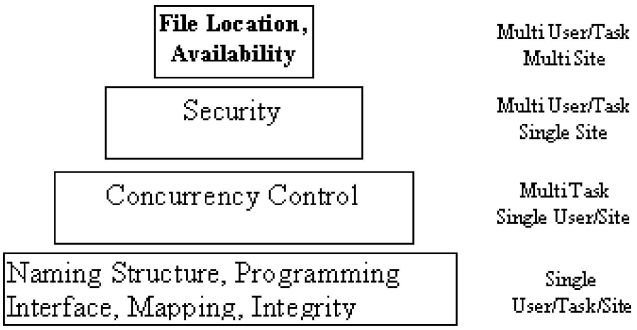
Distributed File Systems

A File System is a refinement of the more general abstraction of permanent storage. Databases and Object Repositories are other examples. A file system defines the naming structure, characteristics of the files and the set of operations associated with them.

The classification of computer systems and the corresponding file system requirements are given below. Each level subsumes the functionality of the layers below in addition to the new functionality required by that layer.



Distributed File Systems constitute the highest level of this classification. Multiple users using multiple machines connected by a network use a common file system that must be efficient, secure and robust.

The issues of file location and file availability become significant. Ideally, file location should be transparent so that files are accessible by a path name in the naming structure that is independent of the physical location of the file. It is of course simpler to statically bind names or sub trees of the namespace to machine locations that must be provided by the user to access the relevant files. This is not satisfactory for users as they are required to remember machine names and when these names are hardwired into applications it is inconvenient to relocate parts of the file system data when carrying out ordinary administration activities. Availability is achieved through replication, which introduces complications for maintaining consistency. Availability is important, as a user may not have control over the physical location where a file is stored and, when using that file at another site, should still have access irrespective of the status of the host where the file may be located.

File Systems and Databases
File Systems and Databases have many similar properties and requirements. However there are many conceptual differences as well.

Encapsulation: File systems view data in files as a collection of uninterpreted bytes whereas databases contain information about the type of each data item and relationships with other data items. As the data is typed the database can constrain the values of certain items. A database therefore is a higher level of abstraction of permanent storage as it subsumes a portion of the functionality that would be required by applications built on file systems. In addition to enforcing constraints it provides query and indexing mechanisms on data.

Naming: File Systems organise files into directory hierarchies. The purpose of these hierarchies is to help humans deal with a large number of named objects. However, for very large file namespaces this method of naming can still be difficult for users to deal with. Databases allow associative access to data, that is, data is identified by content that matches search criteria.

The *ratio of search time to usage time* is the factor that determines whether access by name is adequate. If an item is used very frequently, the ratio is low and a file system is adequate. Not surprisingly, the usage patterns of a file system exhibit considerable temporal locality of this kind. Databases usage patterns exhibit very little temporal locality.

File Systems and Databases (Continued)
Dealing with usage patterns: Distributed file systems and databases therefore use two different strategies for enhancing performance. Distributed file systems use *data shipping* where data is brought to the point of use. It is cached and commonly reused frequently. Distributed databases use function shipping where computation is shipped to the site of the data storage. Transporting the search function requires only a small amount of network bandwidth. The alternative would be to transport the entire amount of data to be searched to where the search was to be done.

Granularity of Concurrency: Databases are typically used by applications that involve concurrent read and write sharing of data at fine granularity by large numbers of users. In addition there are requirements for strict consistency of this data and atomicity of groups of operations (transactions). With file systems, sharing is more coarse grained, at the file level, and individual files are rarely shared for write access by multiple users although read sharing is quite common, for example, system executable files. It is this combination of application characteristics that makes it a lot more difficult to implement distributed databases than distributed file systems.

Empirical Observations
Many of the current techniques for designing distributed file systems have arisen from empirical studies of the usage of existing centralised file systems.

> The *size distribution* of files is useful for determining the most efficient means of mapping files to disk blocks. It is found that most files are relatively small.
>
> The relative and absolute *frequency of* different types of *file operations* has influenced the design of cache fetch algorithms and concurrency mechanisms. Most file access is sequential rather than random and read operations are more common than write. It is also noted that users rarely share files for writing.
>
> Information on *file mutability* is also useful. It is found that most files that are written are overwritten often. Consider for example temporary files generated by a compiler or files which we are editing and save repeatedly. This information can guide a cache write policy.
>
> The *type of a file* may substantially influence its properties. Type specific file information is useful in file placement and in the design of replication mechanisms. Executable files are easily replicated as access is read only.
>
> What is the *size of the set of files referenced* in a short period? This will influence the size of the cache and catch fetching policy. Most applications exhibit temporal locality within the file namespace and this set if found to be small in general.

Problems with Empirical Data
Gathering empirical data requires *modification to an operating system* to monitor these parameters. This process itself may impact on the system performance in a small way.

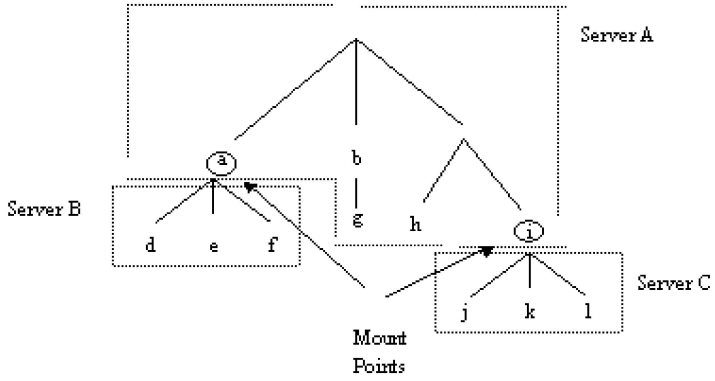
When interpreting the data, consideration must be given to the environment in which it was gathered. A study of a file system used in an academic environment may not be sufficiently *general* for other kinds of environments.

