Huygens File Server: Storage Architecture

Peter Bosch
Sape Mullender
Tage Stabell-Kulø

Pegasus paper 93–3
August 1993

University of Cambridge Computer Laboratory

University of Twente Faculty of Computer Science

Pegasus, ESPRIT *BRA 6865, is a project of the University of Cambridge and the University of Twente.*
The project is aimed at the design of an operating systems architecture for scalable distributed multimedia systems and the development of a validating prototype, design and implementation of a distributed complex-object service and a global name service, mechanisms for the creation, communication, and rendering of fully digital multimedia documents in real time and in a distributed fashion (with support for full-screen digital motion video and digital hi-fi stereophonic sound), and the design and implementation of an application for the system — a digital TV director.

The Pegasus Papers can be obtained from the Pegasus Secretariat, Faculty of Computer Science, Vakgroep SPA, P.O. Box 217, 7500 AE Enschede, Netherlands.

They can also be obtained through the World-Wide Web:
`http://www.pegasus.esprit.ec.org/default.html`

Pegasus Paper 93–3

Huygens File Server: Storage Architecture

Peter Bosch **Sape Mullender** **Tage Stabell-Kulø**

August 1993

Abstract

The Huygens file server is a high-performance file server which is able to deliver multimedia data in a timely manner while also providing clients with ordinary "Unix" like file I/O. The file server integrates client machines, file servers and tertiary storage servers in the same storage architecture. The client's view on the storage system is a very large file system, while in practice it is built out of a hierarchy of storage devices with a single site semantics protocol.

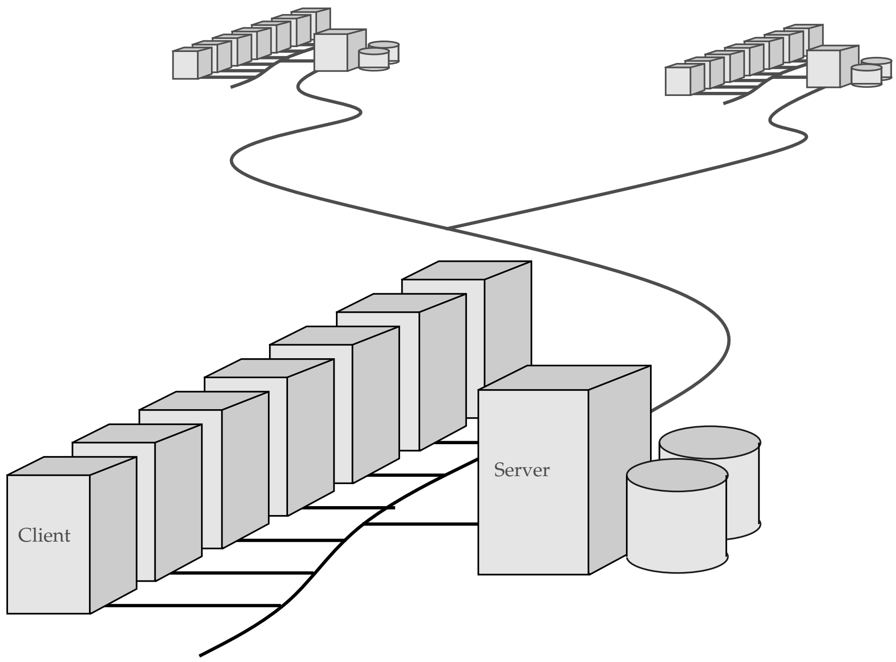

Figure 1. Huygens File System configuration

1 Architecture

The Huygens File Server will be deployed in the setting of the Pegasus project, that of distributed operating systems supporting multimedia applications. The Huygens File Server will have to cater both to the needs of multimedia applications — high data rates with minimal jitter — and those of distributed applications — good consistency semantics, high reliability in the presence of failures.

1.1 Architectural Considerations

HFS will have to be flexible, because the environment in which it is used will be one of constant flux. We cannot predict today what demands will be made of the file system tomorrow.

Figure 1 shows the setting of HFS in the context of the Pegasus system. A local system will have one or more HFS servers, and clients varying from compute servers, workstations, multimedia terminals, to 'networked devices'.

On those client machines that require service of some sophistication, such as caching, there will be an HFS *agent* server which can provide caching, buffering, consistency, crash recovery, etc.

The Huygens File Server is intended to be usable in a wide range of applications in systems varying in size from single machines to large-scale systems.

It is, therefore, vital that the file system can house a variety of storage hierarchies, caching algorithms, and file semantics — a careful separation of mechanism and policy in all aspects of the file system.

This separation, combined with the desire of high performance, also dictates a separation of data and metadata — low levels of the system will provide simple and efficient mechanisms to move data from client memory to secondary and tertiary

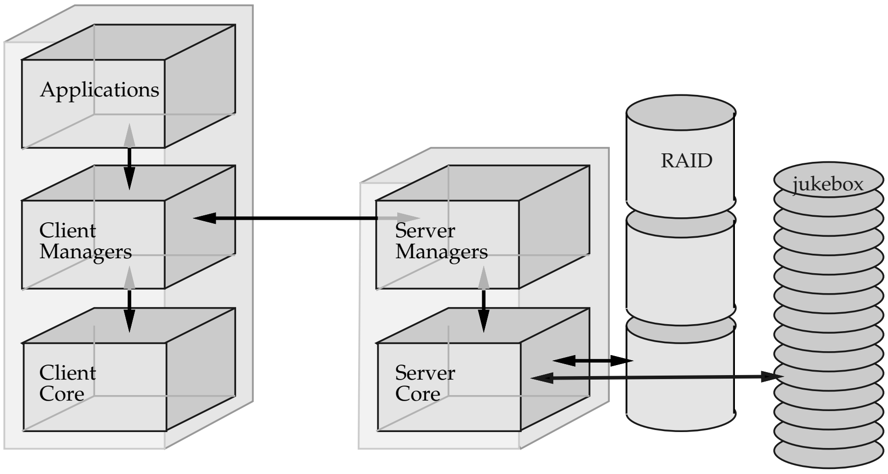

Figure 2. Huygens File System layering

storage, while higher levels use metadata to make informed policy decisions of when to move data where.

In Figure 2, we show the architecture of an HFS configuration. The bottom layer of the file server and the client agent is formed by the 'mechanism layer', a *storage system* to move data to and from storage devices, a *server cache system* and a *client cache system* to manipulate file data in primary memory. Together, these systems form the *core layer* of HFS.

