1

STORAGE ARCHITECTURES FOR DIGITAL IMAGERY

Harrick M. Vin† and Prashant Shenoy‡

† *Department of Computer Sciences, University of Texas at Austin*

‡ *Department of Computer Science, University of Massachusettes, Amherst*

Rapid advances in computing and communication technologies coupled with the dramatic growth of the Internet has led to the emergence of a wide variety of multimedia applications such as distance education, online virtual worlds, immersive telepresence, and scientific visualization. These applications differ from conventional distributed applications in at least two ways. First, they involve storage, transmission, and processing of heterogeneous data types—such as text, imagery, audio, and video—that differ significantly in their characteristics (e.g., size, data rate, real-time requirements, etc.). Second, unlike conventional best-effort applications, these applications impose diverse performance requirements—for instance, with respect to timeliness—on the networks and operating systems. Unfortunately, existing networks and operating systems do not differentiate between data types and offer to all applications a single class of best-effort service. Hence, to support emerging multimedia applications, existing networks and operating systems need to be extended along several dimensions.

In this chapter, we discuss the issues involved in designing storage servers that can support such a diversity of applications and data types. First, we describe the specific issues that arise in designing a storage server for digital imagery and then discuss the architectural choices for designing storage servers that efficiently manage the storage and retrieval of multiple data types. Note that since it is difficult, if not impossible, to foresee requirements imposed by future applications and data types, a storage server that supports multiple data types and applications will needs to facilitate easy integration of new application classes and data types. This dictates that the storage server architecture be extensible, allowing it to be easily tailored to meet new requirements.

The rest of the chapter is organized as follows. We begin by examining techniques for placement of digital imagery on a single disk, a disk array, and a hierarchical storage architecture. We then examine fault-tolerance techniques employed by servers to guarantee high availability of image data. Next, we examine retrieval techniques employed by storage servers to efficiently access images and image sequences. Finally, we examine architectural issues in incorporating all of these techniques into a general purpose file system.

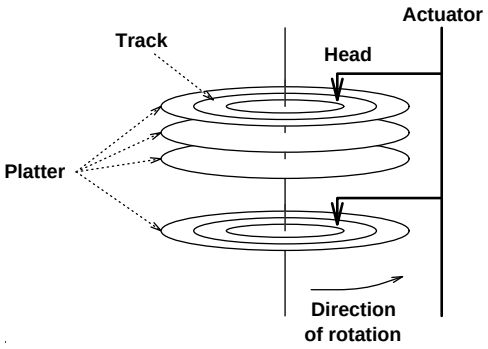


Figure 1 Architectural model of a conventional magnetic disk

1 STORAGE MANAGEMENT

1.1 Single Disk Placement

A storage server divides images into *blocks* while storing them on disks. In order to explore the viability of various placement models for storing these blocks on magnetic disks, let us first briefly review some of the fundamental characteristics of magnetic disks. Generally, magnetic disks consist of a collection of platters, each of which is composed of a number of circular recording tracks (see Figure 1). Platters spin at a constant rate. Moreover, the amount of data recorded on tracks may increase from the inner-most track to the outer-most track (e.g., in the case of zoned disks). The storage space of each track is divided into several disk blocks, each consisting of a sequence of physically contiguous sectors. Each platter is associated with a read/write head that is attached to a common actuator. A cylinder is a stack of tracks at one actuator position.

In such an environment, the access time of a disk block consists of three components: *seek time*, *rotational latency*, and *data transfer time*. Seek time is the time needed to position the disk head on the track containing the desired data, and is a function of the initial start-up cost to accelerate the disk head as well as the number of tracks that are traversed. Rotational latency, on the other hand, is the time for the desired data to rotate under the head before it can be read or written, and is a function of the angular distance between the current position of the disk head and the location of the desired data, as well as the rate at which platters spin. Once the disk head is positioned at the desired disk block, the time to retrieve its contents is referred to as the data transfer time, and is a function of the disk block size and data transfer rate of the disk.

The placement of data blocks on disks in storage servers is generally governed by either contiguous, random, or constrained placement policy. Contiguous placement policy requires that all blocks belonging to an image be placed together on the disk. This ensures that once the disk head is positioned at the beginning of an image, all of its blocks can

be retrieved without incurring any seek or rotational latency. Unfortunately, the contiguous placement policy results in disk fragmentation in environments with frequent image creations and deletions. Hence, contiguous placement is well-suited for read-only systems (such as compact discs, CLVs, etc.), but is less desirable for a dynamic, read-write storage systems.

Storage servers for read-write systems have traditionally employed random placement of blocks belonging to an image on disk [21, 39]. This placement scheme does not impose any restrictions on the relative placement on the disks of blocks belonging to a single image. This approach eliminates disk fragmentation, albeit at the expense of incurring high seek time and rotational latency overhead while accessing an image.

Clearly, the contiguous and random placement models represent two ends of a spectrum; whereas the former does not permit any separation between successive blocks of an image on disk, the latter does not impose any constraints at all. The *constrained* or the *clustered* placement policy is a generalization of these extremes; it requires the blocks to be clustered together such that the maximum seek time and rotational latency incurred while accessing the image is does not exceed a pre-defined threshold.

For the random and the constrained placement policies, the overall disk throughput depends on the total seek time and rotational latency incurred per byte accessed. Hence, to maximize the disk throughput, image servers use as large a block size as possible.

1.2 Multi-disk Placement

Due to the large sizes of images and image sequences (i.e., video streams), most image and video storage servers utilize disk arrays. Disk arrays achieve high performance by servicing multiple I/O requests concurrently, and by utilizing several disks to service a single request in parallel. The performance of a disk array, however, is critically dependent on the distribution of the workload (i.e., the number of blocks to be retrieved from the array) among the disks. The higher the imbalance in the workload distribution, the lower is the throughput of the disk array.

To effectively utilize a disk array, a storage server interleaves the storage of each image or image sequence among the disks in the array. The unit of data interleaving, referred to as a *stripe unit*, denotes the maximum amount of logically contiguous data that is stored on a single disk [9]. Successive stripe units of an object are placed on disks using a round-robin or random allocation algorithm. In either case, the stripe unit size should be chosen such that it achieves high overall disk throughput while minimizing the load imbalance across the disks in the array.

