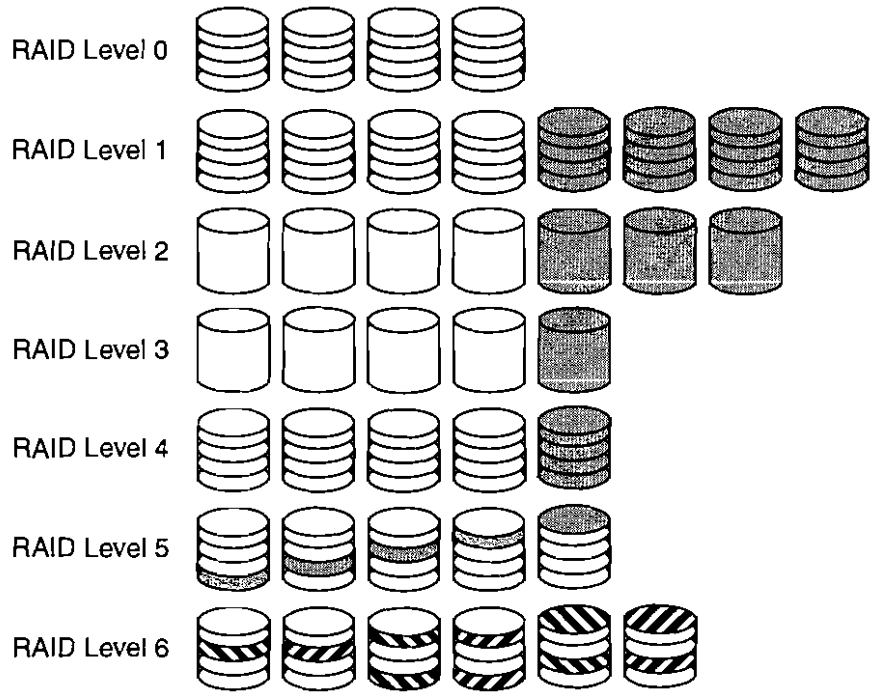


Figure 8: RAID levels ([CLG+94])

4 Disk Controller

4.1 Data Placement

Striping

If a system grows large enough to make the usage of multiple disks affordable, an issue that concerns many home-used PC nowadays, the means of accessing these disks must be taken into account. The simple approach is to arrange file systems in a convenient way on either of the disks and gain performance by put the more frequently used data on the fastest disks. Although this is sufficient if the amount of disk space is the only concern, this is not the most effective way in which multiple disks can be used. Striping techniques have been developed that take more than just the amount of available space into account.

RAID

The original way of combining disks in a more efficient way is the "Redundant Arrays of Inexpensive Disks", or RAID ([PGK88]). RAID addresses both performance and security problems to various extents in its various sub-specifications, which are called RAID-Levels. Seven RAID levels are defined (0-6), each of which makes a different approach at combining performance enhancement with security enhancements. Some of these levels can be implemented in software while others require hardware support.

- A RAID-0 disk array is nonredundant, so basically, the name RAID is even misleading in this case. It is a purely performance-oriented RAID-level. It does not introduce any redundancy (or security) into the system but allows file systems to spread out across multiple disks, e.g. to achieve higher throughput for the delivery of data to applications that can consume data quickly but are throttled by the read-performance of a single disk.

- RAID-1 implements mirroring or shadowing by storing all data twice. This is the traditional approach to data security. Whenever data is written to the RAID system, each block is stored on two disks that are mirrors of each other. When data is retrieved, this system can be used to retrieve data from the disk with the lower access delay. In case of a disk failure, all requests are handled by the mirror of that disk. The recreation of a mirror disk after a failure is a copy operation from the remaining disk. The storage efficiency is low in this level, but the number of read transactions is high.
- RAID-2 implements error correcting codes similar to ECC memory. For all of the primary data disks, the Hamming codes ([PW72]) are computed and stored on additional parity disks. When a disk fails, the parity information on multiple parity disks identifies correctly the failed disk and in conjunction with all remaining data disks, the data of the failed disk can be reconstructed with a single parity disk. Since parity information must be modified on multiple parity disks at each write operation, this is a rather expensive RAID level and must be implemented in hardware.
- RAID-3 implements bit-interleaved parity, which requires only a single parity disk. In this scheme, the fact is exploited that the disk controller can easily detect when a disk fails. Thus, the identification of the failed disk that is possible with level 2 is not needed and only the recreation of lost information remains an issue. This recreation is possible after the failure of one data disk by computing the sum of each bit on all of the remaining data disks and the parity disk modulo 2. As in level 2, the constant maintenance of this parity information restricts the write performance to that of the parity disk and requires hardware support for the parity calculation.
- RAID-4 is named block-interleaved parity. Instead of looking at each bit individually, the term *striping unit* is introduced for this RAID level. A stripe extends across all data disks of the disk array and is composed of data blocks of arbitrary size, the striping unit, on each disk. If a write operation is smaller than the striping unit, all data is written to one disk, otherwise striping units of more disks are modified. A block of new parity data on the parity disk is than computed from all striping units in the affected stripe. As with level 3, this RAID level is bound by the performance of the single parity disk that is affected by all write operations.
- RAID-5 reduces this performance bottleneck by implementing block-interleaved distributed-parity. Instead of keeping the parity information of a stripe on a specific parity disk, parity blocks are equally distributed over the disks in the stripe units. The decision for placement of these parity blocks affects the performance of the system, as shown in [LK91]. In case of a disk failure, the missing data can be reconstructed as in level 4, without any additional consideration whether the reconstructed data is original data and parity data.
- RAID-6, called P+Q redundancy, uses Reed-Solomon codes to protect against the failure of two disks by increasing the basic size of the array by two redundant disks.These codes provide a better means of reconstructing the original data in case of a disk failure. This is relevant because the parity-protected levels all require the no read fails until the failed disk has been replaced. However, in large installations, additional errors become more probable, and protections against this are neceassry. Level 6 provides this protection for a failure of up to two complete disks by distribution redundant data in a similar way to level 5.

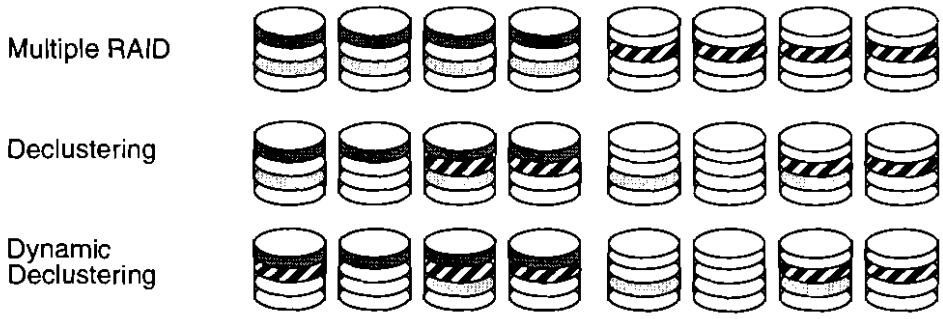


Figure 10: Varieties of group creation

The performance of the various RAID levels is shown in Table 1.

Table 1: Throughput and storage efficiency of RAID levels
The efficiency is relative to RAID 0, G is the number of disks in a group.

RAID level	small read	small write	large read	large write	space usage
0	1	1	1	1	1
1	1	1/2	1	1/2	1/2
2	(G+1)/2	(G+1)/2	(G+1)/2	(G+1)/2	(G+1)/2G
3	1/G	1/G	(G-1)/G	(G-1)/G	(G-1)/G
4	1/G	1/G	(G-1)/G	(G-1)/G	(G-1)/G
5	1	max(1/G,1/4)	1	(G-1)/G	(G-1)/G
6	1	max(1/G,1/6)	1	(G-2)/G	(G-2)/G

RAID was not developed to support multimedia applications, although the higher throughput of striped disks is an asset in that case. For the scalability of Multimedia Server, however, the throughput is only one issue among many. An increase in the scale of throughput is typically not necessary for a MM Server to handle new data formats and to deliver that data as quickly as possible to a client. Rather, the number of clients that are requesting data concurrently increases, which increases not only the amount of data that has to be delivered but also the number of files that have to be retrieved in parallel. This implies an increased number of seek operations per unit time, a scaling issue which is not covered by RAID technology. Similarly, since multimedia data requires time-conforming delivery of data streams, the buffer requirements at the server grow when disk throughput is considerably higher than the delivery rate. A buffer allocated for each single client is filled in a short read burst due to the high throughput of the parallel disks, and subsequently, this data is delivered at the requested rate from that buffer. With an increasing number of disks to support more clients, the larger data blocks delivered per read burst as well as the increased number of parallel streams contribute to a quick increase in buffer requirements. Figure 9 illustrates the increase of buffer sizes that have to be made available to buffer a single retrieval operation when the size of a RAID stripe group grows. A variety of techniques have been developed to address also this issue.

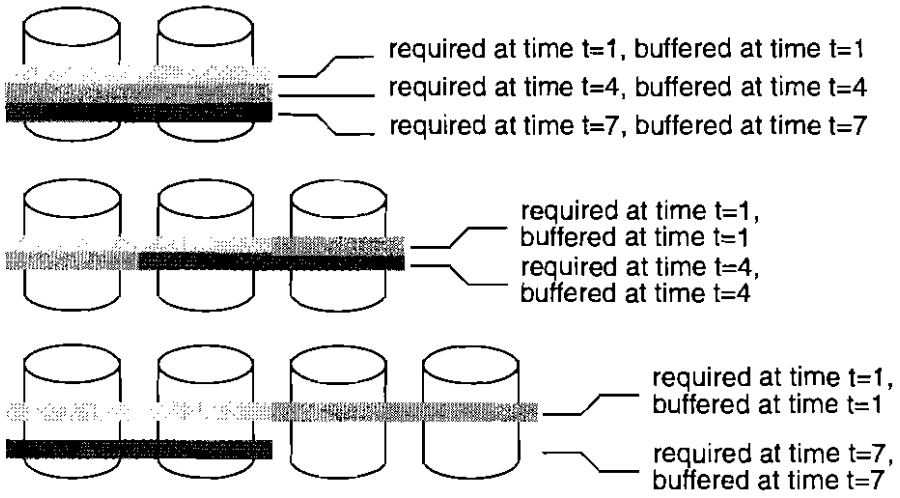


Figure 9: Growth of buffer requirements

Multiple RAID

The most intuitive of these techniques is the creation of subgroups of disks into independant logical disk arrays. This limits the number of disks across which a file is striped to the size of such a group.

Declustering

In declustering, groups are not made up from complete disks. Rather than this, the stripe units of each disk are considered. Stripe units are logically connected into a stripe that spans only a subset of the disks (using the typical RAID protection mechanism). The number of disks for any such stripe is fixed and the same, but the disks on which a stripe is located differs. In such a way, all load is better distributed then with Multiple RAID and the I/O throughput of all disks is exploited even if only a limited number of stripes is accessed.

Dynamic Declustering

In dynamic declustering, this scheme is extended to assign stripes not statically to a set of disks but rather, decide for each file the size and location of the stripes used. This scheme has two drawbacks: it a very management-intensive and it can not be used with protection mechanisms. It is a lot more management-intensive because a selection of a stripe must be made for each write operation. This selection must be augmented by the application because the file system disk controller is not ware of the throughput requirements of a content file.However, this scheme allows for an adaptation of the buffer size to the bandwidth requirements of the content file. When a server is supposed to deliver a mixed workload which ranges from bulk data to various continuous media formats, this can be worthwhile. A second drawback is that the group assignment in software makes it difficult to compute parity information in the disk controller. Special hardware and special interfaces would be necessary.

Weighted Striping

Weighted striping ([WD97]) takes into account that it is unlikely for a real-world system to operate a multimedia server for a long time without adding or replacing part of the disks with newer or cheaper models. Since this introduces an inhomogenity in the performance of the disks in the system, the throughput of a stripe may be limited by the least performant disk. In the *variable size weighted striping*, the size of stripe units are varied depending on the throughput of each disk in the stripe group.