1.2 The Core Layer

The modules in the core layer have interfaces through which *managers* in higher layers can steer their actions. The set of these is extensible so that the storage needs of new, as yet unenvisioned, applications can be catered for. These managers are used to implement persistent objects for a variety of purpose with a range of semantics.

Applications can use managed objects as a basis for implementing typed objects with the rich type systems that modern object-oriented systems demand. They can also use them as ordinary Unix-like files, or even as persistent memory segments.

The unit of storage is the *file*. To the low-level systems, a file is an uninterpreted array of bytes. Pieces of files are moved between client and server and between primary memory and secondary or tertiary storage. Which pieces are moved end when they are moved is decided by the managers of the system.

At the core level, files are identified and located through a fixed-length number, called *P-number*. The location and size of a file is recorded in a data structure called *P-node*.

1.3 Managers and Types

Hierarchical path names can be (and will normally be) provided through a naming service that resides as a manager in a module on top of the core layer. This naming service will use the base-storage services of the core: it will use core files to store directories.

Files containing diretories require different caching strategies and consistency semantics than, say, files containing video. This implies thar they will be controlled by different sets of managers. Files, thus, have a *"type"*, an identifier that specifies the managers directly responsible for controlling their management by the core.

This type must be extensible so that new managers can be added to the system and so that hierarchies of managers can be built. As an example of a fairly complicated hierarchy, consider a multimedia document with text, images, audio and video. At the core level, it will be built up out of a number of files, one or more for the audio, one or more for the video, and so on. Each of these base files will be managed by the manager appropriate for the kind of data stored on it, while there will also be a higher-level manager that controls the multimedia document as a whole.

1.4 Reliability

Storage systems must be reliable. Applications entrust their data to them and must expect to be able to get the data back. We have decided that, as a system-wide minimum level or reliability, HFS must guarantee not to lose data under single failures, or system-wide power failure.

More, precisely, the failure model for HFS is that

- Servers are fail stop — when they fail, they will lose all internal state, including the contents of non-volatile-memory banks;
- Clients exhibit arbitrary failures — it is assumed that system software running on client machines can be arbitrarily and maliciously corrupted;
- Server disks have detectable amnesia failures — they will stop working entirely, which is evidently detectable, or they will fail to read data previously written, which is detected by the disk's CRC;
- All nodes will stop as a consequence of a power failure — non-volatile-memory banks in the servers are assumed to survive power failures (and no other failures).

Data, written by *correct* clients, survives any single failure from the list above. When an application running on an incorrect client writes data, no guarantees can be made, of course.

The techniques for fault tolerance are described in Section 2.1. Roughly, the techniques used consist of using RAID-5 for recovery from disk failures and replication of data in client buffer and server non-volatile memory when data has not yet reached the disks.

1.5 Consistency

Designers of distributed, fault-tolerant file systems are always confronted with the dilemma of choosing between providing full consistency (a read provides the last data written anywhere) and full availability (a file operation can always succeed without having to wait for a failed module to be repaired).

It is fundamentally impossible to do both simultaneously. Some designers chose to increase availability by relaxing the consistency requirements (e.g., Coda [...]), others to maintain consistency at the cost of making the system less available (e.g., ???).

HFS can accommodate both strategies, due to its separation of mechanism and policy. Most of the system-provided cache managers, however, provide full consistency. This is done through the use of *read tokens* and *exclusive tokens*, the possession of which allows a cache manager to assume up-to-dateness of the file for which it has a token. Tokens are revoked by timeout, so that link failures and network partitions cannot cause inconsistencies.

The token manager's algorithm is explained in Section 2.3.1.

1.6 Scale

HFS must be scalable, both in terms of number of servers and amount of storage. We deal with server numbers by dividing the system into *sites*. A site is a collection of servers and clients connected by a local network which functions autonomously.

Most files are accessed at one site only. If such files need to be accessed from another site, then this is done by sending read and write operations to he home site of the file.

There are also files that are regularly accessed in several sites. These files are controlled by a set of managers that we call *replication servers*. Replication servers realize far-flung replication and store replicas as local files. This type of replication is described by Dini and Mullender [1993].

Scaling a file system to terabytes of storage is probably the most difficult task we have set ourselves to solve for HFS. Not only must all datastructures be carefully chosen to scale, but we must also take into consideration that it will no longer be possible to "*fsck* the file system," or make a "level-0 dump" of it.

File-system consistency must be achieved piecemeal and backup must be integrated with the file system itself. To do the latter, we have integrated tertiary storage (probably mostly on juke boxes containing optical media) with the file system itself and we are devising ways of naming and finding files and older versions of files efficiently. Here it greatly helps to have a name server outside the file system's core.

1.7 Performance

HFS will be used in an operating system that supports multimedia applications. Obviously, HFS itself will also have to support multimedia data. Of the media, video is the most demanding at the moment. It is possible that, in the future, more demanding media will show up — three-dimensional video is a candidate.

HFS is designed to keep up with live audio and video. It can store and reproduce data at a constant rate for long periods. A video server and an audio server are layered on the core for this purpose. These servers can interact directly with video and audio devices, streaming data to them, or receiving streamed data from them. They build indices in separate core files that can be used for finding scene changes, fast forwarding, reverse playing, etc.

The data paths through the file system are designed to have higher bandwidth than the network (currently 100 Mbps). Non-continuous-media performance will be further boosted by extensive caching. Caching is not useful for audio and video; first, because one video file alone is often bigger than the cache and, second, because this data does not have to be cached, since there is no point in delivering data at a higher rate than the play-back rate.

In the prototype, HFS will not use bandwidth allocation to guarantee timely per-

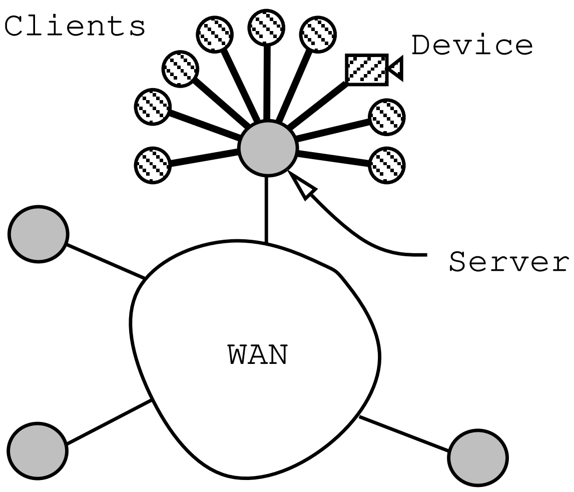

Figure 3. Overall view

formance for audio and video. There is a good possibility that a significant number of video and audio streams can be simultaneously served by HFS without having to do this. In the future, most system components will get bigger and faster, while the bandwidth required for audio and video may even go down slightly as a result of better compression techniques.