From Images to Multi-resolution Imagery

The placement technique becomes more challenging if the imagery is encoded using a multi-resolution encoding algorithm. In general, multi-resolution imagery consists of multiple layers. Whereas all layers need be retrieved to display the imagery at the highest resolution, only a subset of the layers need to be retrieved for lower resolution displays. To efficiently support the retrieval of such images at different resolutions, the placement algorithm needs to ensure that the server can *access only as much data as needed and no more.* To ensure this property, the placement algorithm should store multi-resolution images such that: (1) each layer is independently accessible, and (2) the seek and rotational latency while accessing any subset of the layers is minimized. Whereas the former requirement can be met by storing layers in separate disk blocks, the latter requirement can be met by storing these disk blocks adjacent on disk. Observe that this placement policy is general, and can be used to interleave any multi-resolution image or video stream on the array.

From Images to Video Streams

Consider a disk-array based video server. If the video streams are compressed using a variable bit-rate (VBR) compression algorithm, then the sizes of frames (or images) will vary. Hence, if the server stores these video streams on disks using fixed size stripe units, then each stripe unit will contain a variable number of frames. On the other hand, if each stripe unit contains a fixed number of frames (and hence data for a fixed playback duration), then the stripe units will have variable sizes. Thus, depending on the striping policy, retrieving a fixed number of frames will require the server to access a fixed number of variable-size blocks or a variable number of fixed-size blocks [4, 26, 45].

Due to the periodic nature of video playback, most video servers service clients by proceeding in terms of periodic rounds. During each round, the server retrieves a fixed number of video frames (or images) for each client. To ensure continuous playback, the number of frames accessed for each client during a round must be sufficient to meet its playback requirements. In such an architecture, a server that employs variable-size stripe units (or fixed-time stripe units) accesses a fixed number of stripe units during each round. This uniformity of access, when coupled with the sequential and periodic nature of video retrieval, enables the server to balance load across the disks in the array. This efficiency, albeit, comes at the expense of increased complexity of storage space management. The placement policy that utilizes fixed-size stripe units, on the other hand, simplifies storage space management but yields higher load imbalance across the disks.

1.3 Utilizing Storage Hierarchies

The preceding discussion has focused on fixed disks as the storage medium for image and video servers. This is primarily because disks provide high throughput and low latency relative to other storage media such as tape libraries, optical juke boxes, etc. The startup

latency for devices such as tape libraries is substantial since it requires mechanical loading of the appropriate tape into a reader station. On the other hand, these devices offer very high capacities and a substantially lower storage cost (per megabyte).

In order to construct a cost-effective image and video storage system that provides adequate throughput, it is logical to use a hierarchy of storage devices [10, 18, 23, 30]. There are several possible strategies for managing this storage hierarchy, with different techniques for placement, replacement, etc. In one scenario, a relatively small set of frequently requested images and videos are placed on disks, and the large set of less frequently requested data objects are stored in optical juke boxes or tape libraries. In this storage hierarchy, there are several alternatives for managing the disk system. The most common architecture is the one in which disks are used as a staging area (cache) for the secondary storage devices and the entire image and video files are moved from the secondary storage to the disk. It is then possible to apply traditional cache management techniques to manage the content of the disk array.

For very large-scale servers, it is also possible to use an array of juke boxes or tape readers [30]. In such a system, images and video objects may need to be striped across these tertiary storage devices [7]. Whereas striping can improve I/O throughput by reading from multiple tape drives in parallel, it can also increase contention for drives (since each request accesses all drives). Studies have shown that such systems must carefully balance these tradeoffs by choosing an appropriate degree of striping for a given workload [7, 8].

2 FAULT TOLERANCE

Most image and video servers are based on large disk arrays, and hence the ability to tolerate disk failures is central to the design of such servers. The design of fault-tolerant storage systems has been a topic of much research and development over the past decade [3, 22]. In most of these systems, fault-tolerance is achieved either by *disk mirroring* [2] or *parity encoding* [11, 27]. Disk mirroring achieves fault-tolerance by duplicating data on separate disks (and thereby incurs 100% storage space overhead). Parity encoding, on the other hand, reduces the overhead considerably by employing error correcting codes. For instance, in a RAID level 5 disk array consisting of D disks, parity computed over data stored across $(D-1)$ disks is stored on another disk (e.g., the left-symmetric parity assignment shown in Figure 2(a)) [12, 19, 27]. In such architectures, if one of the disks fails, the data on the failed disk is recovered by taking an exclusive-or operation on the data and parity blocks stored on the surviving disks. That is, each user access to a block on the failed disk causes one request to be sent to each of the surviving disks. Thus, if the system is load balanced prior to disk failure, the surviving disks would observe at least twice as many requests in the presence of a failure [15].

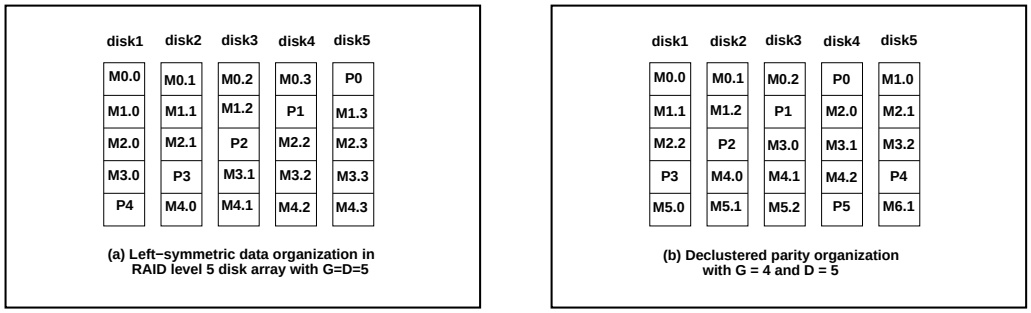


Figure 2 Left-symmetric and declustered parity organizations for RAID architecture. $M_{i.j}$ and P_i denote data and parity blocks, respectively, and $P_i = M_{i.0} \oplus M_{i.1} \cdots \oplus M_{i.(G-2)}$

The *declustered parity* disk array organization [14, 24, 25] addresses this problem by trading some of the array's capacity for improved performance in the presence of disk failures. Specifically, it requires that each parity block protect some smaller number of data blocks (say $(G-1)$). By appropriately distributing the parity information across all the D disks in the array, such a policy ensures that each surviving disk would see an on-the-fly reconstruction load increase of $(G-1)/(D-1)$ instead of $(D-1)/(D-1) = 100\%$ (see Figure 2(b)).