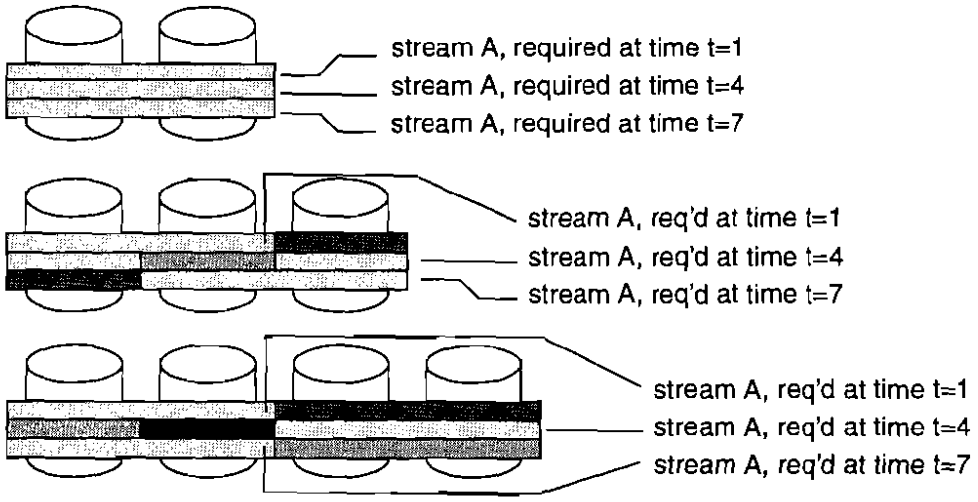


Figure 11: Split Stripe retrieval

However, the replacement of a single disks with a new disk, with different performance characteristics, requires a memory-intensive restriping of a all data bytes of the stripe rather than the reconstruction of a single stripe unit. Because of this, the *constant size weighted striping* is also proposed, which intends to level the throughput demands on single disks in the stripe in the long term. A video is split into units as large as one stripe unit, and these units are distributed onto the disks in such a way that disks with a higher throughput hold more units than disks with a lower throughput. The number of assigned units is equivalent to the throughput of the disks.

Split-Stripe Retrieval

Introduced in [TF95], this technique tries to address the problem of buffer requirements by allowing read operations for more than one stream in a single read operation to a stripe, i.e. while a full stripe is read in every cycle, the results of this read operation do not necessarily fill only the buffer for a single stream. Figure 11 gives a sketch of the retrieval operations. While the smallest addressable unit in RAID is the stripe. split stripe retrieval requires the addressing of the stripe unit, which may be as small as a single sector on a single disk.

Cyclic Retrieval

An enhancement of this technique is to allow not only the addressing in a read operation to be independent of the stripe unit that is read with the stripe but to perform read operations on stripe units without the need for retrieving full stripes in one operation. This cyclic retrieval technique allows for a much smaller buffers per stream because the maximum required buffer size per stream is not the size of a stripe but that of a single stripe unit and not a full stripe.

4.2 Reorganization

The addition of new disks to a media server can result in an overall performance increase for that server if the I/O bandwidth of those new disks can be exploited. For quite a while now, hot-swappable and hot-pluggable disks are available from hardware vendors. Since the newly added disks can be considered initially empty, a reorganization of content that is already located on the server must be iniatiated in that case. In the best case, such a reorganization is handled without disrupting the service.

One scheme for reorganizing a media server after the addition of a disk $(n+1)$ to a stripe group $(1 .. n)$ without disrupting the service is the movement of segments of a content file to rearrange them in such a way that consecutive segments are ordered according to the index of the disk in the stripe group (mod-

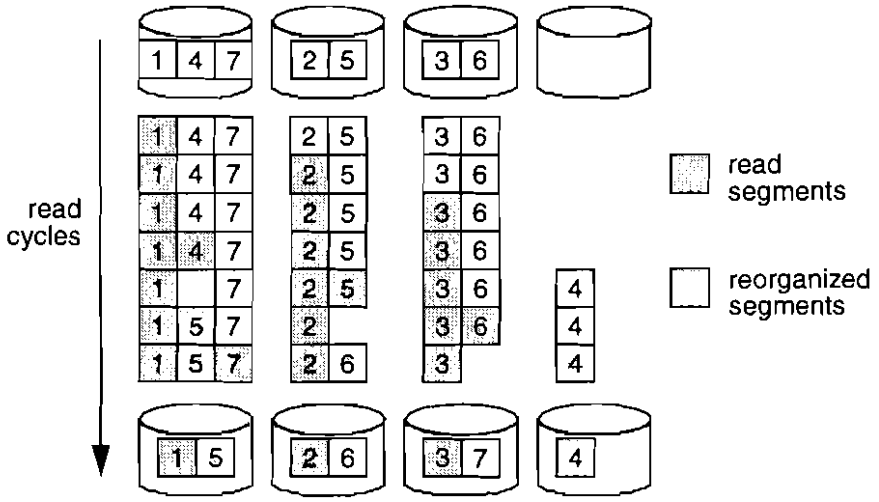


Figure 12: Lazy reorganization of stripe groups

ulo *n+1*). [GK96] presents *lazy* and *eager* on-line reorganization. Lazy reorganization is activated only when a content file is retrieved by a client. If a segment of the content file is retrieved which should be relocated to the new disk based the placement formula for segments in stripe groups, a write operation to the new disk is performed in the cycle following the read operation. Figure 12 visualizes the rearrangement. This scheme has the drawback that the reorganization is executed only for those content files that are accessed by a client. If a file is never retrieved, it is also never reorganized. With eager reorganization, idle time of the system is used to reorganize content. Since the order to segments in this approach is not bound by the playout order, multiple segments can be rearranged at a time, if sufficient buffer space in main memory is available. Assuming sufficient buffer space, the reorganization would be performed as shown in Figure 13. The time that is necessary to perform the reorganization can fur-

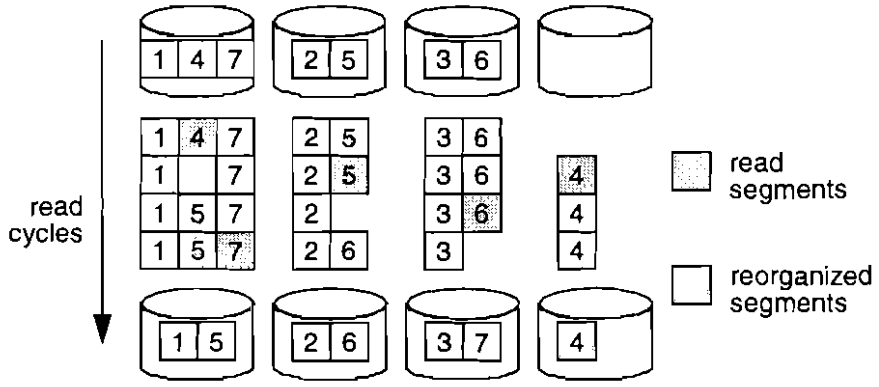


Figure 13: Eager reorganization of stripe groups

ther be reduced by preloading the segments that are targetted at the new disks before the insertion of those disks into the system. The original segments are then removed from their previous location and the other blocks are reorganized.

The *random duplicated assignment* approach proposed in [Kor97] is based on the assumption of a system which is built on a growing number of heterogenous disks and delivers variable bit-rate data streams. Such a system is on average not degraded by randomly placing blocks two times on different disks of the system and retrieving them from the less loaded disk. In case of replacement of a single disk, content can be reconstructed by replicating data blocks onto that disk when only a single copy of these blocks is available in the system, in case of adding a new disk, blocks can be from any of the disks

that are already available in the system and they can be moved to the new disk in case that they have not been moved there from a different disk earlier.

5 Storage Management

Whereas strictly sequential storage devices (e.g., tapes) do not have a scheduling problem, for random access storage devices, every file operation may require movements of the read/write head. This operation, known as "to seek", is very time consuming. Disk management tries to reduce the effects of such operations. Still, the actual time to read or write a disk block is determined by:

- The seek time (the time required for the movement of the read/write head).
- The latency time or rotational delay (the time during which the transfer cannot proceed until the right block or sector rotates under the read/write head).
- The actual data transfer time needed for the data to copy from disk into main memory.

Usually the seek time is the largest factor of the actual transfer time. Most systems try to keep the cost of seeking low by applying special algorithms to the scheduling of disk read/write operations. The access of the storage device is a problem greatly influenced by the file allocation method. For instance, a program reading a contiguously allocated file generates requests which are located close together on a disk. Thus head movement is limited. Linked or indexed files with blocks, which are widely scattered, cause many head movements. In multi-programming systems, where the disk queue may often be non-empty, fairness is also a criterion for scheduling. The approaches to optimize these are called disk scheduling algorithms.

5.1 Disk Management

Disk access is a slow and costly transaction. In traditional systems, a common technique to reduce disk access are *block caches*. Using a block cache, blocks are kept in memory because it is expected that future read or write operations access these data again. Thus, performance is enhanced due to shorter access time. Another way to enhance performance is to reduce disk arm motion. Blocks that are likely to be accessed in sequence are *placed together on one cylinder*. To refine this method, rotational positioning can be taken into account. Consecutive blocks are placed on the same cylinder, but in an interleaved way as shown in Figure 14. Another important issue is the placement of the mapping tables (e.g.,

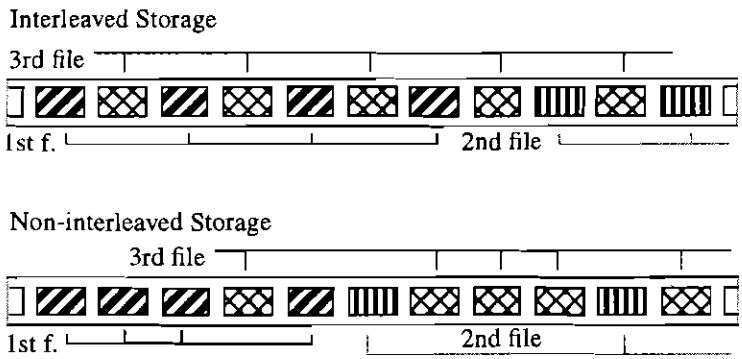


Figure 14: Interleaved and non-interleaved storage

I-nodes in UNIX) on the disk. If they are placed near the beginning of the disk, the distance between them and the blocks will be, on average, half the number of cylinders. To improve this, they can be placed in the middle of the disk. Hence, the average seek time is roughly reduced by a factor of two. In the same way, consecutive blocks should be placed on the same cylinder. The use of the same cylinder for the storage of mapping tables and referred blocks also improves performance.

File Structure

In conventional storage management systems, the main goal of the file organization is to make efficient use of the storage capacity (i.e., to reduce internal and external fragmentation) and to allow arbitrary deletion and extension of files. In multimedia systems, the main goal is to *provide a constant and timely retrieval of data.* Internal fragmentation occurs when blocks of data are not entirely filled. On average, the last block of a file is only half utilized. The use of large blocks leads to a larger waste of storage due to this internal fragmentation. External fragmentation mainly occurs when files are stored in a contiguous way. After the deletion of a file, the gap can only be filled by a file with the same or a smaller size. Therefore, there are usually small fractions between two files that are not used, storage space for continuous media is wasted.

As mentioned above, the goals for multimedia file systems can be achieved through providing enough buffer for each data stream and the employment of disk scheduling algorithms, especially optimized for real-time storage and retrieval of data. The advantage of this approach (where data blocks of single files are scattered) is flexibility. External fragmentation is avoided and the same data can be used by several streams (via references). Even using only one stream might be of advantage; for instance, it is possible to access one block twice, e.g., when a phrase in a sonata is repeated. However, due to the large seek operations during playback, even with optimized disk scheduling, large buffers must be provided to smooth jitter at the data retrieval phase. Therefore, there are also long initial delays at the retrieval of continuous media.

The much greater size of continuous media files and the fact that they will usually be retrieved sequentially because of the nature of the operation performed on them (such as play, pause, fast forward, etc.) are reasons for an optimization of the disk layout. Our own application-related experience has shown that continuous media streams predominantly belong to the write-once-read-many nature, and streams that are recorded at the same time are likely to be played back at the same time (e.g., audio and video of a movie, [LS93]).