In Section 3, we state the performance goals for the HFS prototype system.

2 Design

This section decribes the design of the Huygens File Service. Section 2.1 provides a global overview of the file server's design. Section 2.2 compares our work to other work. Section 2.3 describes in detail the various parts of HFS.

2.1 Overview

In general, we consider a system consisting of a set of cooperating servers. These servers are connected by a WAN. One site may have more than one server. We estimate that the number of servers will be small, in the tens at most. Each server is connected to a LAN. We assume the LAN to have high bandwidth and low latency, at least as good as Ethernet. To the LAN are connected many, possibly more than a hundred, *clients*. These client machines use the file server for permanent storage (in file systems) and, in the case of diskless workstations, also paging. Some clients might not be "machines" running an operating system, cameras for example. In any case, the servers can not trust clients. The overall view is shown in Figure 3, more details can bee seen in Figure 2.

Both server and clients have a layered structure. The client machines will run a *client cache system* to provide basic cache mechanisms, and will run specialised client type managers to provide semantics for the untyped "core" files. The type managers drive the client cache system.

The server machine will run a *server cache system* and a *server storage system*. The server cache system implements a basic caching strategy by using the file server's memory. The server cache will be driven through the server type managers.

The server storage system will implement both file storage on a set of high speed disks and archiving on archival storage of untyped files. The storage system organizes the available disks in a RAID to improve both performance and reliability. The data on the RAID is organized log-structured to improve write performance as is shown by [Ousterhout89, Rosenblum91]. Out-of-band storage is provided for an efficient data path to the RAID. This out-of-band storage is used to retrieve and store multimedia data[1]. Archival storage is integrated in the file service. Every night, backups of the RAID are automatically made on the archival storage. Archived files can be accessed through the storage system.

On top of the server cache and storage systems, type managers are implemented which provide file semantics. Examples of these are the name service and the multimedia service to provide names and timeliness to files.

The storage architecture has been designed not to lose data in case of any single point of failure in the storage system. It is possible that the file system is unavailable during a failure, it is not possible that the file system loses data. This guarantee can be made by employing a number of techniques enforced by HFS Core.

The first is that files are stored on a RAID which is capable of recovering from the loss of any single disk and is described in Section 2.3.3. The second is that files, as long as they are not yet safely on RAID, are replicated over machines. Non volatile memory (NVRAM) is used to ensure that data is not lost due to a power-failure, and the client cache system ensures that data survives even if the file server crashes. This is described in Section 2.3.2.

In order to synchronize file requests from all clients and servers a single site semantics protocol is implemented based on read or write tokens. Read tokens can be shared, write tokens are exclusive. Tokens have timeouts and must be refreshed prior to their expiration. On request of the central token manager, clients must release tokens. This is describved in detail in Section 2.3.1.

Many issues of the Huygens File Service are already researched as seperate research items, but to date, there exists no file service that ties all the issues in one storage architecture. In the next section, we will compare our work to other work.

2.2 Related Work

Many groups are working on file systems and storage architectures. The approaches and aims vary. SWIFT uses more than one network and several machines to obtain higher bandwidth [Cabrera91]. The RAID project aims at providing more bandwidth by adding more disks [Patterson88, Gibson93]. The Andrew File System [Howard88] and Coda [Kistler91] investigate replication and disconnected operation. Existing protocols and solutions are modified to obtain better semantics [Srinivasan89], better reliability [Liskov91] or performance [Macklem91]. Likewise, since most file systems must support non-local operations [Satyanarayanan89, Levy90], caching and thereby consistency, are also of great interest. Prefetching to enhance cache performance is discussed in [Patterson93].

In order to push the file-system throughput closer to that of the disks one uses, log-structured file systems have been introduced. Sprite LFS [Ousterhout89, Rosenblum91] was one of the first systems which built a Unix like file system in a log. The Sprite file

[1] The out-of-band storage is only the mechanism. Timeliness is added by a seperate server.

system present a total file system architecture, including caching [Nelson88], integration of virtual memory [Nelson86] and various storage architectures. The Huygens File Service will use a log-structured data layout on disk, but we will also provide support for different file types (e.g. multimedia files). Many of assumptions made for the Huygens File Service are based on the Sprite file system traces as described in [Baker91]. BSD-LFS [Seltzer92] presents a log-structured file system integrated in BSD Unix 4.4. This work compares the Unix FFS [McKusick84] with an enhanced version of FFS called EFS and a log-structured file system called LFS. It is shown that cleaning a log is an expensive operation which degrades the performance of the log-structured file system to the performance of EFS. EFS shows that storing files consecutively on disk speeds up file read and write requests.

It is attractive to obtain integration of video in file servers with contemporary hardware. The video and audio file server described in [Rangan91] describes a file system that is able to synchronize data streams from the file server, defines an admission control scheme for new multimedia I/O requests and defines the data layout on disk based on hardware definitions. This work does not describe how normal file I/O is integrated in the file system. The difference with our approach is that we relax the data layout requirements. By using large blocks, we can abstract from the precise hardware definitions and use the underlying disks as a "black box". Initially, we will not provide inter-file synchronization in the file server. If, in a later stage we do need such a strategy, we can use *ropes* and *strands* to provide inter-file synchronization.

The network file server design for continuous media as described in [Jardetzky92] contains a thorough description of issues that need to be resolved for a design of such a system. This work defines a *Quality-of-Service (QoS)* as follows: *The goal of QOS is to provide an infrastructure facilitating negotiation between client and server for acceptable service within the capabilities of a given system*. This notion is used in the Huygens File Service. A client is able to negotiate with the file server to determine an optimal throughput which is acceptable for both client and server.

Plan 9 [Pike90] has integrated tertiary storage in the file system. This system automatically archives all the files stored on the file system on an archive and makes the archive available to the users. The Pegasus file-service also makes daily backups of the file system, but we have added a mechanism to *age* data.

2.2.1 Design Considerations

We assume the following about the environment in which the service will be provided:

- The network will have bounded delivery times.
- The file server will be connected to a high speed network as described in [Leslie93, Mullender92].
- An entity may establish an authenticated and secure channel to any other entity, simply by knowing its name.[2]

We will assume that the load generated by the clients using the storage service is according to the usage reported in [Baker91] with the addition of large files containing

[2]Note that this implies the availability of both a naming facility and the necessary inter-machine communication mechanisms.

continuous media (video). Most multimedia files are expected to be read or written consecutive from start to end; as described in [Jardetzky92].