Parity-based failure recovery techniques treat data stored on the array as an uninterpreted sequence of bits and do not exploit any of its characteristics. Furthermore, these techniques assume that perfect reconstruction of the data stored on the failed disk is required for all applications. A storage server can lower substantially the failure recovery overhead by exploiting the semantics of the stored data. For instance, instead of perfectly recovering image data stored on the failed disk using error-correcting codes, a server can exploit human perceptual tolerances and the inherent redundancies in images to *approximately* reconstruct lost image data. In such a server, each image is partitioned into sub-images and if the sub-images are stored on different disks, then a single disk failure will result in the loss of fractions of several images. In the simplest case, if the sub-images are created in the pixel domain (i.e., prior to compression) such that none of the immediate neighbors of a pixel in the image belong to the same sub-image, then even in the presence of a single disk failure, all the neighbors of the lost pixels will be available. In this case, the high degree of correlation between neighboring pixels will make it possible to reconstruct a reasonable approximation of the original image. Moreover, no additional information will have to be retrieved from any of the surviving disks for recovery.

Although conceptually elegant, such *pre-compression image partitioning* techniques significantly reduce the correlation between the pixels assigned to the same sub-image, and hence adversely affect image compression efficiency [29, 40]. The resultant increase in the bit-rate requirement may impose higher load on each disk in the array even during the fault-free state, thereby reducing the number of video streams that can be simultaneously

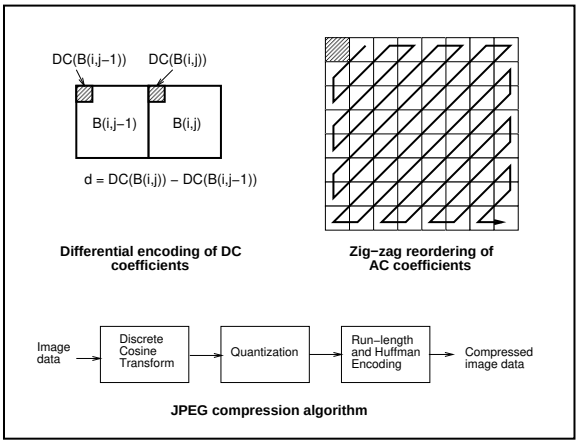


Figure 3 JPEG compression algorithm

retrieved from the server. In what follows, we first present a loss-resilient JPEG (LRJ) compression algorithm that addresses the limitations of the pre-compression image partitioning technique, and then integrate it with placement techniques for disk arrays to derive a self-recovering disk array architecture for images and video streams.

2.1 Loss-Resilient JPEG (LRJ) Algorithm

Since human perception is less sensitive to high frequency components of the spectral energy in an image, most compression algorithms transform images into the frequency domain so as to separate low and high frequency components. For instance, the JPEG compression standard fragments image data into a sequence of 8x8 pixel blocks and transform them into the frequency domain using discrete cosine transform (DCT). DCT uncorrelates each pixel block into an 8x8 array of coefficients such that most of the spectral energy is packed in the fewest number of low frequency coefficients. Whereas the lowest frequency coefficient (referred to as the *DC coefficient*) captures the average brightness of the spatial block, the remaining set of 63 coefficients (referred to as the *AC coefficients*) capture the details within the 8x8 pixel block. The DC coefficients of successive blocks are difference encoded independent of the AC coefficients. Within each block, the AC coefficients are quantized to remove high frequency components, scanned in a zig-zag manner to obtain an approximate ordering from lowest to highest frequency, and finally run length and entropy encoded. Figure 3 depicts the main steps involved in the JPEG compression algorithm [28].

The loss-resilient JPEG (LRJ) algorithm, that we present in this section, is an enhancement of the JPEG compression algorithm, and is motivated by the following two observations:

- Since the DC coefficients capture the average brightness of each 8x8 pixel block and since the average brightness of pixels gradually changes across most images, the DC coefficients of neighboring 8x8 pixel blocks are correlated. Consequently, the value of DC coefficient of a block can be reasonably approximated by extrapolating from the DC coefficients of the neighboring blocks.

  To formally capture this observation, consider an image containing NROW $\times$ NCOL of 8x8 pixel blocks. Let us define the *8-neighborhood* of a block at location (x, y) (denoted by $B(x, y)$) as the set:

  $$\mathcal{N}_8(B(x,y)) = \{B(i,j) \mid |x-i| \leq 1 \text{ OR} |y-j| \leq 1\}$$

  Then, the DC coefficient of the $B(x, y)$ can be approximated as:

  $$\text{DC}_{B(x,y)} = \frac{1}{8} * \sum_{B_{(i,j)} \in \mathcal{N}_8(B(x,y))} \text{DC}_{B(i,j)}$$

  where $\text{DC}_{B(i,j)}$ denotes the DC coefficient of block $B(i, j)$.

- Due to the very nature of DCT, the set of AC coefficients yielded for each 8x8 block are uncorrelated. Moreover, since DCT packs the most amount of spectral energy into a few low frequency coefficients, quantizing the the set of AC coefficients (by using a user-defined normalization array) yields many zeroes, especially at higher frequencies. Consequently, recovering a block by simply substituting a zero for each of the lost AC coefficient is generally sufficient to obtain a reasonable approximation of the original image (at least as long as the number of lost coefficients are small and are scattered throughout the block).