5.2 Traditional Disk Scheduling

Most traditional storage systems apply one of the following scheduling algorithms:

First-Come-First-Served (FCFS)

With this algorithm, the disk driver accepts requests one-at-a-time and serves them in incoming order. This is easy to program and an intrinsically fair algorithm. However, it is not optimal with respect to head movement because it does not consider the location of the other queued requests. This results in a high average seek time. Figure 15 shows an example of the application of FCFS to a request of three queued blocks.

Shortest-Seek-Time First (SSTF)

At every point in time, when a data transfer is requested, SSTF selects among all requests the one with the minimum seek time from the current head position. Therefore, the head is moved to the closest track in the request queue. This algorithm was developed to minimize seek time and it is in this sense optimal. SSTF is a modification of Shortest Job First (SJF), and like SJF, it may cause starvation of some requests. Request targets in the middle of the disk will get immediate service at the expense of requests in the innermost and outermost disk areas. Figure 16 demonstrates the operation of the SSTF algorithm.

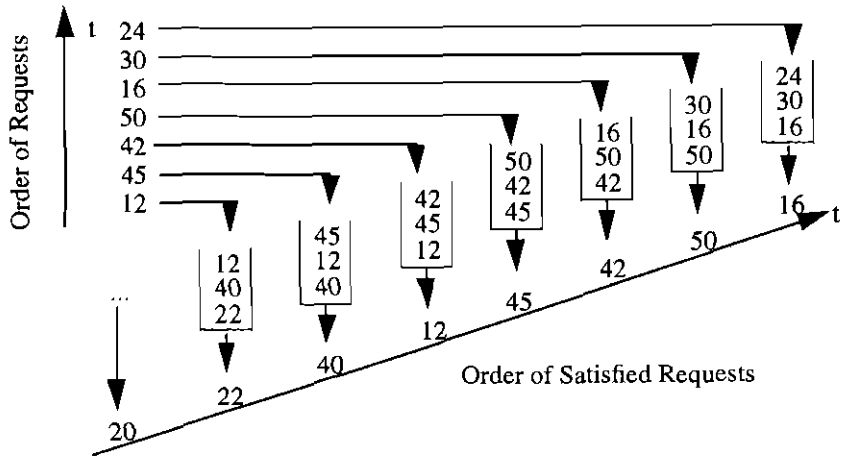


Figure 15: FCFS disk scheduling

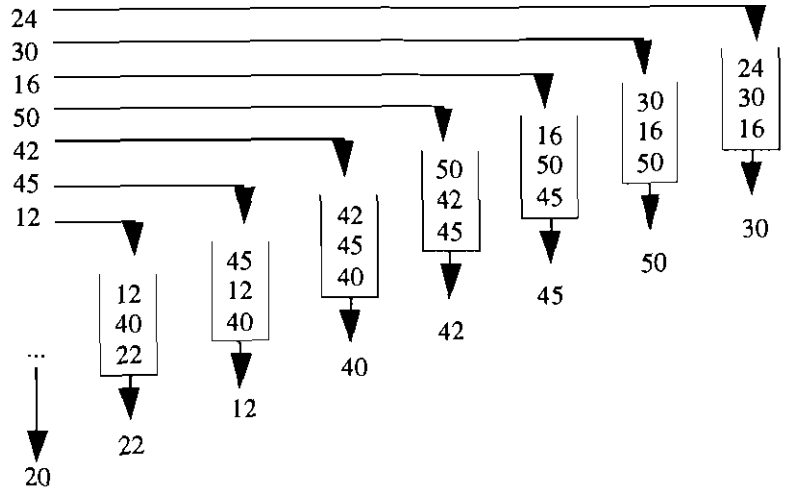


Figure 16: SSTF disk scheduling

SCAN

Like SSTF, SCAN orders requests to minimize seek time. In contrast to SSTF, it takes the direction of the current disk movement into account. It first serves all requests in one direction until it does not have any requests in this direction anymore. The head movement is then reversed and service is continued. SCAN provides a very good seek time because the edge tracks get better service times. Note that middle tracks still get a better service then edge tracks. When the head movement is reversed, it first serves tracks that have recently been serviced, where the heaviest density of requests, assuming a uniform distribution, is at the other end of the disk. Figure 17 shows an example of the SCAN algorithm.

N-Step-SCAN

This variation of the SCAN tries to reduce delays that are introduced into the SCAN scheme by requests that arrive after the SCAN has started. As a result of this, incoming requests may have to wait although the disk head passes by the requested position on the disk. The N-Step-SCAN approach gains fairness for the requests to data on outer tracks for a lower average response time. The scheme can be modified

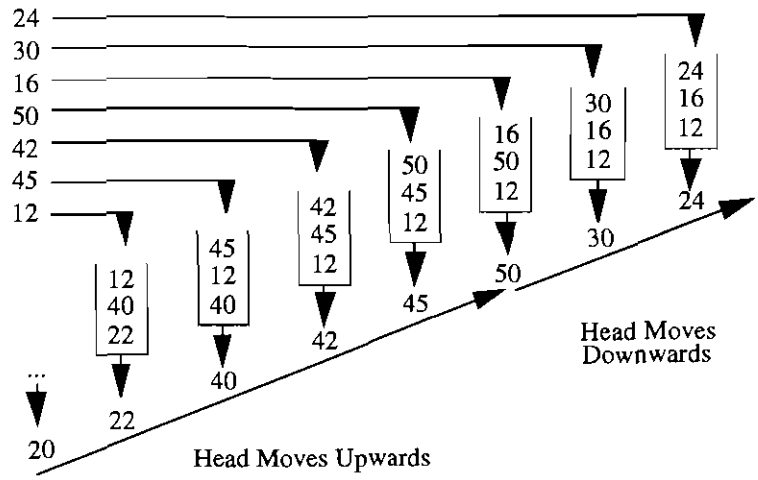


Figure 17: SCAN disk scheduling

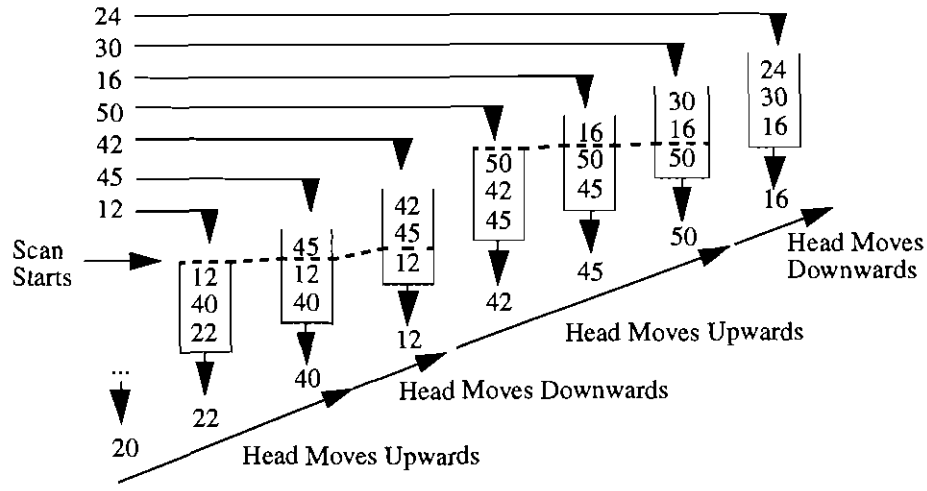


Figure 18: N-step-SCAN disk scheduling

to move the disk head to the outermost position that is requested for the next SCAN instead of starting the next SCAN from the position of the last read track of the previous SCAN. One effect is that SCAN are not always performed in upwards-downwards order, but the direction of the movement can change. Another is that this approach, called Preseek-Sweep-Scheduling [REF] yields a lower average seek times.

C-SCAN

C-SCAN also moves the head in one direction, but it offers fairer service with more uniform waiting times. It does not alter the direction, as in SCAN. Instead, it scans in cycles, always increasing or decreasing, with one idle head movement from one edge to the other between two consecutive scans. The performance of C-SCAN is somewhat less than SCAN. Figure 19 shows the operation of the C-SCAN algorithm.

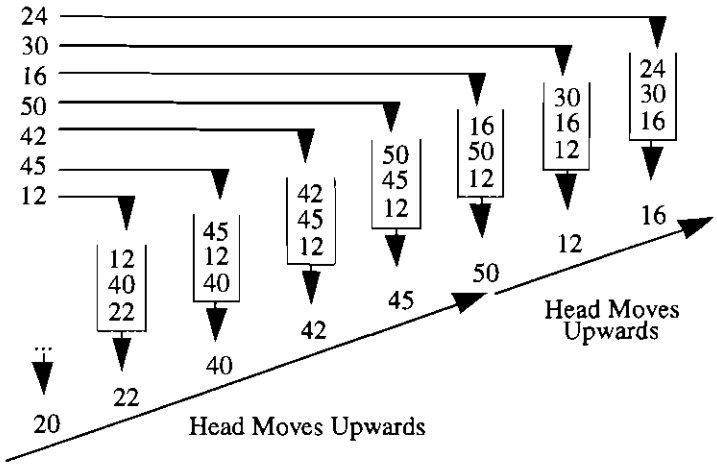


Figure 19: C-SCAN disk scheduling

T-SCAN

T-SCAN, introduced in [REF], uses a method called period transformation to prevent blocking of individual requests. Such a period transformation is actually a modification of the sizes by which individual requests are serviced. With the goal of supporting media streams without blocking, T-SCAN use one stream's request behaviour as a reference to to service all requests. That implies that, if the reference stream's requested rate is R_1 and the block size that is requested per cycle is B_1, and another streams requested rate is R_2, that other stream is serviced with blocks of size of B_1*R_2/R_1, no matter what the actual requests of the application are. In this way, requests are serviced in the order of their arrival using the SCAN mechanism, and all streams will be provided a fair share of the I/O bandwidth. However, without admission control, this scheme affects all serviced streams when the server gets overloaded.

Traditional file systems are not designed for employment in multimedia systems. They do not, for example, consider requirements like real-time which are important to the retrieval of stored audio and video. To serve these requirements, new policies in the structure and organization of files, and in the retrieval of data from the disk, must be applied. The next section outlines the most important developments in this area.

5.3 Multimedia Disk Scheduling

The main goals of traditional disk scheduling algorithms are to reduce the cost of seek operations, to achieve a high throughput and to provide fair disk access for every process. The additional real-time requirements introduced by multimedia systems make traditional disk scheduling algorithms, such as described previously, inconvenient for multimedia systems. Systems without any optimized disk layout for the storage of continuous media depend far more on reliable and efficient disk scheduling algorithms than others. In the case of contiguous storage, scheduling is only needed to serve requests from multiple streams concurrently. In [LS93], a round-robin scheduler is employed that is able to serve hard real-time tasks. Here, additional optimization is provided through the close physical placement of streams that are likely to be accessed together.

The overall goal of disk scheduling in multimedia systems is to meet the deadlines of all time-critical tasks. Closely related is the goal of keeping the necessary buffer space requirements low. As many streams as possible should be served concurrently, but aperiodic requests should also be schedulable without delaying their service for an infinite amount of time. The scheduling algorithm must find a balance between time constraints and efficiency.