2.3 Design Architecture

The Huygens File Service is divided into a set of modules which can either belong to the HFS Core or to the HFS Managers. The HFS Core implements files as untyped byte streams. All files managed by the HFS Core are subject to a single site semantics protocol. This causes all file operations to be serialized and is explained in detail in Section 2.3.1.

HFS Core consists of the following modules:

The Storage System

The storage system is part of the core layer. It provides basic storage of untyped files on high speed disks as well as archival storage of untyped files on archival storage. The storage system uses a set of large high speed disks organized in a RAID 5 as storage medium and uses a large optical box as archive medium. The storage system provides two methods for file storage: "normal" file storage for Unix like untyped files and *out-of-band* storage for multimedia data. The out-of-band storage is used as an efficient and fast data path to a set of high speed disks connected to the file server machine. Note that out-of-band storage does not imply timeliness. Timeliness is implemented by an HFS Manager called the Multimedia Service. The storage system is described in detail in Section 2.3.3.

The Server Cache System

The server cache system is part of the core layer. It is responsible for maintaining the server memory cache. The server cache system is explained in detail in Section 2.3.4.

The Client Cache System

The client cache system is part of the core layer. It is responsible for maintaining the client memory cache. The client cache system is explained in detail in Section 2.3.5.

The type managers run on top of the HFS Core. They are responsible for providing file semantics based on the untyped "core" files. The following type servers exist or will be implemented:

The HFS Name Service

The HFS Name Service implements a Unix like name space by using the untyped core files as directories. The name service is explained in detail in Section 2.3.6.

The HFS Multimedia Service

The HFS Multimedia Service is able to playback and record multimedia data streams in a truly timely manner. It will use HFS Core out-of-band storage to read and write untyped core files for efficient data transport and it will create and use indexing information to select individual frames in the multimedia stream. The multimedia service is explained in detail in Section 2.3.7.

2.3.1 Single Site Semantics

HFS Core implements a single site semantics protocol for all the files it manages. By doing so, the system ensures that there will be no *conflicting* updates and any read concurrent with a write will see the old *or* the new data. Note that this does not ensure that newly written data cannot be overwritten and thereby lost. In [Lamport85] this is described as a *regular register*.

Single-site semantics are realized by having a token manager running on the file server, and by requiring that the type servers obtain a read or write token before accessing any file. It is then the responsibility of the type server to delegate the token and enforce any sharing it may deem necessary. The prototype assumes whole-file tokens. We will modify this to smaller-granularity tokens if we discover that performance suffers.

The only way a client can get access to a file, is through a type server. The type server obtains a token from the lock manager. Note that the token is not granted to the client, but to the type server. The type server can then access the file on behalf of the client, or inform the client that it has a token. It is in this case the type managers responsibility to enforce at the client level the same single-site semantics enforced between the type servers by the token manager. In other words, the type service can choose whether it wants to multiplex client requests on one lock granted to it by the lock manager, or if it will serialize all requests by leaving all matters concerning locking to the token manager.

Since some type servers, notably the cache, will multiplex requests among clients[3] a protocol is needed to ensure that progress is always possible. We will describe the protocol as it must be realized by a type server which multiplexes requests from clients onto tokens it hold itself. Note that, since communication within the file server is assumed to be reliable and all the type servers are trusted, no special care has to be taken by the token manager when it grants locks.

All tokens have a time-out assigned to them, and in order to keep a token the client must *refresh* it. We assume the time between refresh (the lifetime of a token) is τ seconds. τ will vary with different network technologies and topologies, but will be on the order of a few seconds; remember that we are assuming that the clients are connected to the server by a low latency network, see Section 2.2.1.

A token is *valid* if it protects the file; it is *live* if it was refreshed $t_l < \tau$ seconds ago; and *dead* if not refreshed for $t_d \geq \tau$ seconds. Note that a token that is "live" is always valid and that a token that is "dead" *may* still be valid, at the discretion of the server.

The server must enforces the following behavior:

1. If a client asks for a token, it will be given one. Together with the token it will receive the value of τ.

2. If a client C_1 has a write token, and another client C_2 wants a token (of any type), C_1 is requested by the server to return the lock, together with any new data. The new data is written to the file, and sent to C_2. The token is then granted to C_2. C_1 must request the lock again if the user has more to write. This new request can be sent together with the data when the lock was released.

3. If a client does not answer, the server must wait until the lock has expired before granting it to other clients, a period which may span a full τ seconds.

[3] Which have a Byzantine failure mode.

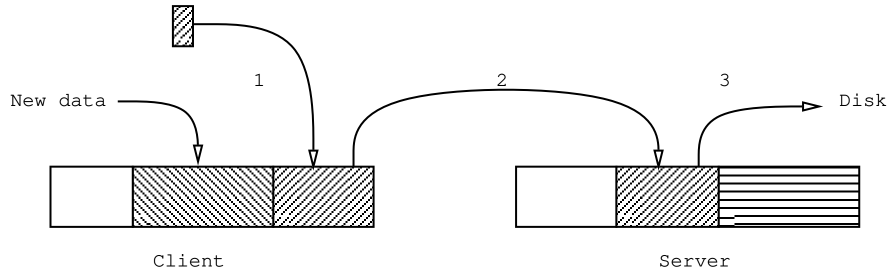

Figure 4. Write Buffer

4. In the case when the request from the server is lost, i.e. no answer is received from the client, but the client is alive and no further errors occur, the client will try to refresh the lock, which is then declined. In this case, it is the client's responsibility to ensure that any data is shipped to the server *before* the lock expires.

If a write token is circulating between n different clients and each client keeps the lock for c seconds, and $nc \geq \tau$, one or more clients will have to inform their users that the write failed. In other words, after τ seconds the client knows its request has not been honored. The length of τ must be set for each configuration. Note that some form of clock synchronization is needed, although in a much less strict way than described in [Schneider87]. It is the clients that need to synchronize with the server. The server will assume that its own clock is correct.

2.3.2 Write Buffer

In general, no single failure will cause data written by any user to be lost. This is ensured by having the data flowing from the user towards the disks entering a *write buffer*, and by adding redundancy to the secondary storage. In this sense we view the failure of one (1) disk as a single failure.

When a user writes data, the following events occur. First it is copied into the clients write-buffer, operation 1 in Figure 4. If a user process has asked for asynchronous I/O it is immediately released, otherwise it is blocked. When the buffer has reached a "high water mark" it is flushed to the server (operation 2) and copied into NVRAM [baker92]. The data is *not* deleted from the client write-buffer, and new data is added to the buffer. Later, when the server write-buffer is full (or an event provoking flushing occurs), the data (together with data from all the other clients) is written to stable storage (operation 3).