Thus, even when parts of a compressed image have been lost, a reasonable recovery is possible if: (1) the image in the frequency domain is partitioned into a set of sub-images such that none of the DC coefficients in the 8-neighborhood of a block belong to the same sub-image, and (2) the AC coefficients of a block are scattered amongst multiple sub-images. Note that this is distinct from the pre-compression image partitioning technique since the sub-images are created in the frequency domain, as opposed to in the pixel domain. In fact, as we shall demonstrate later, it is this feature of the LRJ compression algorithm that enables reasonable failure recovery without incurring any significant degradation in compression efficiency. To clearly define the LRJ compression algorithm, we will first determine the degree of image partitioning (i.e., the number of sub-images that must be created so as to satisfy the above requirement), and then define a method for scattering the AC coefficients.

Determining the Degree of Image Partitioning

It has been shown that a necessary and sufficient condition to ensure that none of the blocks contained in a sub-image are in the 8-neighborhood of each other, the image must

be partitioned into 4 sub-images [44]. Notice that when an image is partitioned into 4 sub-images, each sub-image contains 25% of the image data in the frequency domain. Consequently, if the information contained in a sub-image is not available, the image will have to be reconstructed from the remaining 75% of the data. Since the quality of the reconstructed image is directly dependent on the amount of original image data available for reconstruction, increasing the degree of image partitioning improves the quality of the reconstructed images. However, as we shall point out later, increasing the degree of image partitioning decreases the correlation between the DC coefficients of the blocks assigned to the same sub-image, and thereby deteriorates the compression efficiency. Hence, the degree of image partitioning must be chosen so as to simultaneously optimize the quality of reconstructed image and the compression efficiency.

Scrambling AC Coefficients

To enable better recovery, the AC coefficients of a block should be scattered amongst multiple sub-images. To achieve this objective, the LRJ compression algorithm employs a scrambling technique which when given a set of N blocks of AC coefficients, creates a new set of N blocks such that the AC coefficients from each of the input blocks are equally distributed amongst all of the output blocks.

To precisely describe the scrambling technique, let us denote the set of N blocks of the original image as O_i, $i \in [0, N-1]$, and the new set of N blocks created as $\widehat{O_i}$. Assuming that the AC coefficients are numbered from left-to-right in a row-major order and that $\mathrm{AC}^k_{O_i}$ denotes the k^{th} AC coefficient ($k \in [1, 63]$) of block O_i, then the scrambling operation assigns $\mathrm{AC}^k_{O_i}$ to be the k^{th} coefficient of block $\widehat{O_j}$ where $j = (i + k) \bmod N$. Thus, each resulting block contains exactly $\frac{64}{N}$ coefficients of each of the original blocks. Specifically, one of the blocks contain the DC coefficient and $\left(\frac{64}{N} - 1\right)$ AC coefficients, and all the remaining $(N-1)$ blocks contain $\left(\frac{64}{N}\right)$ AC coefficients. Figure 4 illustrates the operation of scrambling AC coefficients of four 4x4 blocks.

Combining Image Partitioning and Scrambling Techniques

Given that each image in a video stream is being partitioned into $N (N \geq 4)$ sub-images, the LRJ compression algorithm involves two steps: (1) select a set of N blocks from the original image, and scramble the the set of AC coefficients within the blocks to create a new set of N blocks; and (2) assign the resulting blocks to sub-images (one block per sub-image) such that none of the DC coefficients contained in a sub-image belong to blocks that are in the 8-neighborhood of each other. Since $N \geq 4$, the latter objective can be achieved by assigning the scrambled blocks to sub-images in a round-robin manner, and by ensuring that the assignment of the first block from each row is offset by 2 sub-images from the corresponding block in the previous row.

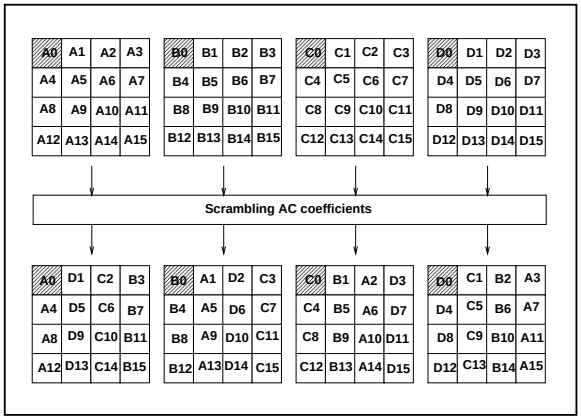


Figure 4 Scrambling AC coefficients. Here A_0, B_0, C_0 and D_0 denote DC coefficients, and $\forall i \in [1, 15] : A_i, B_i, C_i$ and D_i represent AC coefficients.

Notice that since each invokation of the scrambling technique requires N blocks from the same row of the image, the number of blocks within each row of the original image must be an integral multiple of N. In the event that this condition is not met for the original image, each row of blocks may required to be padded with additional "zero" blocks (i.e., blocks with the DC as well as all of the AC coefficients set to zero). Since a sequence of zeroes can be efficiently run-length and huffman encoded, the addition of such zero blocks does not yield any noticeable increase in the size of the compressed image.

Once all the blocks within the image have been processed, each of the N sub-images can be independently encoded. Specifically, the DC coefficients within each sub-image are encoded with a loseless DPCM scheme using the DC coefficient from the previous block as a 1-D predictor. Similarly, the 2-D array of 63 AC coefficients within each block is formatted as a 1-D vector using a zigzag reordering, and then run-length and huffman encoded. Note that the huffman tables utilized for this purpose can either be optimized over each individual sub-image or over the entire image. Whereas the former approach will require a huffman table to be stored with each sub-image, the latter requires a single huffman table to be stored for an entire image. However, in such a scenario, to guarantee the availability of the huffman table even when one or more of the sub-images are not available, it must be replicated across multiple sub-images.

At the time of decompression, once each sub-image has been run-length and huffman decoded, the LRJ algorithm employs an unscrambler to recover blocks of the original image from the corresponding blocks of the sub-images. In the event that the information contained in a sub-image is not available, the unscrambling module also performs a predictive reconstruction of the lost DC coefficients from the DC coefficients of the neighboring 8x8 blocks. Lost AC coefficients, on the other hand, are replaced by zeroes. Since the scrambler module employed by the encoder ensures that each block within a sub-image contains

coefficients from several blocks of the original image, the artifacts yielded by such a recovery mechanism are dispersed over the entire reconstructed image, thereby significantly improving the visual quality of the image.