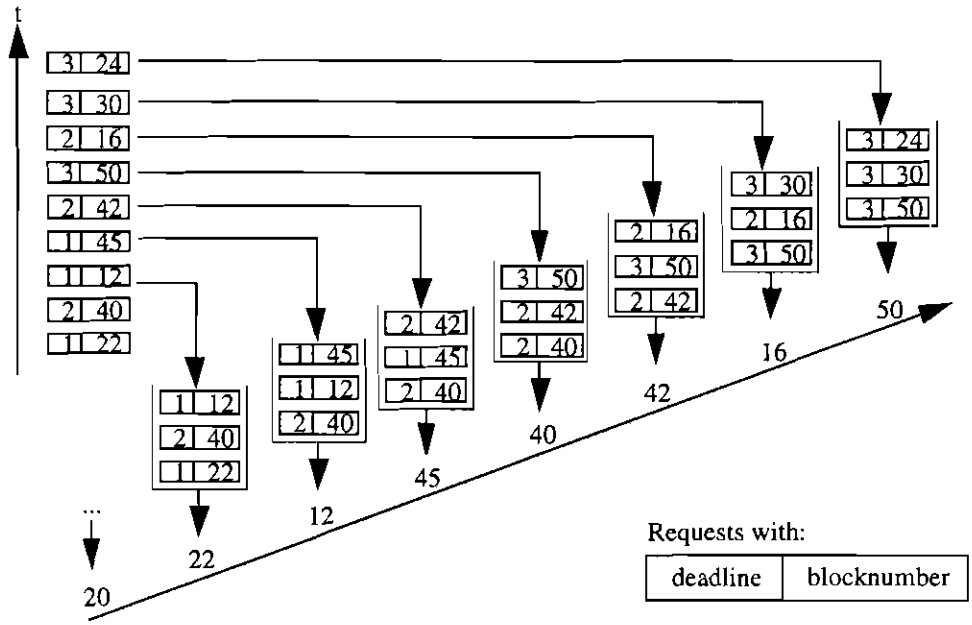


Figure 20: EDF disk scheduling

Earliest Deadline First

Let us first look at the EDF scheduling strategy as described for CPU scheduling, but used for the file system issue as well. Here the block of the stream with the nearest deadline would be read first. The employment of EDF, as shown in Figure 20, in the strict sense results in poor throughput and excessive seek time. Further, as EDF is most often applied as a preemptive scheduling scheme, the costs for preemption of a task and scheduling of another task are considerably high. The overhead caused by this is in the same order of magnitude as at least one disk seek. Hence, EDF must be adapted or combined with file system strategies.

SCAN-Earliest Deadline First

CG: somebody noticed that cycle-based filesystems collapse SCAN-EDF into SCAN. Does a long SCAN-EDF description still make sense?

The SCAN-EDF strategy is a combination of the SCAN and EDF mechanisms [RW93]. The seek optimization of SCAN and the real-time guarantees of EDF are combined in the following way: like in EDF, the request with the earliest deadline is always served first; among requests with the same deadline, the specific one that is first according to the scan direction is served first; among the remaining requests, this principle is repeated until no request with this deadline is left.

Since the optimization only applies for requests with the same deadline, its efficiency depends on how often it can be applied (i.e., how many requests have the same or a similar deadline). To increase this probability, the following tricky technique can be used: all requests have release times that are multiples of the period p. Hence, all requests have deadlines that are multiples of the period p. Therefore, the requests can be grouped together and be served accordingly.

SCAN-EDF can be easily implemented. To do this, EDF must be modified slightly. If D_i is the deadline of task i and N_i is the track position, the deadline can be modified to be $D_i + f(N_i)$. Thus the deadline is deferred. The function $f()$ converts the track number of i into a small perturbation of the deadline, as

shown in the example of Figure 21. It must be small enough so that $D_i + f(N_i) \leq D_j + f(N_j)$ holds for all

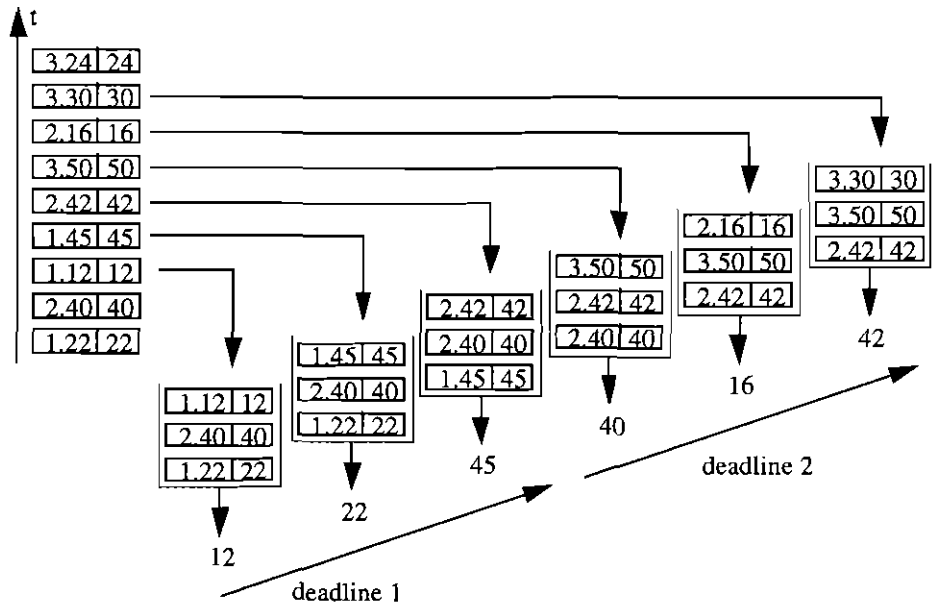


Figure 21: SCAN-EDF disk scheduling with N_{max} = 100 and $f(N_i) = N_i$

$D_i \leq D_j$, it was proposed to choose the following function [RW93]:

$$f(N_i) = \frac{N_i}{N_{max}}$$

where N_{max} is the maximum track number on disk. Other functions might also be appropriate.

We enhanced this mechanism by proposing a more accurate perturbation of the deadline which takes into account the actual position of the head (N). This position is measured in terms of block numbers and the current direction of the head movement (see also Figures 22 and 23):

1. If the head moves toward N_{max}, i.e., upward, then

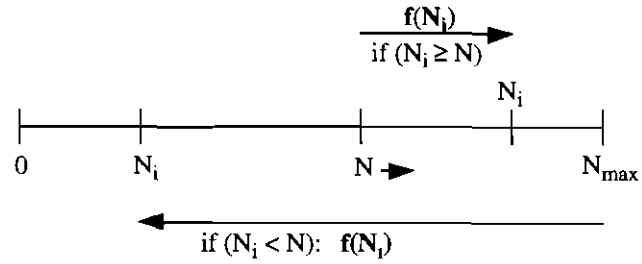


Figure 22: Accurate EDF-SCAN algorithm, head moves upward

(a) for all blocks N_i located between the actual position N and N_{max}, the perturbation of the deadline is:

$$f(N_i) = \frac{N_i - N}{N_{max}} \quad for\ all\ \ N_i \geq N$$

(b) for all blocks N_i located between the actual position and the first block (no. 0):

$$f(N_i) = \frac{N_{max} - N_i}{N_{max}} \quad for\ all\ \ N_i < N$$

2. If the head moves downward towards the first blocks, then

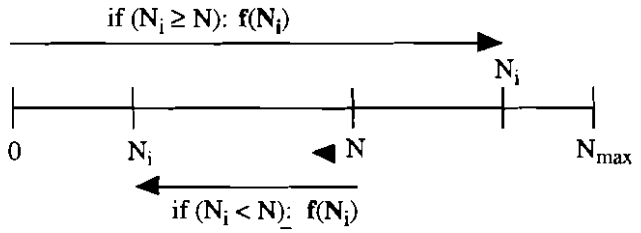


Figure 23: Accurate EDF-SCAN algorithm, head moves downward

(a) for all blocks located between the actual position and N_{max}:

$$f(N_i) = \frac{N_i}{N_{max}} \quad for\ all\ \ N_i > N$$

(b) for all blocks located between this first block with the block number 0 and the actual position:

$$f(N_i) = \frac{N - N_i}{N_{max}} \quad for\ all\ \ N_i \leq N$$

Our algorithm is more computing-intensive than those with the simple calculation of [RW93]. In cases with only a few equal deadlines, our algorithm provides improvements and the expenses of the calculations can be tolerated. In situations with many, i.e., typically more than five equal deadlines, the simple calculation provides sufficient optimization and additional calculations should be avoided. SCAN-EDF was compared with pure EDF and different variations of SCAN. It was shown that SCAN-EDF with deferred deadlines performed well in multimedia environments [RW93].

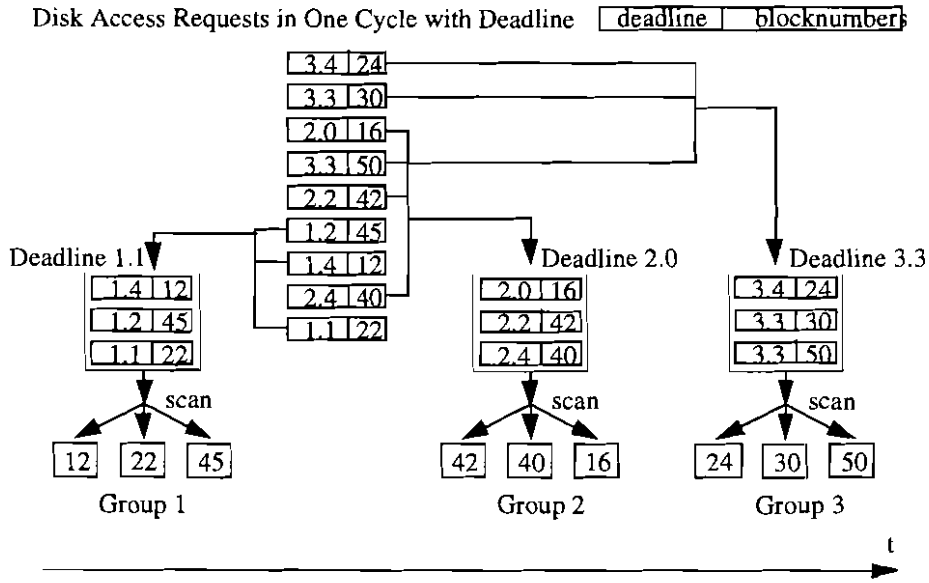


Figure 24: Group sweeping scheduling as a disk access strategy

Group Sweeping Scheduling

With Group Sweeping Scheduling (GSS), requests are served in cycles, in round-robin manner ([YCK92], [GH94]). To reduce disk arm movements, the set of n streams is divided into g groups. Groups are served in fixed order. Individual streams within a group are served according to SCAN; therefore, it is not fixed at which time or order individual streams within a group are served. In one cycle, a specific stream may be the first to be served; in another cycle, it may be the last in the same group. A smoothing buffer which is sized according to the cycle time and data rate of the stream assures continuity. If the SCAN scheduling strategy is applied to all streams of a cycle without any grouping, the playout of a stream cannot be started until the end of the cycle of its first retrieval (where all requests are served once) because the next service may be in the last slot of the following cycle. As the data must be buffered in GSS, the playout can be started at the end of the group in which the first retrieval takes place. Whereas SCAN requires buffers for all streams, in GSS, the buffer can be reused for each group. Further optimizations of this scheme are proposed in [CKY93]. In this method, it is ensured that each stream is served once in each cycle. GSS is a trade-off between the optimization of buffer space and arm movements. To provide the requested guarantees for continuous media data, we propose here to introduce a "joint deadline" mechanism: we assign to each group of streams one deadline, the "joint deadline". This deadline is specified as being the earliest one out of the deadlines of all streams in the respective group. Streams are grouped in such a way that all of them comprise similar deadlines. Figure 24 shows an example of GSS.

Mixed Strategy

CG: dont' know what to do with this scheme; never seen and never heard of; to be deleted or replaced ?

In [Abbo84a]????, a *mixed strategy* was introduced based on the *shortest seek* (also called greedy strategy) and the *balanced strategy*. As shown in Figure 25, every time data are retrieved from disk they are transferred into buffer memory allocated for the respective data stream. From there, the application process removes them one at a time. The goal of the scheduling algorithm is:

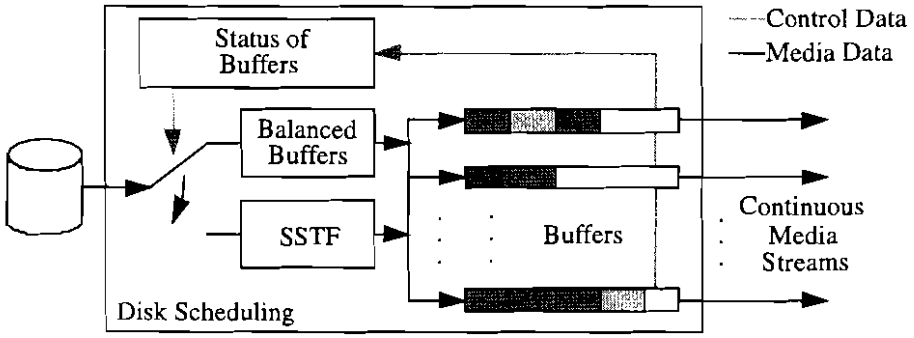


Figure 25: Mixed disk scheduling strategy

- To maximize transfer efficiency by minimizing seek time and latency.
- To serve process requirements with a limited buffer space.