After operation 2, the data has arrived in the NVRAM at the server, the following conditions hold:

1. The data will survive a power failure since it is in NVRAM. When the server recovers, it will detect that it was stopped (as opposed to crashed) and will write the data to disk.

2. The data will survive a client crash.

3. The data will survive a server crash since it is also kept in the client write buffer.

When the server receives the data, the client can release any user process which can then continue. The cost of copying the data to the server is much less than actually writing it to disk, and the user is released earlier than they otherwise would have been. In order for this to work, the following three operations must be performed:

1. The client sends the contents of its write buffer to the server.

2. The server notifies the client when the data has been received.

3. The server notifies the client when the data has been written (so that the client can remove the data from the buffer.)

The two first can be embedded in a blocking RPC initiated by the application, while the last will be an asynchronous event in the client's cache manager.

Note that the NVRAM is only used for recoverly from power failures, not from server crashes.

2.3.3 Storage System

The storage system provides basic storage of contiguous byte-oriented files based on unique file numbers. The storage system is responsible for writing the files to disk and maintaining meta information to retrieve the files later. It uses a set of parallel disks organized in a RAID on which it stores the files and meta information.

The disk is partitioned in a set of large blocks (typically 1MB). The time to transfer a block to and from disk is partitioned in two parts:

1. The disk overhead time (rotational delay and head settle time).

2. The data transfer time.

When using large blocks, the time in 1) is amortized over more data, and our experiments show that when the block size is set to 1 MB (or more) data can be read (or written) to disk at almost full disk-speed.

The file server uses the large blocks in a *log-structured* manner. Each block is considered a file system *segment* and many file system updates are recorded in a segment. A set of updates to the file system is first collected in a memory segment and is than written to disk as a whole. The segments are linked together through a pointer list to form one logical file system log.

Multimedia files, however, are treated differently. A block of multimedia data is first recorded in memory and flushed to disk as a whole which is called out-of-band storage. Care is taken that the data, inclusive administrative information (see below) exactly fills one segment. To play a multimedia file, the block is read into memory as a whole before it is played back. The meta information of the multimedia files is recorded in the "normal" file system log. The storage system does not deal with multimedia timing constraints. This is handled in the multimedia type manager, which is described in Section 2.3.7. Since a block is defined over all the available disks, transferring one block uses all the disks. Hence, all disks are equally loaded when reading or writing multimedia files.

File System Log

Each file is the file system is described by a *pnode*. The pnodes can be compared to the UNIX *inodes*. The main difference is that the pnodes do not have a fixed location on disk, and do not have fixed size. Pnodes consist of:

- Time of creation (of the file) and time of last modification.
- File size.
- A list of log ranges which hold the file data.
- A unique pnode number.
- Data needed by the type of the file.
- Information related to access control.

The size of the pnode is dependent upon the number of file ranges in the log, information stored by the type manager, and the length of the access control information.

Each time a file system update is done, the pnode is extended with a new entry describing the log range of the new entry. The storage system tries to store the files consecutively on disk thereby making the pnode holding only one entry. The storage system always stores the pnode in front of the file.

We will show two different types of updates to a file. First, Figure 5 shows a typical sequence of creating a file and then append data to it. In this example, the file described by the pnode `P1` is written to the log with some initial data (`entry 1`). Next, more data (`entry 2`) is added to the same file. A new version of the pnode (`P1'`) and the new file data is appended to the log. We can see that the pnode (`P1'`) now contains a copy of the old pnode[4] plus a pointer to the new data.

In Figure 6, an update is made to the same file by that the data initially written is replaced. The pnode (`P1''`) still contains only two pointers. The data initial data (`entry 1`) and the two old pnodes are now inaccessible and will be *cleaned*. Cleaning is explained below.

When reading a pnode; If the data following the pnode is the initial data of the file, the storage system will read it as well. Since most applications reads files sequentially as is shown by the Sprite traces [Baker91] this storage strategy improves the file server's performance. By storing the files consecutively on disk, read performance is improved.

Check-pointing the File System

In an ordinary type file system (e.g. Unix FFS), the location of the metadata is fixed: there exists a special area on disk where all the metadata is stored. But if the file system is log-structured, as is the case in HFS, the location of the metadata is not constant: they are stored in the log. In order to find for example a pnode, the storage system maintains a pnode map. The pnode map lists for every pnode its location in the log and its size. The pnode map is maintained in memory. Periodically, the memory resident version of the pnode map is written to disk and the location of the pnode map is recorded at a known location (e.g. the super block).

[4] The time of last modification has changed in the "general information" part of the pnode.

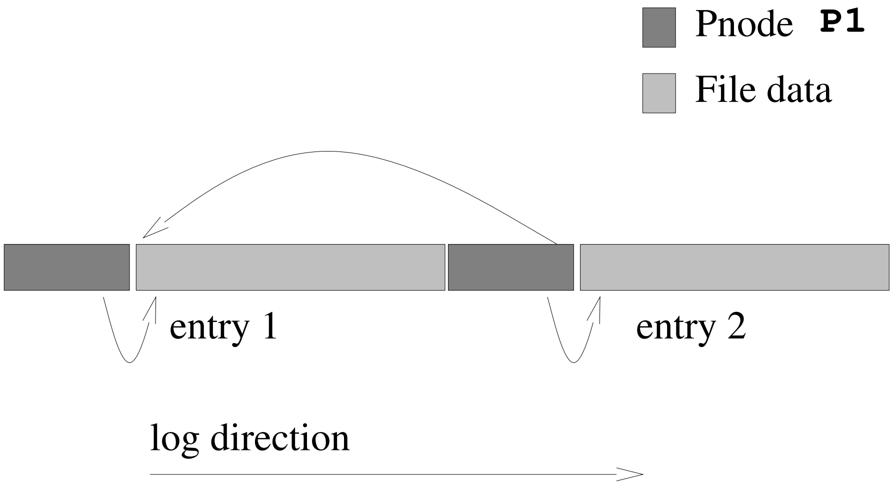

Figure 5. Writing a file

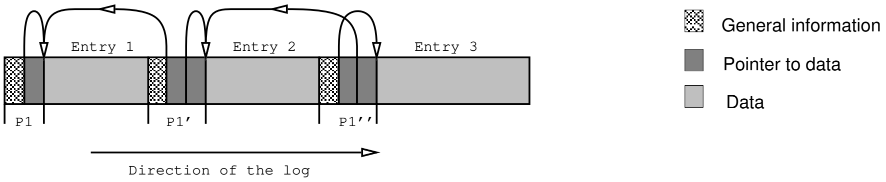

Figure 6. Update on a file

When the number of files stored on the file system grows, the amount of data that needs to be written to disk to store the map also grows. HFS aims at storing at least 10^7 files, and writing the pnode map consecutively in the log is prohibitive.