Observe that since successive blocks within a sub-image do not belong to the 8-neighborhood of each other, the correlation between their DC coefficients is smaller than the neighboring DC coefficients in the original image. The reduced correlation diminishes the efficiency of DPCM encoding of DC coefficients, and hence increases the total size of the compressed image (as compared to the corresponding JPEG image). The effect of scattering AC coefficients of a block across several sub-images, on the other hand, does not have any significant impact on the compressed image size. This is because, due to the very nature of DCT, AC coefficients are uncorrelated. Moreover, a large fraction of the quantized AC coefficients are zeroes. Since the scattering algorithm does not alter the relative position of an AC coefficient within the zig-zag ordering, the effect of scattering on the efficiency of run-length and huffman encoding is minimal. Thus, the increase in compressed image size yielded by the LRJ algorithm can be mostly attributed to the need for replicating huffman tables and the uncorrelation of successive DC coefficients.

Finally, the failure recovery techniques described in this section can be extended image sequences as well. Since compression algorithms for image sequences (e.g., MPEG) exploit both spatial and temporal redundancies, doing so will require techniques that recover lost spatial information (DCT blocks) as well as temporal information such motion vectors [35].

2.2 Self-Recovering Array of Disks (SRAD)

The LRJ compression algorithm, when applied to a sequence of images constituting a video stream yields N sequences of sub-images. We refer to each such sequence as *sub-stream*. To ensure effective recovery, the server must organize each of the sub-streams on the array such that the disks over which the sub-streams are striped do not overlap (i.e., even in the presence of a single disk failure, at least $(N-1)$ sub-streams are available). We refer to a disk array architecture that employs such placement methodologies as a *Self-Recovering Array of Disks (SRAD)*. A careful analysis of this process of recovering from disk failure illustrates the following salient characteristics of the SRAD architecture:

- Since each image in the video stream is reconstructed by extrapolating information retrieved from the surviving disks, the failure recovery process does not impose any additional load on the disk array.
- Since the recovery of lost image data is integrated with the decompression algorithm, the reconstruction process is carried out at client sites. This is an important departure from the conventional RAID technology — distributing the functionality of

failure recovery to client sites will significantly enhance the scalability of multi-disk multimedia servers.

- Since the recovery process only exploits the inherent redundancy in imagery, client sites will be able to reconstruct a video stream even in the presence of multiple disk failures. The quality of the reconstructed image, albeit, will degrade with increase in the number of simultaneously failed disks (i.e., SRAD supports graceful degradation in the image quality with increase in the number of failed disks).
- Since the cause of the data loss is irrelevant to the recovery algorithm, the unscrambling algorithms in LRJ can be adapted to mask packet losses due to network congestion as well. [1]

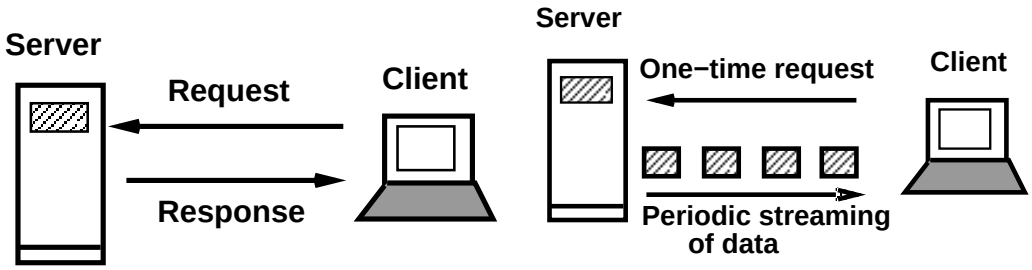
Observe also that although the quality of the recovered image in the presence of a single disk failure is acceptable for most applications, to prevent any accumulation of errors across multiple disk failures, the server must also maintain parity information to perfectly recover the contents of the failed disk onto a spare disk. In such a scenario, on-line reconstruction onto a spare disk can proceed simply by issuing low-priority read requests to access media blocks from each of the surviving disks [15]. By assigning low priority to each read request issued by the on-line reconstruction process, the server can ensure that the performance guarantees provided to all the clients are met even in the presence of disk failures.

3 RETRIEVAL TECHNIQUES

Traditionally, storage servers have employed two fundamentally different architectures for the storage and retrieval of images and image sequences. Storage servers that employ the *client-pull* architecture retrieve data from disks only in response to an explicit client request. Servers that employ the *server-push* or *streaming* architecture, on the other hand, periodically retrieve and transmit data to clients without explicit client requests. Figure 5 illustrates these two architectures. From the perspective of retrieving images and image sequences, both architectures have their advantages and disadvantages.

Due to its request-response nature, the client-pull architecture is inherently suitable for one-time requests for an image or a portion of an image (e.g., the low-resolution component of a multi-resolution image). Adapting the client-pull architecture for retrieving image sequences, however, is difficult. This is because maintaining continuity of playback for an image sequence requires that retrieval requests be issued sufficiently in advance of the playback instant. To do so, applications must estimate the response time of the server and issue requests appropriately. Since the response time varies dynamically depending on the server and the network load, client-pull based applications that access image sequences are non-trivial to develop [37]. Alternatively, rather than estimating the response

[1] Several techniques have been proposed which scramble media streams prior to network transmission to enable approximate reconstruction in case of packet losses [6, 29].



(a) Client-pull architecture (b) Server-push architecture

Figure 5 Client-pull and server-push architectures for retrieving data.

time prior to each request, a client can issue requests based on the worst case response time; however, such a strategy can significantly increase buffer space requirements at the client. The server-push architecture does not suffer from these disadvantages, and hence, is better suited for retrieving image sequences. In such an architecture, the server exploits the sequential nature of data playback by servicing clients in periodic *rounds*. In each round, the server determines the amount of data that needs to be retrieved for each client and issues read requests for these clients. Data retrieved in a round is buffered and transmitted to clients in the next round. Due to the round-based nature of data retrieval, clients need not send periodic requests for data retrieval, and hence, this architecture is suitable for efficiently retrieving image sequences. The server-push architecture, however, is inappropriate for aperiodic or one-time requests for image data.