With shortest seek, the first goal is served, i.e., the process of which data block is closest is served first. The balanced strategy chooses the process which has the least amount of buffered data for service because this process is likely to run out of data. The crucial part of this algorithm is the decision of which of the two strategies must be applied (shortest seek or balanced strategy). For the employment of shortest, seek two criteria must be fulfilled: the number of buffers for all processes should be balanced (i.e., all processes should nearly have the same number of buffered data) and the overall required bandwidth should be sufficient for the number of active processes, so that none of them will try to immediately read data out of an empty buffer. In [Abbo84a]????, the urgency is introduced as an attempt to measure both. The urgency is the sum of the reciprocals of the current "fullness" (amount of buffered data). This number measures both the relative balance of all read processes and the number of read processes. If the urgency is large, the balance strategy will be used; if it is small, it is safe to apply the shortest seek algorithm.

Continuous Media File System

CMFS Disk Scheduling is a non-preemptive disk scheduling scheme designed for the Continuous Media File System (CMFS) at UC-Berkeley [AOG91]. Different policies can be applied in this scheme. Here the notion of the slack time H is introduced. The slack time is the time during which CMFS is free to do non-real-time operations or workahead for real-time processes, because the current workahead of each process is sufficient so that no process would starve, even if it would not be served for H seconds. The considered real-time scheduling policies are:

- The *Static/Minimal policy* is based on the minimal *Workahead Augmenting Set (WAS)*. A process p_i reads a file at a determined rate R_i. To each process, a positive integer M_i is assigned which denotes the time overhead required to read a block covering, for example, the seek time. The CMFS performs a set of operations (i.e., disk operations required by all processes) by seeking the next block of a file and reading M_i blocks of this file. Note, we consider only read operations; the same also holds, with minor modifications, for write operations. This seek is done for every process in the system. The data read by a process during this operation "last" $\frac{M_i \times A}{R_i}$, where A is the block size in bytes. The WAS is a set of operations where the data read for each process "last longer" than the worst-case time to perform the operations (i.e., the sum of the read operations of all processes is less than the time read data last for a process). A schedule is derived from the set that is workahead-augmenting and feasible (i.e., the requests are served in the order given by the WAS). The *Minimal Policy*, the minimal WAS, is the schedule where the worst-case elapsed time needed to serve an operation set is the least (i.e.,

the set is ordered in a way that reduces time needed to perform the operations, for example, by reducing seek times). The *Minimal Policy* does not consider buffer requirements. If there is not enough buffer, this algorithm causes a buffer overflow. The *Static Policy* modifies this schedule such that no block is read if this would cause a buffer overflow for that process. With this approach, starvation is avoided, but its use of short operations causes high seek overhead.

- With the *Greedy Policy*, a process is served as long as possible. Therefore, it computes at each iteration the slack time H. The process with the smallest workahead is served. The maximum number n of blocks for this process is read; n is determined by H (the time needed to read n blocks must be less than or equal to H) and the currently available buffer space.
- The *Cyclical Plan Policy* distributes the slack time among processes to maximize the slack time. It calculates H and increases the minimal WAS with H milliseconds of additional reads; an additional read for each process is done immediately after the regular read determined by the minimal WAS. This policy distributes workahead by identifying the process with the smallest slack time and schedules an extra block for it; this is done until H is exhausted. The number of block reads for the least workahead is determined. This procedure is repeated every time the read has completed.

The *Aggressive* version of the Greedy and the Cyclical Plan Policy calculates H of all processes except the least workahead process that is immediately served by both policies. If the buffer size limit of a process is reached, all policies skip to the next process. Non-real-time operations are served if there is enough slack time. Performance measurements of the above introduced strategy showed that Cyclical Plan increases system slack faster at low values of the slack time (which is likely to be the case at system setup). With a higher system slack time, apart of the Static/Minimal Policy, all policies perform about the same.

All of the disk scheduling strategies described above have been implemented and tested in prototype file systems for continuous media. Their efficiency depends on the design of the entire file system, the disk layout, tightness of deadlines, and last but not least, on the application that is behaving. It is not yet common sense which algorithm is the "best" method for the storage and retrieval of continuous media files. Further research must show which algorithm serves the timing requirements of continuous media best and ensures that aperiodic and non-real-time requests are efficiently served.

5.4 Replication

Content replication is a means to answer two issues at the storage management level: availability in case of disk of machine failures, and limits to the number of concurrent access to individual titles because of limits on the throughput of the hardware. The failure handling argument is very similar at the storage management level as at the disk controller level, with the major difference that the storage management can apply various kinds of storage media to store replica (e.g. tapes, disks or main memory). Considerations on this issue have been elaborated in [RW94]. The alternatives for using replication to increase the number of concurrent deliveries of one file, however, are increased in this component.

On-demand applications can be partitioned into two families by the aging characteristics of their content. The content of online archives is assumed to be relatively time-independent and it is accessed based on the current interests of the customers. The content of news-on-demand and video-on-demand systems is expected to exhibit a popularity life-cycle like a newspaper or a movie. For the latter, the existance of a single copy of the content on a media server may not be sufficient to serve the necessary number of concurrent streams for a true on-demand systems from the storage subsystems where it is located.

Static replication

The simplest approach to replication that can be taken is the explicit duplication of content files, by storing the file on multiple machines and providing the user with a choice of access points. This is fre-

quently done in the Internet today: the content provider stores keeps copies of the original version up to date on servers close to the user. Using the more elaborate manual options, the content is duplicated manually, and an application provides alternating copies of the file under the same name.

A static placement policy that uses such estimated load information for the placement of video objects is proposed in [DS95b]. This static placement policy is complementary to the proposed policy, as it reduces, but cannot elimitate, dynamic imbalances.

Dynamic Segment Replication

Dynamic segment replication as it is introduced in [DKS95] is designed for content which is accessed read-only and which can be split into equally-sized segments of a size that is conveniently handled by the file system. Fixing segment sizes as well as chosing segments that are large in comparison to a disk block are decisions that are made to keep the implementation overhead low. Since contiunous media data is delivered in linear order, a load increase on a specific segment can be used as a trigger to replicate this segment and all following segments to other disks. Such segments are considered temporary segments in contrast to the original segments, which are permanent segments. One of the major advantages of this replication policy is that it takes not only the request frequency of individual movies into account. Rather than this, the load of the disk is also considered. Specifically, the decision is made in the following way: each disk x has a pre-specified threshold for the number of concurrent read requests B_x that must be exceeded by the sum of all segments' read operations in the current read cycle of the disk (where 'cycle' means the playout time of one segment) as well as by next read cycles to initiate the replication algorithm.

To simplify the calculation, the read requests are considered uniformly distributed over all replicas rather than taking requests to other segments on the same disk into account. In this way, the future load in t cycles for the i-th segment is predicted as n_{i-t}/r_i where n_{i-1} is the number of viewers of segment $i-t$ and r_i is the number of current replicas of the segment. For all segments j ($j<t$), it is assumed that the current arrival rate n_j/r_i will be maintained in the future. If the sum of the expected load for all segments on a disk exceeds B_x, the replication is triggered. Then, the algorithm must identify a segments for replication. Since the the approach replicates segments only when they are retrieved from disk because of a client request, in order not to add additional load, replication can start only when a stream starts reading a new segment. Hence, if the disk load exceeds B_x at a segment boundary crossing, we must decide whether it is desirable to replicate this segment. Not in any case, but only if the replication of this segment has the highest estimated payoff among all the segments on the disk, it is replicated. If the gain in replicating a different segment is considerable, a boundary crossing to that segment is awaited. The estimated payoff p_i is computed as

$$p_i = \left(\frac{1}{r_i} - \frac{1}{r_i + 1}\right) \sum_{j=0}^{i-1} n_j w^{i-j-1}$$

where w is a weighting factor. w can be chosen big to put a stronger weight on long-term predictions; this is a good selection when the load on individual segments stays similar for a relatively long time. If the load on segments is fluctuating strongly, the expectation of future behaviour is unreliable and should have less relevance, expressed by a lower weight w.

Threshold-Based Dynamic Replication

The threshold-based dynamic replication introduced in [LLG98] considers whole movies rather than movie segments, and it takes all disks of the system into account to determine whether a movie should be replicated. It is assumed for this approach that the term 'disk' does not necessarily mean a single physical disk but a logical disk which may also be an array of physical disks with a single representation to the storage management. Still, it is assumed that the media server is large and consists of many

such logical disks. The service capacity in number of concurrent streams of such a disk x is called B_x, the average service capacity of all disks is called $\bar{B}$.

A replica of a movie is assumed to be stored completely on one of these disks. For each movie i of length m_i, a probability to be selected in a new request P_i as well as an request arrival rate λ must be computed from earlier requests. The replication threshold T_i is than computed as $T_i = min(p_i \lambda m_i, h\bar{B})$, where h a constant value to limit the probability of replication. For each disk x, the measured current load L_x is taken into account to compute the current available service capacity A_i for serving video i by calculating

$$A_i = \sum_{x \in R_i} (B_x - L_x)$$

where R_i is the set of disks that carry replicas of i. If $A_i < T_i$, a replication of movie i is triggered. Similarly, [LLG98] proposes a decision for discarding replications when the number of concurrent requests n_{ix} on a movie i at disk x decreases. The following condition is checked before a replica is removed:

$$\sum_{y \in R_i \setminus x} (B_y - L_y) - n_{ix} > T_i + D$$

This inequality integrates two important conditions. The inequality

$$A_i = \sum_{x \in R_i} B_x - L_x > \sum_{y \in R_i \setminus x} (B_y - L_y) - n_{ix} > T_i + D > T_i$$

implies that the replication is not triggered again immediately after a de-replication, and

$$\sum_{y \in R_i \setminus x} (B_y - L_y) - n_{ix} > T_i + D > 0$$

guarantees that all streams on disk x can be served from the remaining replicas. D is an additional threshold to reduce the probability of an oscillation between replication and de-replication further.

The approach includes also the proposal to replicate a movie from the least loaded disk to the destination disk because an overhead may be induced by an additional read operation on the source disk. For the selection of the destination disk out of the set of disks that do not yet hold a replica of the movie in question, multiple approaches are considered. The most complex one takes the number of current streams into account, but assumes that all ongoing replications are already finished and the streams are distributed onto the disks as if the replicas were already active. For the replication itself, various policies are proposed.

Injected Sequential Replication adds additional read load to one disk because it behaves like an additional client, by copying the movie at the normal play rate from the source disk to the target disk.

Piggybacked Sequential Replication is identical to the replication used in the Dynamic Segment Replication: the movie is written to the destination disk while it is delivered to one client from the same memory buffer. Since this scheme makes replication decisions for a movie always during admission control for new clients, this does not add complexity to identify the source copy of the operation. However, the copy operation is affected when VCR operations on the movie are performced.

Injected Parallel Replication use a multiple of the normal data rate of the movie to replicate the movie faster from the source disk to the destination disk. In order not to inhibit admission of new customers, this multiple of the normal data rate is limited.

Piggybacked Parallel Replication copies at the normal rate of the movie, but not only from the position of the newly admitted client. Instead, later parts of the movie are copied at the same time from the buffers which serve clients that are already viewing the movie. Obviously, this approach needs unusual low level support because data is written in parallel to different positions in a not-yet complete file.

Piggybacked and Injected Parallel Replication combines the other parallel replication approaches to replicate parts by the injected approach of the movie that would have to be copied late in a piggybacked parallel replication mode because no client is expected to view those parts in the near future.