To avoid this, the pmap is stored in a file, the *pnode file*. When the file system is *check-pointed* the difference between the on-disk version and the memory resident version is saved. The rest of the pmap file stays intact. The location of the pnode describing the pmap file, is recorded in the super block. The pmap file's pnode is recorded in the log as a normal pnode, and the cost of a checkpoint is therefore the time of writing the pmap to the log, plus the time to seek to the super block and record the location of the pnode.

When booting, the storage system locates the pnode describing the pmap file, read in all its different parts and reconstructs the memory pnode map by traversing the rest of the file system log, recording the location of the pnodes which are written after the last update to the pmap file.

Roll Forward

In Unix FFS [McKusick84] a large amount of time is spent in order to keep the file system consistent. This is expensive since the inodes, in which the meta data for files are recorded, are not stored at the same location as the file itself—requiring a costly seek. In a log-structured file system, the problem of consistency changes to reconstructing the end of the log (the part written since the last checkpoint). The reconstruction process is called *roll-forward*.

When the file server start after a crash, the pnode map is reconstructed. The location of the pmap file's pnode is taken as the end of the log. Each successfully written segment contains a pointer to the next segment, and the storage system rolls forward through the segments. Each segment is described by the *segment header*. Among other things, the headers describe for each pnode stored in the segment its offset in the segment, its number and size. When it arrives at the segment which is stored in NVRAM (the write buffer), the end of the log has been found, and the file system is again consistent.

The time to recover must be weighed against the cost of check-pointing.

Cleaning

Ideally, the amount of disk space assigned to the log would be infinite. In a practical system, however, the amount of available disk space is limited. At some point, the available disk space will be exhausted and the log must be compressed to remove the data which cannot be accessed anymore. The process that compresses the log is called *cleaning*. The base objects of cleaning are the segments. The cleaner selects segments from the log which have inaccessible data. The cleaner copies the accessible data from the segment to the end of the log, creates new pnodes, and makes the segment available for reuse in the log.

BSD LFS and Sprite LFS show that cleaning the log is an expensive operation and takes away lot of the performance benefits of log-structured file systems when cleaning the file system log. The Sprite traces [Baker91] show that most updates to the file system are immediately thrown away. This results in a poor fill rate of the segments at the end of the log and thus more cleaning.

In order to perform cleaning, the storage system maintains a *garbage file*. This file holds an unordered list of log ranges which can not be reached through the pnodes. Whenever there is a file system update, the storage system puts an entry in the garbage file describing which part of the file has become inaccessible. Every once in a while the cleaner is activated. The cleaner reads the garbage file into memory and sorts the file. It then determines the inverse of the garbage file: the *reachable* set. By comparing the reachable set with the header for each segment, the cleaner is able to find the pnode numbers to which the reachable range is connected. Next, the cleaner reads in the latest version of the pnode and creates a new file version by copying the file data to the end of the log. This operation will add another entry to the garbage file. If the cleaner determines that no life data is in the segment, the segment is considered *clean*.

Note that until cleaning has been performed, all (logically) deleted data is still present, and not overwritten, in the log. Any previous version of a file, or a deleted file, can be found by traversing back through the log. This means that an `undelete` utility can be written.

Out-of-Band Storage

Out-of-band storage is provided for multimedia data. Out-of-band storage is used by the multimedia server (see Section 2.3.7) to read and write "raw" disk segments. This strategy allows the multimedia server to make use of the "raw" disk performance. It is our belief that there is not a need for a more elaborate interface. The multimedia server needs to provide enough slack to cope with slight variations in disk throughput. We expect "raw" disk throughput to be reasonably constant even when there is a lot of "normal" file server traffic.

An empty segment is allocated which can be used to store the file data for one file. Since the segment is defined "over" the RAID (see below), out-of-band storage of file data can proceed at full (parallel) disk speed. To register out-of-band storage normal pnodes are used. These pnodes are simply stored in the normal file system log. Hence, the multimedia files can also be read through the normal file system's interface without timing guarantees.

RAID

A file server segment is defined "over" a set of parallel disks. Transferring a segment to the disks therefore proceeds at full parallel disk speed. However, when the storage system stripes the segment over the parallel disk, parts of the segment may fail: if a disk fails, parts of the segment become inaccessible. If a part of the segment cannot be read, the complete segment cannot be read. The change that a segment becomes inaccessible is related to the number of parallel disks. We considered this unacceptable. HFS therefore uses a RAID [Patterson88] organization. Data redundancy is introduced to cope with failing disks. Redundant data is stored on an extra disk. Initially, the file server uses a RAID 5 organization. An 1MB segment is defined over 4 disks, the 5th disk contains the *exclusive or* of the segment. If a disk fails, the data is reconstructed by XOR-ing the redundant data disk with the non-failed disks.

If RAID technology is used in an in-place type file system, recalculation of the redundant data when the file server is updated becomes a bottleneck [Cao92]. The contents of the redundant data disks need to be read back to recalculate the XOR for every disk write operation. Log-structured file systems can make effective use of

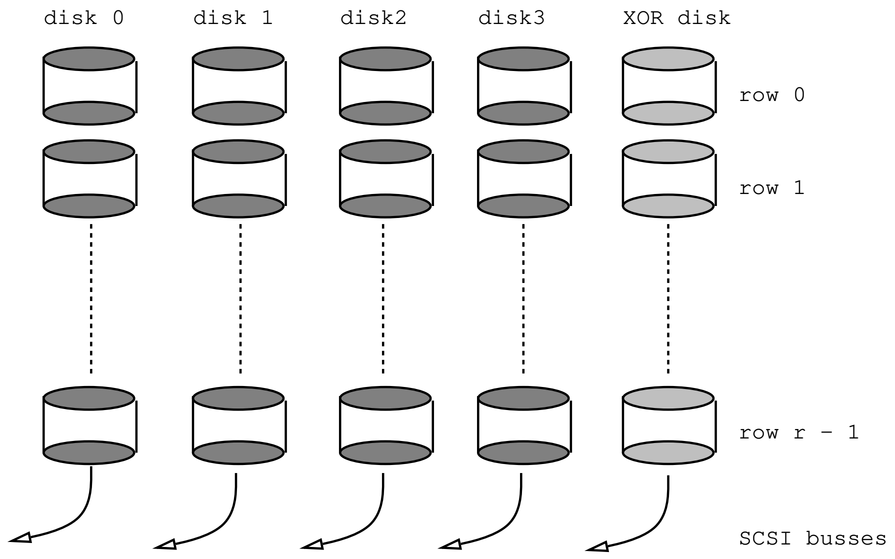

Figure 7. sequential RAID 5 configuration

the non-overwriting strategy: the old contents of the disks is always zero. Writing a segment is a simple operation of calculation of the XOR and writing the segment and the XOR to the 5 disks. Also, since the XOR data does not have to be read back when writing a new segment, *hot-spots* in disk accesses do not occur. There is no need to distribute the XOR data over all the available disks. This simplifies the storage algorithm. The disadvantage of this strategy is that segments needs to be written as a whole: unwritten parts of the segments need to be zero. This, however, does not degrade file system performance: partial segments are only written if the file server is lightly loaded.