Whereas conventional applications are best-effort in nature, certain image applications need performance guarantees from the storage server. For instance, to maintain continuity of playback for image sequences, a storage server must guarantee that it will retrieve and transmit images at a certain rate. Retrieval of images for applications such as virtual reality impose bounds on the server response time. To provide performance guarantees to such applications, a storage server must employ resource reservation techniques (also referred to as *admission control* algorithms). Typically, such techniques: (i) determine the resource requirements for each new client, (ii) admit the client only if the resource available at the server are sufficient to meet its resource requirements. Admission control algorithms can provide either deterministic or statistical guarantees, depending on whether they reserve resources based on the worst case load or a probability distribution of the load [43, 42]. Regardless of the nature of guarantees provided by admission control algorithms, designing such algorithms for the server-push architecture is simple— the sequential and periodic nature of data retrieval enables the server to accurately predict the data rate requirements of each client and reserve resources appropriately. Designing admission control algorithms for client-pull architectures, on the other hand, is challenging (since the aperiodic nature of client requests makes it difficult to determine and characterize the resource requirements of a client).

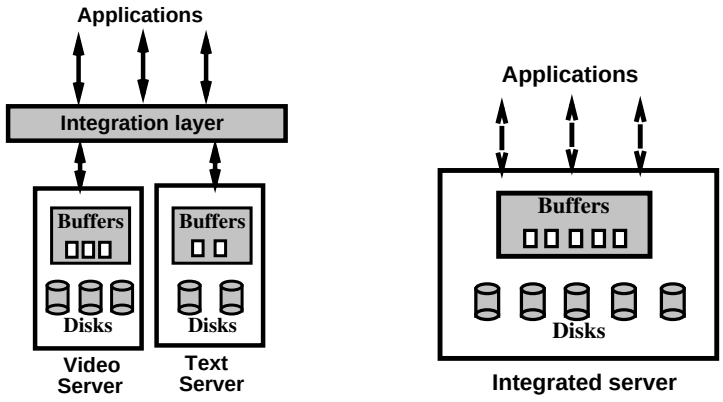
A fundamental advantage of the client-pull architecture is that it is inherently suitable for supporting adaptive applications with dynamically changing resource availability. This is because with changes in resource availability, the client can alter its request rate to keep pace with the server. For instance, if the load on CPU increases or the response time estimates indicate that the network is congested, an adaptive application can reduce its bandwidth requirements by requesting only a subset of data, or by requesting the delivery of a lower resolution version of the same object. The server-push architecture, on the other hand, does not assume any feedback from the clients. In fact, admission of a client for service constitutes a "contract" between the server and the client: the server guarantees that it will access and transmit sufficient information during each round so as to meet the data rate requirements of the client; and the client guarantees that it will keep pace with the server by consuming all the data transmitted by a server during a round within a round duration. Any change in resource availability or client requirements necessitates a renegotiation of this contract, making the design of the server as well as adaptive applications more complex.

Finally, regardless of the architecture, all storage servers employ disk scheduling algorithms to improve I/O performance through intelligent scheduling of disk requests. Disk scheduling algorithms can be broadly divided into two classes: those optimized to service best-effort requests (e.g., SCAN, Shortest Access Time First (SATF) [38, 17, 33]), and those optimized to service real-time requests (e.g., Earliest Deadline SCAN, Feasible Deadline SCAN [1, 5, 31, 46]). Whereas the former class of algorithms attempts to improve the performance of best-effort requests by reducing disk seek and rotational latency overheads, the latter class attempts to meet deadlines of real-time requests while reducing disk overheads. In homogeneous environments, depending on the application requirements, a storage server can employ an scheduling algorithm from one of these two classes. Neither class of algorithms is appropriate for heterogeneous computing environments consisting of a mix of best-effort and real-time applications. For such environments, sophisticated disk schedulers that (i) support multiple application classes simultaneously, (ii) allocate disk bandwidth among classes in a predictable manner, and (iii) align the service provided within each class with application needs [34] are more appropriate.

4 ARCHITECTURAL ISSUES

Sections 1, 2 and 3 examined placement, failure recovery, and retrieval techniques that are suitable for image and video servers. In this section, we examine how all of these techniques can be incorporated into a general-purpose file system and the implications of doing so on the file system architecture.

There are two methodologies for designing file systems that support simultaneously support heterogeneous data types and applications: (i) a *partitioned* architecture that consists of multiple component file servers, each optimized for a particular data type (and glued



(a) Partitioned file server (b) Integrated file server

Figure 6 Partitioned and integrated file servers supporting text/images and video applications. The partitioned architecture divides the server resources among multiple component file systems, and employs an integration layer that provides a uniform mechanism to access files. The integrated architecture employs a single server that multiplexes all the resources among multiple application classes.

together by an integration layer that provides a uniform interface to access these files); and (ii) an *integrated* architecture that consists of a single file server that stores all data types. Figure 6 illustrates these architectures.

Since techniques for building file servers optimized for a single data type are well known [21, 41], partitioned file systems are easy to design and implement. In such file systems, resources (disks, buffers) are statically partitioned among component file servers. This causes requests accessing different component servers to access mutually exclusive set of resources, thereby preventing interference between user requests (e.g., servicing best-effort requests does not violate deadlines of real-time requests). The partitioned architecture, however, has the following limitations:

- Static partitioning of resources in such servers is typically governed by the expected workload on each component server. If the observed workload deviates significantly from the expected, then repartitioning of resources may be necessary. Repartitioning of resources such as disks and buffers is tedious and may require the system to be taken off-line [16]. An alternative to repartitioning is to add new resources (e.g., disks) to the server, which causes resources in under-utilized partitions to be wasted.

- The storage space requirements of files stored on a component file server can be significantly different from their bandwidth requirements. In such a scenario, allocation of disks to a component server will be governed by the maximum of the two values. This can lead to under-utilization of either storage space or disk bandwidth on the server.