5.5 Supporting Heterogenous Disks

Approaches of measuring the performance of disks and assigning data to them based on their performance characteristics becomes relevant when large-scale systems are considered. Such systems are assumed to grow over a long period of time, and considering the availability of time for which a specific series of hard disks is produced today, the chance to maintain a server that consists of homogeneous disks is low. The simple approach is to identify the disks with the smallest I/O-bandwidth and make this the reference bandwidth for all calculations. This approach would not collide with typical buffer management strategies since the strategies to keep the playout buffers filled is so resource-conservative that even disk read times are taken into account for refill operations.

Since both disk space and bandwidth have increased considerably in the past, the simple approach may be extremely pessimistic when the number of potentially supported streams is calculated. For example, an SSA storage system may deliver data at a rate of 100 MByte/s, while an typical SCSI-II fast/wide RAID system connected to the same media server delivers only 20 MByte/s. Various means can be applied to reduce the impact of heterogenous storage systems.

Bandwidth to Space Ratio

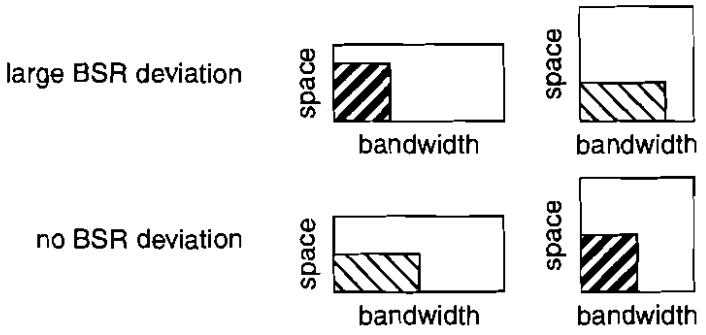


Figure 26: Bandwidth to space ratio deviation

In [DS95b], not only the raw throughput of such logical disks is considered, but rather the ratio of throughput to storage capacity (*bandwidth to space ration*, or *BSR*). This approach assumes that approaches to replication such as the dynamic segment replication policy mentioned above take care of a smoothing the average number of concurrent streams from the same movie. However, if throughput requirements of movies' copies (the product of data rate and number of concurrent viewers) differ, the throughput requirements for equally-sized segments of that video differ, too, and locating popular, high data-rate movies on large but throughput-restricted disks wastes space in comparison with storing them on smaller or faster disks. The same argument holds for variable-sized movies if the threshold-based dynamic replication is used. The decision to replicate a video according to the BSR scheme is identical to that of the dynamic segment replication, but the destination disk is chosen according to least expected deviation of the movie's BSR (data rate*concurrent viewers/length) from the disk's BSR (maximum throughput/size). Figure 26 illustrates this BSR deviation. It is a detail of the BSR approach that as many replicas as possible are created to approach the identity of used to available throughput ratios among all disks of the system as good as possible. When the number of viewers for a movies changes, the best distribution is newly computed.

6 File System

The *file system* is said to be the most visible part of an operating system. Most programs write or read files. Their program code, as well as user data, are stored in files. The organization of the file system is an important factor for the usability and convenience of the operating system. A file is a sequence of information held as a unit for storage and use in a computer system [Krak88]????.

Files are stored in secondary storage, so they can be used by different applications. The life-span of files is usually longer than the execution of a program. In traditional file systems, the information types stored in files are sources, objects, libraries and executables of programs, numeric data, text, payroll records, etc. [PeSi83]????. In multimedia systems, the stored information also covers digitized video and audio with their related real-time "read" and "write" demands. Therefore, additional requirements in the design and implementation of file systems must be considered.

The file system provides access and control functions for the storage and retrieval of files. From the user's viewpoint, it is important how the file system allows file organization and structure. The internals, i.e., the organization of the file system, deal with the representation of information in files, their structure and organization in secondary storage.

6.1 Traditional File Systems

Although there is no such term as a traditional file system, a couple of file systems can be considered traditional for their wide-spread use in computer systems for all-round operations. In the operating system family stemming from MS-DOS, the FAT filesystem is the original one, in the family of Unix (-like) operating system, the Berkeley Fast FileSystem is a typical representative. Log-structured filesystems provide some additional functionality but must be counted among these all-round filesystems rather than multimedia filesystems.

FAT

One way is to use linked blocks, where physical blocks containing consecutive logical locations are linked using pointers. The file descriptor must contain the number of blocks occupied by the file, the pointer to the first block and it may also have the pointer to the last block. A serious disadvantage of this method is the cost of the implementation for random access because all prior data must be read. In MS-DOS, a similar method is applied. A *File Allocation Table (FAT)* is associated with each disk. One entry in the table represents one disk block. The directory entry of each file holds the block number of the first block. The number in the slot of an entry refers to the next block of a file. The slot of the last block of a file contains an end-of-file mark [Tane87]????.

Berkeley FFS and relatives

Another approach is to store block information in mapping tables. Each file is associated with a table where, apart from the block numbers, information like owner, file size, creation time, last access time, etc., are stored. Those tables usually have a fixed size, which means that the number of block references is bounded. Files with more blocks are referenced indirectly by additional tables assigned to the files. In UNIX, a small table (on disk) called an i-node is associated with each file (see Figure 27). The indexed sequential approach is an example for multi-level mapping; here, logical and physical organization are not clearly separated [Krak88]????.

Log-structured filesystem

The log-structured file system was devised to ensure fast crash recovery, increased write performance and an option for versioning in the file system. The basic approach is to write data always asynchro-

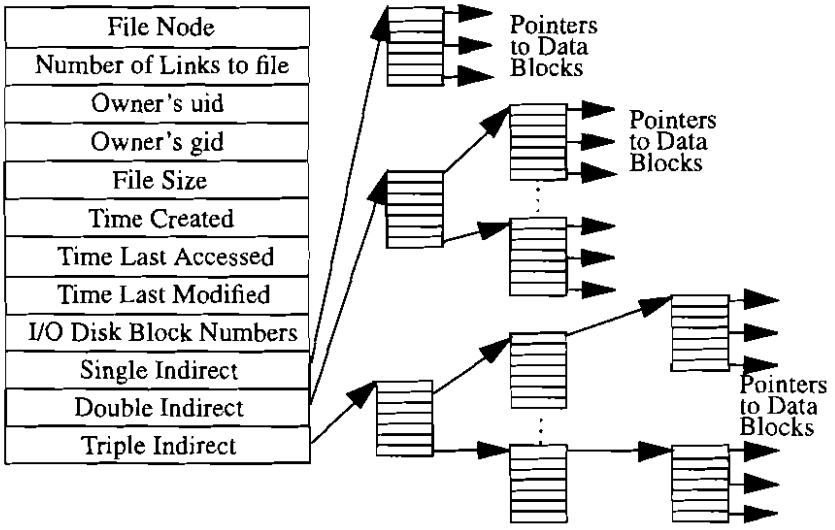


Figure 27: The UNIX i-node [Tane87]????

nously to free space on the disk and to keep a log of all write operations on the disk. This ensures that in case of a machine crash, the information that was last written to disk can be identified from the log file, and the blocks that have most likely been corrupted can be checked explicitly rather than checking the whole disk. If the block can not be recovered, a consistant state can always be recovered in spite of this by examining the log; even in case of a modify operation, the old block is still present on the disk and can be identified by the log. The effect of writing always target write operations to a contiguous free space on the disk makes writing to the disk also more efficient since head movement is large reduced. In case of a small number of concurrent write operation, this results also a largely contiguous files.

Various ways to use a log-structured filesystem are conceivable. Figure 28 shows in-band logging as it was presented in [OD89] and a variation that uses a separate log partition, which was mentioned by [OD89] as an alternative approach. Obviously, write operations in in-band logging are faster because no seek operation to the log partition is necessary. On the other hand, log partitions may be located on seperate disks, close to the logged partition, or the log records can be kept in memory until sufficiently large block of log information has been collected to make the seek operation feasible. While in-band logging provides faster write operations, it suffers from the complexity of necessary compression operations that perform similar to a garbage collection. Using a separate log partition, superseeded information from an earlier write operation can be flushed everytime a new log information block is write and the log can be compressed to contain only the remaining relevant information.

Directory Structure

Files are usually organized in *directories*. Most of today's operating systems provide tree-structured directories where the user can organize the files according to his/her personal needs. In multimedia systems, it is important to organize the files in a way that allows easy, fast, and contiguous data access.

6.2 Multimedia File Systems

Compared to the increased performance of processors and networks, storage devices have become only marginally faster [Mull91]????. The effect of this increasing speed mismatch is the search for new storage structures, and storage and retrieval mechanisms with respect to the file system. Continuous media data are different from discrete data in:

- Real Time Characteristics

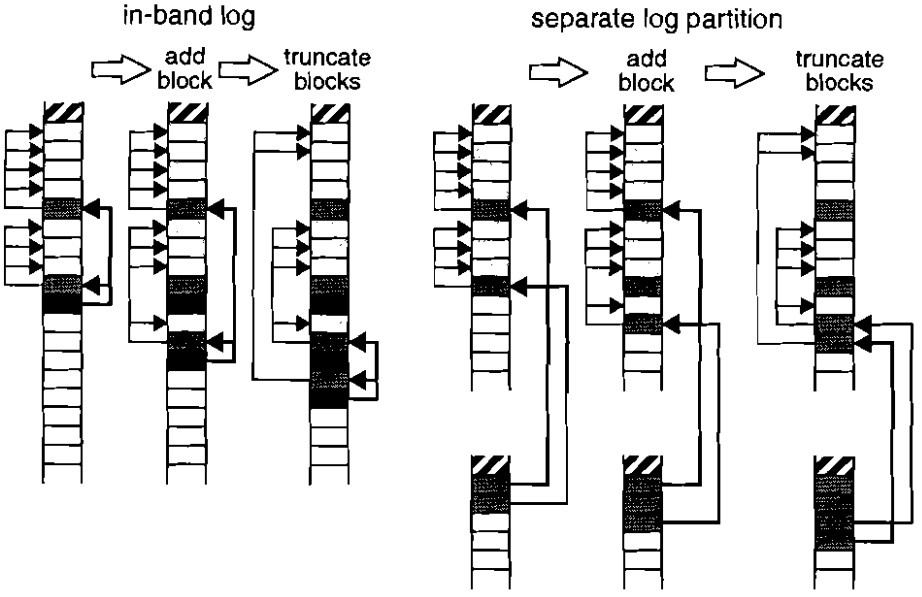


Figure 28: Disk operations in log-structured file systems

As mentioned previously, the retrieval, computation and presentation of continuous media is time-dependent. The data must be presented (read) before a well-defined deadline with small jitter only. Thus, algorithms for the storage and retrieval of such data must consider time constraints, and additional buffers to smooth the data stream must be provided.

- File Size

  Compared to text and graphics, video and audio have very large storage space requirements. Since the file system has to store information ranging from small, unstructured units like text files to large, highly structured data units like video and associated audio, it must organize the data on disk in a way that efficiently uses the limited storage. For example, the storage requirements of uncompressed CD-quality stereo audio are 1.4 Mbits/s; low but acceptable quality compressed video still requires about 1Mbit/s using, e.g., MPEG-1.

- Multiple Data Streams

  A multimedia system must support different media at one time. It does not only have to ensure that all of them get a sufficient share of the resources, it also must consider tight relations between different streams arriving from different sources. The retrieval of a movie, for example, requires the processing and synchronization of audio and video.

There are different ways to support continuous media in file systems. Basically there are two approaches. With the first approach, the organization of files on disk remains as is. The necessary real-time support is provided through special disk scheduling algorithms and sufficient buffer to avoid jitter. In the second approach, the organization of audio and video files on disk is optimized for their use in multimedia systems. Scheduling of multiple data streams still remains an issue of research.

In this section, the different approaches are discussed and examples of existing prototypes are introduced. First, a brief introduction of the different storage devices employed in multimedia systems is given. Then, the organization of files on disks is discussed. Subsequently, different disk scheduling algorithms for the retrieval of continuous media are introduced.