The file server will use a sequential RAID 5 configuration. The stripe is defined over $4 + 1$ parallel disks to increase disk throughput. Several RAID 5 configurations will be built and used in sequence to increase the file system's storage capacity. Each disk in the RAID 5 will be equipped with a separate SCSI bus to the host computer, each n-th disk in the sequence of RAIDs will use the same SCSI bus. Figure 7 shows the configuration.

Archiving

The storage system provides automatic archival storage for the file system. Once per day, all the changes made to the file system are sent to the archive server. Once files are archived, they can be retrieved from the archive using three methods:

- Retrieve the last version of a file.
- Retrieve a specific version of a file.
- Retrieve a specific version of a file at a specific date.

In order to retrieve specific file versions, files are equipped with a version number. The file number is also extended with the time of file creation to make a unique file number over time.

Every set of files (a backup) is called an *archive*. Every day, the file server sends a new archive to the archive server. The file server calculates which files are in the archive by reading in a description of the last archive from the archive server and comparing it to the current file system. The file server requests the archive server to make updates based on the file system's contents. The file server can ask the archive server to delete files or to archive a new version of a file. Once all changes have been sent to the archive server, the file server *commits* all changes in the archive server. The archive server starts to calculate a new archive description and makes the changes permanent. Once all changes are safely stored, the archive server acknowledges the changes to the file server. During the archive action, the file server is not allowed to clean its log: a file might be thrown away of which a backup still has to be made.

The archive server maintains its data log structured. Every update to the archive system is appended to the log. Files and file meta data (pnodes) are written intermixed. In order to find pnodes in the log, an archive pnode map is maintained. An archive pnode map describes, for every version of the file stored in the archive, the log range of the file's pnode. This structure allows the first two query types which are described above. The third query type is solved by maintaining a pmap map (or $pmap^2$). The $pmap^2$ specifies the log ranges of the various pnode maps in the log. The address and size of the $pmap^2$ table is replicated on an archive disk. This is used to find the last $pmap^2$ when the archive server boots. If the address is lost, the complete archive log needs to be searched to find the end of the log. The archive disk is also used to cache the last pmap and $pmap^2$. This allows faster retrieval of archived data.

In order to keep the amount of archived data within bounds, we've invented a mechanism to *age* data. The archive server has defined 4 levels of archiving: daily, weekly, monthly and yearly. The daily archive is made as described above. Once a week, the archive server compares the last daily archive to the weekly archive made a week earlier. All changes to this weekly archive are stored as the new weekly archive. After some weeks the oldest daily archives can be discarded. This strategy is applied to the monthly and yearly backups as well. We call this *iterative logging*.

2.3.4 Server Cache System

To obtain good performance, the file server will cache data written by clients, and data read from disk. When the server cache obtains a token for a file, it will multiplex requests from the client caches. By adhering to the protocol shown above, caches are kept consistent even in the presence of failure. Since tokens are for the whole file, caches are encouraged to cache whole files.

The server cache has, in the prototype, been implemented based on caching of blocks. The block size has been chosen so large that most files will fit in one block.

Most files are deleted very shortly after they were created, and the data written to them can be discarded [Baker91, Floyd86]. We believe that very few of these files will ever be written to disk. Furthermore, most of these files are temporary, and will be regenerated (as opposed to recovered) if the machine (or application) crashes. Files of this nature will be cached in memory and the machinery needed to ensure permanent storage will not be invoked [McKusick90].

2.3.5 Client Cache System

The client cache will provide a file system cache on the client's machine. It will requests files from the file server on demand of clients. Furthermore, updates to the file system first arrive in the client cache before they are sent to the server machine. Before a file can be cached, the client cache needs to obtain a token from the file server.

The client caches have not been designed yet, since there is no clear view on what is needed by the Pegasus operating system.

Access from traditional workstations

Initially, users of the file server will be workstations running UNIX. Two approaches are planned to support integration into UNIX name space. This first is to integrate the cache manager with a `vnode` interface and include it in the kernel. This will give access to the full set of services, such as single site semantics.

For clients were this not possible, a user level NFS server will be supported. It will then translate NFS requests generated by the local client into request sent to the file server. Such a server will probably result in poor performance, in relation to a kernel implementation, but will be a simple and portable way to provide access to the server.

2.3.6 Name Service

In the file server, files are located by their *name*. The name space is meant to be efficient to manage; both in time and space. It is therefore flat. Users, on the other hand, want naming to be based on strings, and the name space to be hierarchical. This name service will be provided by a type server.

In the UNIX file system all *inodes* are kept in a fixed location. In UNIX this is a portion of the disk. We use a file. The advantage is that we can increase the number of inodes dynamically. When the name server boots, it opens a reserved file[5] and reads in the *super block*. The super-block contains administrative data, and the location (in which files) the inodes can be found. Inode number 2 contains the name of the file containing the contents of the directory "/". Since some directories, such as "/", is heavily read-shared, the name server will multiplex read tokens on this file on the single token it got from the token manager.

The name server has been made available to make it simple (for humans) to name and find files. We believe that other type server will implement their own name space. For example, the multimedia server will store several physical files containing indexing information in one file, the audio in another, and the video in a third. For the user, only one (logical) multimedia file exist. A meta file will be made available to the user in the "traditional" name space.

2.3.7 Multimedia Server

The *multimedia server* is one of the type servers. The multimedia server is responsible for managing incoming and outgoing multimedia streams. In particular, the multimedia server is able to allocate a part of the available throughput from the storage system, buffer incoming multimedia streams in memory and deliver the outgoing

[5] Since we are providing UNIX like naming, we choose file number 2 as this reserved file.

data streams in a timely manner. Furthermore, the Multimedia Server is able to provide *Quality-of-Service* guarantees to its clients. Clients can negotiate with the server to establish a QoS which is acceptable for the client and server. The QoS manager in the multimedia server manages the available disk throughput.