The main feature of the integrated file system architecture is dynamic resource allocation: storage space, disk bandwidth, and buffer space are allocated to data types on demand; static partitioning of these resources is not required. This feature has several benefits. First, by co-locating a set of files with large storage space but small bandwidth requirements with another set of files with small storage space but large bandwidth requirements, this architecture yields better resource utilization. Second, since resources are allocated on demand, it can easily accommodate dynamic changes in access patterns. Finally, since all the resources are shared by all applications, more resources are available to service each request, which in turn improves the performance.

Such improvements in the resource utilization, however, come at the expense of increased complexity in the design of the file system. This is because, wide disparity in the applications/data requirements dictate that a single storage management technique or policy is often inadequate to meet the requirements of all applications and data types. For instance, the best-effort service model, although adequate for many applications, is clearly unsuitable for applications and data types that impose timeliness constraints. Consequently, a key principle in designing integrated file systems is that they should *enable the coexistence of multiple data type-specific and application-specific policies.* For instance, to align the service its provides to the needs of individual data types, an integrated file system should enable the coexistence of data type-specific policies for common file system tasks such as placement, meta data management, caching, and failure recovery. Similarly, it may need to support multiple retrieval architectures —such as client-pull and server-push — as well as multiple service classes — such as interactive best-effort, soft real-time, and throughput-intensive best-effort. Enabling the co-existence of such diverse techniques requires the development of mechanisms that achieve high resource utilization through sharing while isolating the service exported to the different application classes [36].

Figure 7 depicts a two-layer architecture for implementing such an integrated file system. This architecture separates data type- and application-independent mechanisms from specific policies; and implements these mechanisms and policies in separate layers. The lower layer implements core file system mechanisms that are required for all applications and data types. The upper layer then employs these mechanisms to instantiate specific polices, each tailored for a particular data type or an application class; multiple policies can coexist since the mechanism in the lower layer are designed to multiplex resources among various classes. For instance, in case of placement, the lower layer may consist of a storage manager that can allocate a disk block of any size, while the upper layer uses the storage manager to instantiate different placement policies for images, images sequences, and textual data (each policy can tailor the block size used to store files based on the requirements of the data type). Observe that, such a two layer architecture is inherently extensible—implementing a powerful set of mechanisms in the lower layer enables new application and data types to be supported by adding appropriate policies to the upper layer.

Whereas the storage servers employing the partitioned architecture were common in the early nineties (due to the concurrent and independent development of conventional file

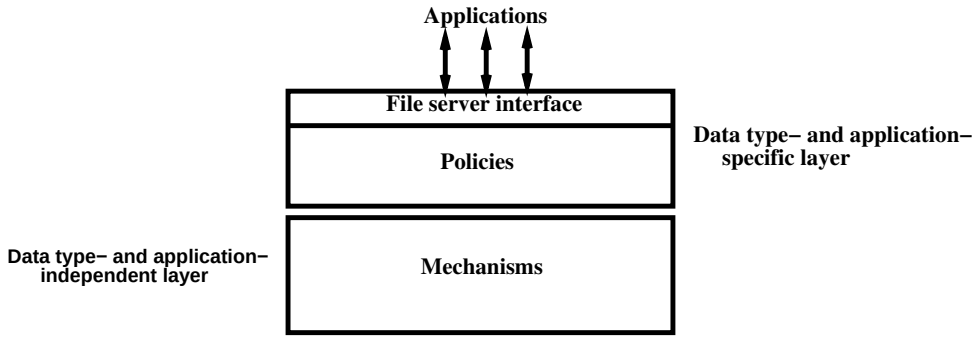


Figure 7 A two layer file system architecture that separates mechanisms from policies.

systems and video-on-demand servers), integrated file systems have received significant attention recently. Several research projects such as Symphony [36], Fellini [20] and Nemesis [32], as well as commercial efforts such as IBM's Tiger Shark [13] and SGI's XFS [16] have resulted in storage servers employing the integrated architecture.

5 CONCLUDING REMARKS

Emerging multimedia applications differ from conventional distributed applications in the type of data they store, transmit, and process; and also in the requirements they impose on the networks and operating systems. In this chapter, we focussed on the problem of designing storage servers that can meet the requirements of these emerging applications. We described the techniques for designing a storage server for digital imagery and video streams and then examined the architectural issues in incorporating these techniques into a general purpose file system.

We conclude by noting that a Quality-of-Service (QoS) aware file system is just one piece of the end-to-end infrastructure required to support emerging distributed applications; to provide to the applications the service they require, such file systems will need to be integrated with appropriate networks and operating systems.

REFERENCES

[1] R K. Abbott and H. Garcia-Molina. Scheduling I/O Requests with Deadlines: A Performance Evaluation. In *Proceedings of RTSS*, pages 113–124, December 1990.

[2] D. Bitton and J. Gray. Disk Shadowing. In *Proceedings of the 14th Conference on Very Large Databases*, pages 331–338, 1988.

[3] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP Parallel RAID Architecture. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 52–63, May 1993.

[4] E. Chang and A. Zakhor. Scalable Video Placement on Parallel Disk Arrays. In *Proceedings of IS&T/SPIE International Symposium on Electronic Imaging: Science and Technology, San Jose*, February 1994.

[5] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. *Journal of Real-Time Systems*, 3:307–336, 1991.

[6] J. M. Danskin, G. M. Davies, and X. Song. Fast Lossy Internet Image Transmission. In *Proceedings of the Third ACM Conference on Multimedia, San Francisco, California*, pages 321–332, November 1995.

[7] A. Drapeau. *Striped Tertiary Storage Systems: Performance and reliability.* PhD thesis, University of California at Berkeley, December 1993.

[8] A L. Drapeau and R H. Katz. Striping in Large Tape Libraries. In *Proceedings of Supercomputing'93*, November 1993.

[9] H. Garcia-Molina and K. Salem. Disk Striping. *International Conference on Data Engineering*, pages 336–342, February 1986.

[10] Robert M. Geist and Kishor S. Trivedi. Optimal Design of Multilevel Storage Hierarchies. C-31:249–260, March 1982.