6.3 Example Multimedia File Systems

Video File Server

Continuous media data are characterized by consecutive, time-dependent logical data units. The basic data unit of a motion video is a frame. The basic unit of audio is a sample. Frames contain the data asso-

ciated with a single video image, a sample represents the amplitude of the analog audio signal at a given instance. Further structuring of multimedia data was suggested in the following way [RaVi91, Rang93, StFr92]????: a strand is defined as an immutable sequence of continuously recorded video frames, audio samples, or both. It means that it consists of a sequence of blocks which contain either video frames, audio samples or both. Most often it includes headers and further information related to the type of compression used. The file system holds primary indices in a sequence of *Primary Blocks*. They contain mapping from media block numbers to their disk addresses. In *Secondary Blocks* pointers to all primary blocks are stored. The *Header Block* contains pointers to all secondary blocks of a strand. General information about the strand like, recording rate, length, etc., is also included in the header block.

Media strands that together constitute a logical entity of information (e.g., video and associated audio of a movie) are tied together by synchronization to form a multimedia rope. A rope contains the name of its creator, its length and access rights. For each media strand in this rope, the strand ID, rate of recording, granularity of storage and corresponding block-level are stored (information for the synchronization of the playback start for all media at the strand interval boundaries). Editing operations on ropes manipulate pointers to strands only. Strands are regarded as immutable objects because editing operations like insert or delete may require substantial copying which can consume significant amounts of time and space. Intervals of strands can be shared by different ropes. Strands that are not referenced by any rope can be deleted, and storage can be reclaimed [RaVi91]????. The following interfaces are the operations that file systems provide for the manipulation of ropes:

- RECORD [media] [requestID, mmRopeID]
  A multimedia rope, represented by mmRopeID and consisting of media strands, is recorded until a STOP operation is issued.
- PLAY [mmRopeID, interval, media] requestID
  This operation plays a multimedia rope consisting of one or more media strands.
- STOP [requestID]
  This operation stops the retrieval or storage of the corresponding multimedia rope.
- Additionally, the following operations are supported:
  - INSERT [baseRope, position, media, withRope, withInterval]
  - REPLACE [baseRope, media, baseInterval, withRope, withInterval]
  - SUBSTRING [baseRope, media, interval]
  - CONCATE [mmRopeID1, mmRopeID2]
  - DELETE [baseRope, media, interval]

Figure 29 provides an example of the INSERT operation, whereas Figure 30 shows the REPLACE operation.

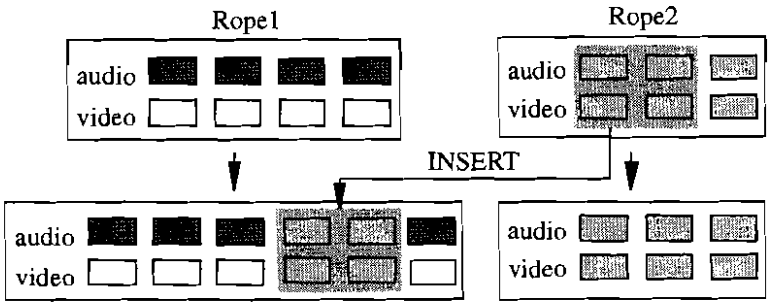


Figure 29: INSERT operation

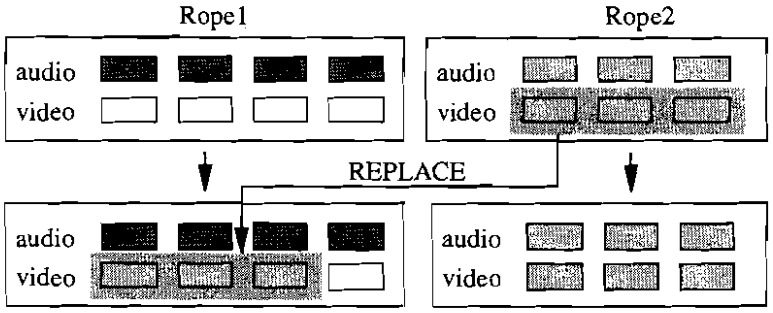


Figure 30: REPLACE operation

The storage system is divided into two layers:

- The *rope server* is responsible for the manipulation of multimedia ropes. It communicates with applications, allows the manipulation of ropes and communicates with the underlying *storage manager* to record and play back multimedia strands. It provides the rope abstraction to the application. The rope access methods were designed similarly to UNIX file access routines. Status messages about the state of the play or record operation are passed to the application.
- The *storage manager* is responsible for the manipulation of strands. It places the strands on disk to ensure continuous recording and playback. The interface to the rope server includes four primitives for manipulating strands:

  1. "PlayStrandSequence" takes a sequence of strand intervals and displays the given time interval of each strand in sequence.
  2. "RecordStrand" creates a new strand and records the continuous media data either for a given duration or until StopStrand is called.
  3. "StopStrand" terminates a previous PlayStrandSequence or RecordStrand instance.
  4. "DeleteStrand" removes a strand from storage.

The experimental Video File Server introduced in [Rang93]???? supports integrated storage and retrieval of video. The "Video Rope Server" presents a device-independent directory interface to users (Video Rope). A Video Rope is characterized as a hierarchical directory structure constructed upon stored video frames. The "Video Disk Manager" manages a frame-oriented motion video storage on disk, including audio and video components.

Fellini

The Fellini Multimedia Storage Server ([MNO96]) has the goal to support real-time as well as non-real-time data, but its file system is dedicated to the storage and retrieval of continuous media data. It organizes data similar to a Unix system, from which it is derived, but data and meta-information about the data are stored separately.

The data stored on disk using the raw disk interface and it is addressed both in main memory and on disk in terms of pages. These pages are countable and they are sorted, intuitively, in the order of a normal forward playout. Figure 31 shows the connection of buffer headers and the Current Buffer List (CBL). For each file on disk, one CBL exists as a representation of that file. The CBL header contains an *ondisk id*, representing the file uniquely to all clients, and an *incore id*, containing the open file handle of the file on disk. This abstraction is necessary because a file should be opened only once for all concurrent accesses. Besides other information, the CBL header refers to a hash array that allows quick access to the buffer headers representing pages which are currently in main memory. At startup time,

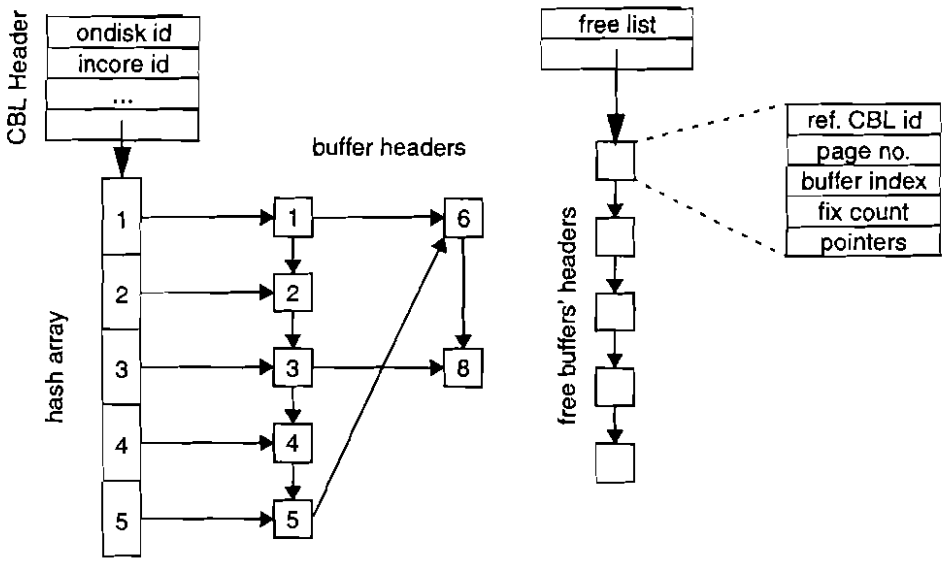


Figure 31: Fellini's Current Buffer List

the Fellini server allocates free buffers in main memory, pins* them and stores their headers, with a 0 fix count, in the free list.

When a page is requested by a client for the first time, a buffer from the free list is chosen, inserted into the appropriate CBL list, and its fix count is increased to 1. When it is allocated again by another client, its fix count is increased. In order to share the buffers between the server and its clients, the buffer headers and buffers are located in shared memory. When a client stops using a page, its fix count is decreased. When the fix count reaches 0, the page is not automatically deallocated. Only if a new page is requested and the free list is empty, the CBL which has had not active clients for the longest time (a so-called aged CBL) is forced to release their page with the highest logical page number (the one that would be the last to be accessed by a new client that plays the file from the beginning to the end). If no aged CBLs are available, all CBLs are checked for unfixed pages, and such a page is selected according to a weighting calculation that aims at identifying the pages that will not be used any more by any of the clients that are already in the system, and from that set, selecting the one with the highest page number (the one that would be last to be accessed by a newly arriving client at its queue at the given time).

The management of the data on disk is handled in a way that makes the use of multiple disks transparent to the user. The information about the location of the data is maintained in file control blocks (FCBs) that are similar to Unix file i-nodes at the root of the directory tree. Subdirectories are not supported. All FCBs of the system are stored in a single Unix file. This file is big enough to store FCBs for the entire space that is available to the Fellini server.

Table 2: Indentification of Unix and Fellini APIs

Unix API	Fellini RT-API	Fellini non-RT-API
open	begin_stream	fe_nr_open
read	retrieve_stream	fe_nr_read
write	store_stream	fe_nr_write

*. a memory page is said to be pinned when the OS is forbidden to swap it out of the main memory

Table 2: Indentification of Unix and Fellini APIs

Unix API	Fellini RT-API	Fellini non-RT-API
seek	seek_stream	fe_nr_seek
close	close_stream	fe_nr_close

To the rest of the system, the Fellini server offers an API that is very similar to the Unix filesystem API for convenience. Table 2 identifies the Fellini function calls with their Unix couterparts.

Symphony

The Symphony file system ([SGRV97]) is a filesystem designed for the storage and delivery of heterogeneous data types. Particularly, Symphony addresses the following issues:

- support real-time as well as non-real-time requests
- support multiple block sizes and control over their placement
- support a variety of fault-tolerance techniques
- provide a two-level meta data structure that alles type-specific information to be attached to each file

To do this, the file system is divided into a data-type specific and a data-type independent layer. The data-type independent layer implements a scheduler that uses a modified SCAN-EDF approach to schedule real-time requests and adds non-real-time requests according to C-SCAN as long as no deadlines for the real-time requests are violated. It offers to the data-type specific layer variable block sizes that are multiples of a minimal basic block size that is defined at file system creation time and an option to express a preferred, but not guaranteed, location of blocks in terms of disk and location on that disk. To locate variable-sized blocks on disk efficiently, each block is addressed through an indirection as shown in Figure 32. Finally, the fault tolerant layer can optionally use parity information to deliver verified data, or skip this check in favour of speed.

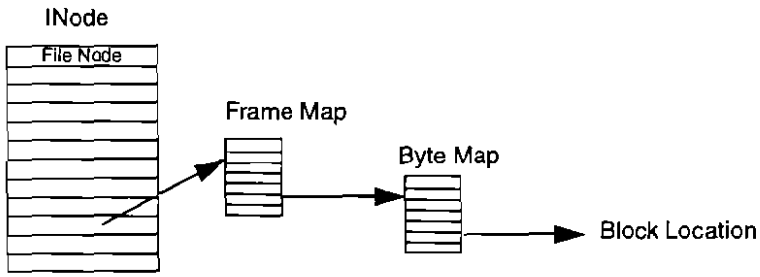


Figure 32: Symphony's indirect block location

The data-type specific layer offers a set of modules (audio, video and text) for different kinds of data, where each of the modules makes use of the features provided by the underlying layer according to its requirements.