The multimedia server views the core as a black box which can deliver a fixed throughput through the *Out-of-Band* storage interface as described in Section 2.3.3. We expect the available throughput to be reasonably stable.

The Sprite traces [Baker91] show that each user generate small amounts of data per second (only 8 KB on average) but that this traffic is bursty. We believe that the large file-server cache will absorb the bursts. The rest of the throughput is used for multimedia data transfer, distributed over all multimedia clients. The multimedia server will allocate a sufficient amount of memory to buffer variations in data stream and file system throughput. If clients temporarily transmits more than the available file system throughput, the multi-media server will buffer the data in memory.

3 Assessments Criteria

The prototype of the Huygens File Service needs to be able to perform at least as good as current research file systems such as Sprite LFS, BSD-LFS and EFS. The performance numbers will be obtained by running the benchmarks as described in [Baker91].

Furthermore, HFS needs to be able handle at least 25 GB of disk storage and 15 Mb/s sustained throughput. The network which will be used by the Pegasus project runs at 100 Mbps; HFS must be able to saturate the network. At least 50% of this throughput must be useable for concurrent multimedia file I/O. This will be shown by playing back and recording a set of multimedia streams.

The prototype archive server needs to be able to scale at least up to 1 TB of data. Since 1 TB of archival storage is too expensive to buy, we will prove our concept with a prototype of 100 GB of data. The data structures and storage algorithms will be designed such that the available storage capacity does not influence the archive server's performance.

4 Implementation Plan

A preprototype of HFS has been constructed in order to give us access to a research vehicle. There are several areas that need to be investigated, and the shortcomings of the prototype must be seen in this light.

The preprototype is nearly finished which allows us to read and write ordinary files and to store data out-of-band. In the two years, we will:

- Configure the disks as a RAID 5;
- Measure the performance of the file server and make adjustments to the preprototype;
- Stabilize the preprototype and use the file server as our "standard" file server;
- Design and implement NVRAM support for the file service;

- Design and implement the Multimedia Service;
- Design and implement client cache machinery and integrate it into the Pegasus operating system as well as in contemporary machinery;
- Design and implement the archival storage system;

References

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 198–212. Association for Computing Machinery SIGOPS, 13 October 1991.

[baker92] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, October 12-15, 1992). Published as *OSR*, **26**(Special issue):10–22. ACM Press, oct 1992.

[Cabrera91] Luis Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. Technical report CRL-91-46. UC Santa Cruz, 1991. To appear, *Computing Systems*.

[Cao92] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The tickertaip parallel raid architecture. HPL-OSR-92-6, 6 november 1992.

[Floyd86] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical report 177. Computer Science Department, University of Rochester, NY, March 1986.

[Gibson93] Garth A. Gibson and David Patterson. Designing disk arrays for high data reliability. *Journal of paralell and distributed computing*, **17**:4–27. Academic Press, Incorporated, 1993.

[Howard88] John H. Howard, Michael Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, **6**(1):51–81, feb 1988.

[Jardetzky92] Paul Wenceslas Jardetzky. *Network file server design for continuous media*. PhD thesis. Computing Laboratory at the University of Cambridge, Computer Laboratory, Pembroke Street, Cambridge CB2 3QG, England, April 1992. Techical report no. 268.

[Kistler91] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda file system. *13th SOSP* (Pasific Grove, Ca, USA 13 October 1991). Published as *SIGOPS*, **25**(5):213–25, October 1991.

[Lamport85] Leslie Lamport. On interprocess communication. Technical report 8. Digital Equipment Corporation, Systems Research Center, Palo Alto, Ca, December 1985.

[Leslie93] Ian Leslie, Derek McAuley, and Sape J. Mullender. Pegasus - operating system supports for distributed multimedia systems. *SIGOPS*, **27**(1):69–78, jan 1993.

[Levy90] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. *ACM Computing Surveys*, **22**(4):321–74, December 1990.

[Liskov91] Barbara Liskov, Robert Gruber, Paul Johnson, and Liuba Shrira. A replicated Unix file system. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review*, **25**(1):60–4, January 1991.

[Macklem91] Rick Macklem. Lessons learned tuning the 4.3BSD Reno implementation of the NFS protocol. *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 53–64, 21–25 January 1991.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[McKusick90] Marshall Kirk McKusick, Michael J. Karels, and Keith Bostic. A pageable memory based filesystem. *UKUUG Summer 1990* (London), pages 109–15. United Kingdom UNIX systems User Group, Buntingford, Herts, 9–13 July 1990.

[Mullender92] Sape J. Mullender, Ian M. Leslie, and Derek McAuley. Pegasus Project Description. Memoranda Informatica 92–75. University of Twente, Faculty of Computer Science, September 1992.

[Nelson86] Michael N. Nelson. Virtual memory for the Sprite operating system. Technical report UCB/CSD 86/301. University of California at Berkeley, June 1986.

[Nelson88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, **6**(1):134–54, February 1988.

[Ousterhout89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM Operating Systems Review*, **23**(1):11–28, January 1989. Also appears as University of California, Berkeley, Technical Report UCB/CSD 88/467.

[Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of SIGMOD.* (Chicago, Illinois), 1–3 June 1988.

[Patterson93] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *OSR*, **27**(2):21–34. ACM Press, apr 1993.

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *UKUUG Summer 1990* (London), pages 1–9. United Kingdom UNIX systems User Group, Buntingford, Herts, 9–13 July 1990.

[Rangan91] P. Venkat Rangan and Harrick M. Vin. Designing file systems for digital audio and video. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 81–94. Association for Computing Machinery SIGOPS, 13 October 1991.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Pacific Grove, CA, USA, October 1991). Published as *SIGOPS*, **25**(5):1–15. Association for Computing Machinery, October 1991.

[Satyanarayanan89] M. Satyanarayanan. A survey of distributed file systems. In Journal F. Traub et.al, editor, *Annual Review of Computer Science, 1990*, pages 73–104. Annual Reviews Incorporated, 1990.

[Schneider87] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical report 87–859. Department of Computer Science, Cornell University, Ithaca, NY, August 1987.

[Seltzer92] Margo Ilene Seltzer. *File system performance and transaction support*. PhD thesis. University of California, 1992.

[Srinivasan89] V. Srinivasan and Jeffrey C. Mogul. Spritely NFS: experiments with cache-consistency protocols. *Proceedings of the 12th ACM Symposium on Operating System Principles* (Litchfield Park, AZ, 3–6 December 1989). Published as *Operating Systems Review*, **23**(5):45–57, December 1989.