[11] G Gibson and D Patterson. Designing Disk Arrays For High Data Reliability. *Journal of Parallel and Distributed Computing*, pages 4–27, January 1993.

[12] J. Gray, B. Horst, and M. Walker. Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput. In *Proceedings of the 16th Very Large Data Bases Conference*, pages 148–160, 1990.

[13] R. Haskin. Tiger Shark–A Scalable File System for Multimedia. *IBM Journal of Research and Development*, 42(2):185–197, March 1998.

[14] M. Holland and G. Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 23–35, October 1992.

[15] M. Holland, G. Gibson, and D. Siewiorek. Fast, On-line Recovery in Redundant Disk Arrays. In *Proceedings of the 23rd International Symposium on Fault Tolerant Computing*, pages 422–431, 1993.

[16] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit File System with Guaranteed Rate I/O. Technical report, Silicon Graphics, Inc, Available online as http://www.sgi.com/Technology/xfs-whitepaper.html, 1996.

[17] D M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical report, Hewlett Packard Labs, February 1991.

[18] B K.Hillyer and A. Silberschatz. On the Modeling and Performance Characteristics of a Serpentine Tape Drive. In *Proceedings of ACM Sigmetrics Conference, Philadelphia, PA*, pages 170–179, May 1996.

[19] E.K. Lee and R. Katz. Performance Consequences of Parity Placement in Disk Arrays. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–199, 1991.

[20] C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz. The Fellini Multimedia Storage Server. *Multimedia Information Storage and Management , Editor S. M. Chung, Kluwer Academic Publishers*, 1996.

[21] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[22] J. Menon and J. Cortney. The Architecture of a Fault-Tolerant Cached RAID Controller. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 76–86, May 1993.

[23] J. Menon and K. Treiber. Daisy: Virtual Disk Hierarchical Storage Manager. 25(3):37–44, December 1997.

[24] A. Merchant and P. S. Yu. Design and Modeling of Clustered RAID. In *Proceedings of the International Symposium on Fault Tolerant Computing*, pages 140–149, 1992.

[25] R R. Muntz and J C.S. Lui. Performance Analysis of Disk Arrays Under Failure. In *Proceedings of the 16th Very Large Data Bases Conference*, pages 162–173, 1990.

[26] S. Paek, P. Bocheck, and S. F. Chang. Scalable MPEG2 Video Servers with Heterogeneous QoS on Parallel Disk Arrays. In *Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video*, April 1995.

[27] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Array of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD'88*, pages 109–116, June 1988.

[28] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard.* Van Nostrand Reinhold, 1993.

[29] E. J. Posnak, S. P. Gallindo, A. P. Stephens, and H. M. Vin. Techniques for Resilient Transmission of JPEG Video Streams. In *Proceedings of Multimedia Computing and Networking, San Jose, CA*, pages 243–252, February 1995.

[30] S. Ranade, editor. *Jukebox and Robotic Tape Libraries for Mass Storage.* Meckler Publishing, 1992.

[31] A.L. Narasimha Reddy and J. Wyllie. Disk Scheduling in Multimedia I/O System. In *Proceedings of ACM Multimedia'93, Anaheim, CA*, pages 225–234, August 1993.

[32] Timothy Roscoe. *The Structure of a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, April 1995. Available as Technical Report No. 376.

[33] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of the 1990 Winter USENIX Conference, Washington, D.C.*, pages 313–323, Jan 1990.

[34] P Shenoy and H M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of ACM SIGMETRICS Conference, Madison, WI*, pages 44–55, June 1998.

[35] P. Shenoy and H M. Vin. Failure Recovery Algorithms For Multimedia Servers. *ACM/Springer Multimedia Systems Journal (to appear)*, 1999.

[36] P J. Shenoy, P. Goyal, S S. Rao, and H M. Vin. Symphony: An Integrated Multimedia File System. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN'98), San Jose, CA*, pages 124–138, January 1998.

[37] S.S.Rao, H.M.Vin, and A. Tarafdar. Comparative Evaluation of Server-push and Client-pull Architectures for Multimedia Servers. In *Proceedings of NOSSDAV'96*, pages 45–48, April 1996.

[38] T. Teorey and T. B. Pinkerton. A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM*, 15(3):177–184, March 1972.

[39] F A. Tobagi, J Pang, R Baird, and M Gang. Streaming RAID – A Disk Array Management System For Video Files. In *Proceedings of ACM Multimedia '93, Anaheim, CA*, pages 393–400, 1993.

[40] C.J. Turner and L.L. Peterson. Image Transfer: An End to End Design. In *Proceedings of ACM SIGCOMM'92, Baltimore*, pages 258–268, August 1992.

[41] M. Vernick, C. Venkatramini, and T. Chiueh. Adventures in Building the Stony Brook Video Server. In *Proceedings of ACM Multimedia'96*, 1996.

[42] H. M. Vin, A. Goyal, and P. Goyal. Algorithms for Designing Large-Scale Multimedia Servers. *Computer Communications*, 18(3):192–203, March 1995.

[43] H. M. Vin, P. Goyal, A. Goyal, and A. Goyal. A Statistical Admission Control Algorithm for Multimedia Servers. In *Proceedings of the ACM Multimedia'94, San Francisco*, pages 33–40, October 1994.

[44] H. M. Vin, P. J. Shenoy, and S. Rao. Efficient Failure Recovery in Multi-Disk Multimedia Servers. In *Proceedings of the 25th International Symposium on Fault Tolerant Computing Systems, Pasadena, CA*, pages 12–21, June 1995.

[45] H.M. Vin, S.S. Rao, and P. Goyal. Optimizing the Placement of Multimedia Objects on Disk Arrays. In *Proceedings of the Second IEEE International Conference on Multimedia Computing and Systems, Washington, D.C.*, pages 158–165, May 1995.

[46] P. Yu, M.S. Chen, and D.D. Kandlur. Design and Analysis of a Grouped Sweeping Scheme for Multimedia Storage Management. *Proceedings of Third International Workshop on Network and Operating System Support for Digital Audio and Video, San Diego*, pages 38–49, November 1992.