E.g. the video module implements variable size data blocks by observing a video stream at writing time and by deriving an approximation of a maximum block size from the start of the stream. This block size is negotiated with the data-type independent layer, and subsequently, space that remain unoccupied because the actual block was smaller than the maximum block size is filled with data from the following block until the buffer space of a full block is saved. This allows for the continuous write as well as read operations at the cost of buffering one additional block size of memory per stream in the worst case. Concerning the placement policy, the video module prefers to stripe a file over as many disks as possible to maximize the utilization of I/O bandwidth for the case of parallel retrieval operations. In addition

to this, during retrieval operations it starts to cache video blocks in main memory for efficiency according to the Interval Caching Policy.

Symphony supports the typical Unix file operations and extends this interface by function calls that are necessary to support quality of service. These additional functions are *ifsPeriodRead()*, *ifsPeriodicWrite()*, *ifsQoSNegotiate()*, *ifsQosConfirm()* and *ifsgetMetaData()*.

7 Memory Management

Memory management in media servers is mainly concerned with the assignment of part of the media server's main memory to the delivery of a multimedia stream. While straightforward implementations of media servers do not exploit all content's movement through the main memory of the media server but rather rely on the filesystem implementation to allocate sufficient bandwidth for a smooth delivery of each stream, a couple of approaches exist for the exploiteation of this phase of data delivery. One approach was demonstrated earlier with the Fellini server's file system ("Fellini" on page 34). Others are listed in the following.

Interval Caching Policy

[DS93] introduces partial replication of multimedia files for load-balancing in multimedia systems. It is based on the observation that if there were a number of consecutive requests for the same video, and if the blocks read in by the first request were copied to another disk, it would be possible to switch the following requests to the partial replica just created.

Generalized Interval Caching Policy

The Interval Caching Policy, proposed in [DS93], exploits the movement of data through the main memory of a video server by keeping the data of such streams in memory, which are followed temporarly close by another stream of the same object. This policy is refined in [DS95] to take into account that the interval caching policy does not handle short files appropriately when the media server is handling a mixed workload rather than a videos.

Batching

Batching is an approach introduced in [DSST94] to exploit the memory bandwidth and to save disk bandwidth in media servers by defining a temporal cycles called batching windows. All requests that arrive within such a cycle are collected and at the end of the cycle, all requests to the same content are serviced from the same file and buffer. This approach weakens the on-demand idea in comparison to the interval caching policy, but it recovers potentially large amounts of main memory because content can be discarded from the main memory immediately after playout and it will be re-loaded only after the next cycle. [DSS94] modifies this approach towards dynamic batching, which services requests as soon as a stream becomes available. Two selection policies, first come first serve (FCFS) and maximum queue length (queue length is defined by the number of user who requested that file), are compared, and FCFS is shown to be the more performant.

Piggybacking

The aggregation of streams that deliver the same content in close sequence without the use of batching window was proposed by means of piggybacking ([GLM96]), i.e. one stream of a content file that is shortly preceeding another stream of the same file should be joined with the later one. The general means to do this is an increase in the speed of the later stream and/or a decrease in the speed of the earlier stream until they join. Various strategies for joining more than a pair of streams are then investigated in detail in [GLM96].

Content Insertion

For the video-on-demand special case, [VL95] proposes the most radical extension of that scheme to date by offering content insertion to force larger numbers of streams into a time window which is small enough to allow the use of the piggybacking technique to join them into a single stream. Such inserted

content from a content loop like an eternal advertisement show or from a continuous news show might be acceptable to the user to stay tuned. Alternatives might be a lengthening or shorting of introducing scenes of a movie. In [KVL97], it is then offered that this technique can also be used for providing a just as pragmatic and radical solution to problems such as server overload or partial server failure by diverting users into an advertisement loop or presenting other fill-in content until the problem can be fixed or until an aggregation with an action stream can be performed.

8 References

[AOG91] D. P. Anderson, Y. Osawa, R. Govindan. Real-Time Disk Storage and Retrieval of Digital Audio/Video Data. TR UCB/CSB 91/646, University of California, Berkely, September 1991.

[BGW97] Michael Bär, Carsten Griwodz, Lars Wolf. Long-term Movie Popularity in Video-on-Demand Systems. In Proceedings of the 5th ACM Int'l Multimedia Conference, pages 349-358, Seattle, Nov. 9-13, 1997.

[BY95] Birk, Yitzhak. Track Pairing: A Novel Data Layout for VOD Servers with Multi-Zone Reocrding Disks. In Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS) 95, Washington D.C.,May 1995.

[Chu96] Soon M. Chung (Ed.). Multimedia Information Storage and Management. Kulwer Academic Publishers, Norwell, Mass. 1996. ISBN 0-7923-9764-9.

[CKY93] Mon-Song Chen, Dilip D. Kandlur, Philip S. Yu. Optimization of the Grouped Sweeping Scheduling (GSS) with Heterogeneous Multimedia Streams.Proceedings of ACM MM '93, Anaheim, CA, August, 1993, pp. 235-242.

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26,2:145-185, June 1994.

[CT97] Shenze Chen, Manu Thapar. A Novel Video Layout Strategy for Near-Video-on-Demand Servers. In Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS) 97, pages 37-45, Ottawa, June 3-6, 1997.

[DKS95] Asit Dan, Martin Kienzle, and Dinkar Sitaram. A dynamic policy of segment replication for load-balancing in video-on-demand servers. *Multimedia Systems*, 3:93-103, 1995.

[DS93] Asit Dan and Dinkar Sitaram. Buffer Management Policy for an On-Demand Video Server. RC 19347, IBM Research Division, 1993.

[DS95] Asit Dan and Dinkar Sitaram. A Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads. RC 20206 (89404), IBM Research Division, September 1995.

[DS95b] Asit Dan and Dinkar Sitaram. An Online Video Placement Policy based on Bandwidth to Space Ratio (BSR). In Proceedings of the 1995 SIGMOD, San Jose, California, May 22-25, pages 376-385, 1995.

[DSS94] Asit Dan, Dinkar Sitaram, Perwez Shahabuddin. Dynamic Batching Policies for an On-Demand Video Server. Multimedia Systems. ??? 1994.

[DSST94] Asit Dan, Perwez Shahabuddin, Dinkar Sitaram, Don Towsley. Channel Allocation under Batching and VCR Control in Video-On-Demand Systems, IBM Research Report, RC 19588, Sept. 1994.

[GH94] D. James Gemmell, Jiawei Han. Multimedia Network File Servers: Multi-Channel Delay Sensitive Data Retrieval. Multimedia Systemes 1(6), pp. 240-252, 1994.

[GIZ96] Shahram Ghandeharizadeh, Doug Ierardi, and Roger Zimmermann. An Algorithm for Disk Space Management to Minimize Seeks. Information Processing Letters, 1996.

[GLM96] Leana Golubchik, John C. S. Lui, Richard R. Muntz. Adaptive Piggybacking: A Novel Technique for Data Sharing in Video-on-Demand Storage Servers. Multimedia Systems 4, pp. 140-155, 1996.

[GK96] Shahram Ghandeharizadeh and Dongho Kim. On-line Reorganization of Data in Scalable Continous Media Servers. In Proceedings of Database and Expert Systems Applications 1996, Zurich, Switzerland, pages 751-768, 1996.

[GKS95] Shahram Ghandeharizadeh, Seon Ho Kim, and Cyrus Shahabi. Continuous Display of Video Objects Using Multi-Zone Disks. TR 94-592, USC, April 1995.

[GZS+96] Shahram Ghandeharizadeh, Roger Zimmermann, Weifeng Shi, Reza Rejaie, Doug Ierardi, and Ta-Wei Li. Mitra: A Scalable Continuous Media Server. TR 96-628, USC, February 1996.

[HS95] Roger L. Haskin, Frank B. Schmuck. The Tiger Shark File System. Online in Almaden.

[KLC97] J. Kim, Y. Lho, K. Chung. An Effective Video Block Placement Scheme on VOD Server based on Multi-Zone Reocrding Disks. In Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS) 97, pages 29-36, Ottawa, June 3-6, 1997.

[Kor97] Jan Korst. Random Duplicated Assignment: An Alternative to Striping in Video Servers. In Proceedings of the 5th ACM Int'l Multimedia Conference, pages 219-226, Seattle, Nov. 9-13, 1997.

[Kra88] S. Krakowiak. Principles of Operating Systems. MIT Press, Cambridge, 1998.

[KVL97] Rajesh Krishnan, Dinesh Venkatesh, Thomas D. C. Little. A Failure and Overload Tolerance Mechanism for Continuous Media Servers. Proceedings of the ACM MM 97 Conference, pp. 131-142, 1997.

[LK91] E. K. Lee, R. H. Katz. Performacen consequences of parity placement in disk arrays. In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV). IEEE, New York, 190-199. 1991.

[LLG98] Peter W. K. Lie, John C. S. Lui, Leana Golubchik. Threshold-Based Dynamic Replication in Large-Scale Video-on-Demand Systems. Accepted for RIDE 98.

[LS93] P. Lougher, D. Shepherd. The Design of a Storage Service for Continuous Media. The Computer Journal, 36(1), pp. 32-42, 1993.

[MNO96] Cliff Martin, P. S. Narayanan, Banu Ozden, Rajeev Rastogi, Avi Silberschatz. The Fellini Multimedia Storage Server. In [Chu96], pp. 117-146, 1996.

[PW72] E. W. Peterson, E. J. Weldon. Error-Correcting Codes. 2nd ed. MIT Press, Cambridge, Mass, 1972. ????

[SDY93] Dinkar Sitaram, Asit Dan, and Philip S. Yu. Issues in the Design of Multiserver File Systems to Cope with Load Skew. In Proceedings of 2nd Int'l Conference on Parallel and Distributed Systems, San Diego, pages 214-221. IEEE Computer Society Press, hos Alamitos, 1993.

[OD89] John Ousterhout, Fred Douglis. Beating the I/O Bottleneck: A case for Log-Structured File Systems. Operating Systems Review, 23(1), pp. 11-28, 1989.

[PGK88] David. A. Patterson, Garth Gibson, Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD), Chicago, IL, pp. 109-116, June 1988.

[RTSP] sorry, too tired

[RW93] A. L. Narasimha Reddy, Jim Wyllie. Disk scheduling in a multimedia I/O system. Proceedings of ACM MM '93, Anaheim, CA, August, 1993, pp. 225-223.

[RW94] A. L. Narasimha Reddy, Jim Wyllie. I/O Issues in a Multimedia System. COMPUTER, 27(3), pp. 69-74, 1994.

[SD95] Scott D. Stoller, John D. DeTreville. Storage Replication and Layout in Video-on-Demand Servers. NOSSDAV95.

[SGRV97] Prashant J. Shenoy, Pawan Goyal, Sriram S. Rao, Harrick M. Vin. Symphony: An Integrated Multimedia File System. In Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking (MMCN'98), San Jose, CA, pp. 124-138, Jan 1998.

[TF95] William H. Tetzlaff and Robert Flynn. Elements of Scalable Video Servers. In Proceedings of COMPCON 1995, pages 239-248, 1995.

[TPBG93] F. A. Tobagi, J. Pang, R. Baird, M. Gang. Streaming RAID - A Disk Array Managment System for Video Files. Proceedings of ACM MM '93, Anaheim, CA, August, 1993, pp. 393-400.

[VL95] D. Venkatesh, T. D. C. Littlc. Dynamic Service Aggregation for Efficient Use of Resources in Interactive Video Delivery. Proceedings of the 5th NOSSDAV conference, pp. 113-116, Nov. 1995.

[WD97] Yuewei Wang, David H. C. Du. Weighted Striping in Multimedia Servers. In Proceedings of the International Conference on Multimedia Computing and Systems (ICMCS) 97, pages 102-109, Ottawa, June 3-6, 1997.

[YCK92] P. S. Yu, M.-S. Chen, D. D. Kandlur. Design and Analysis of a Grouped Sweeping Scheme for Multimedia Storage Management. 3rd Int'l Workshop on Network and Operating System Support for Digitial Audio and Video (NOSSDAV92), San Diego, Nov. 1992.

[ZG97] Roger Zimmermann and Shahram Ghandeharizadeh. Continuous Display Using Heterogeneous Disk-Subsystems. In Proceedings or ACM Multimedia 97, November 8-14, Seattle, pages 0-0, 1997.