Another concern relates to the *interdependency* of the design studied and the data observed. The nature of a particular file system implementation may unduly influence the way in which users use the system. For example, if the caching system used whole-file transfer there would be a disincentive to create very large files.

Another issue is that most empirical studies are carried out on existing centralised file systems under the assumption that *user behaviour and programming characteristics* will not change significantly in a distributed system.

Mechanisms for Building Distributed File Systems

Mounting
Mount mechanisms allow the binding together of different file namespaces to form a single hierarchical namespace. The Unix operating system uses this mechanism. A special entry, known as a mount point, is created at some position in the local file namespace that is bound to the root of another file name space. From the users point of view a mount point is indistinguishable from a local directory entry and may be traversed using standard path names once mounted. File access is therefore location transparent for the user but not for the system administrator. The kernel maintains a structure called a mount table which maps mount points to appropriate file systems. Whenever a file access path crosses a mount point, this is intercepted by the kernel, which then obtains the required service from the remote server.



Client machines can maintain mount information individually as is done in Sun's Network File System. Every client individually mounts all the required file systems at specific points in its local file space. Note that each client need not necessarily mount the file systems at the same points. This makes it difficult to write portable code which can locate files in the distributed file system irrespective of where it is executed. Movement of files between servers requires each client to unmount and remount the affected subtree. Note that to simplify the implementation, Unix does not allow hard links to be placed across mount points, i.e. between two separate file systems. Also, servers are unaware of where the subtrees exported by them have been mounted.

Mount information can be maintained at servers in which case it is possible that every client sees an identical namespace. If files are moved to a different server then mount information need only be updated at the servers. Servers each manage domains or volumes which are subtrees of the overall name space. Clients may initially direct file requests to any server which will direct the client to the server which manages the domain containing the required file. This information may be used to guide the client more efficiently for future accesses.

Client Caching
Caching is the architectural feature which contributes the most to performance in a distributed file system. Caching exploits temporal locality of reference. A copy of data stored at a remote server is brought to the client. Other metadata such as directories, protection and file status or location information also exhibit locality of reference and are good candidates for caching.

Data can be cached in main memory or on the local disk. A key issue is the size of cached units, whether entire files or individual file blocks are cached. Caching entire files is simpler and most files are in fact read sequentially in their entirety, but files which are larger than the client cache

cannot be fetched. Cache validation can be done in two ways. The client can contact the server before accessing the cache or the server can notify clients when data is rendered stale. This can reduce client-server traffic.

A number of approaches to propagating changes back to the server are also possible. A change may be propagated when the file is closed or by deferring it for a set period of time. This would be useful where files are overwritten frequently. A policy of deferred update is also useful in situations where a host can become disconnected from the network and propagates updates upon reconnection.

Hints

Caching is an important mechanism to improve the performance of a file system, however, guaranteeing the consistency of the cached items requires elaborate and expensive client/server protocols. An alternative approach to maintaining consistency is to treat cached data (mostly meta data) as hints. With this scheme, cached data is not expected to be completely consistent, but when it is, it can dramatically improve performance. For maximum performance benefit, a hint should nearly always be corrrect. To prevent the occurrence of negative consequences if a hint is erroneous, the classes of applications that use hints must be restricted to those that can recover after discovering that the cached data is invalid, that is, the data should be self validating upon use. Note that file data cannot be treated as a hint because use of the cached copy will not reveal whether it is current or stale.

For example, after the name of a file or directory is mapped to a physical object in the server, the address of that object can be stored as a hint in the client's cache. If the address fails to map to the object on a subsequent access, it is purged from the cache and the client must consult with the file server to determine the actual location of the object. It is unlikely that file locations will change frequently over time and so use of hints can benefit access performance.

Bulk Data Transfer

All data transfer in a network requires the execution of various layers of communication protocols. Data is assembled and disassembled into packets, it is copied between the buffers of various layers in the communication protocols and transmitted in individually acknowledged packets over the network. For small amounts of data, the transit time across the network is low, but there are relatively high latency costs involved with the communication protocols establishing peer connections, packaging the small data packets for individual transmission and acknowledging receipt of each packet at each layer.

Transferring data in bulk reduces the relative cost of this overhead at the source and destination. With this scheme, multiple consecutive data packets are transferred from servers to clients (or vice versa) in a burst. At the source, multiple packets are formatted and transmitted with one context switch from client to kernel. At the destination, a single acknowledgement is used for the entire sequence of packets received.

Caching amortizes the cost of accessing remote servers over several local references to the same information, *bulk transfer* amortizes the cost of the fixed communication protocol overheads and possibly disk seek time over many consecutive blocks of a file. Bulk transfer protocols depend on the spatial locality of reference within files for effectiveness. Remember there is substantial empirical evidence that files are read in their entirety. File server performance may be enhanced by transmitting a number of consecutive file blocks in response to a client block request.

Encryption

Encryption is used for enforcing security in distributed systems. A number of possible threats exist such as unauthorised release of information, unauthorised modification of information, or unauthorised denial of resources. Encryption is primarily of value in preventing unauthorised release and modification of information.

The Kerberos protocol is most commonly used as a mechanism which employs encryption for initially establishing trusted communication between two parties and for generating a private session key for encrypting and decrypting subsequent messages between them. Private session keys tend to

be shorter than those used for public key encryption and this makes it cheaper (easier) to perform the encryption.

For performance, encryption/decryption may be performed by special hardware at the client and server. It may be difficult to justify the cost or need for this hardware to users. The benefit is not as tangible as extra memory, processor speed or graphics capability and may be viewed as an expensive frill until the importance of security is perceived.

Stateful and Stateless File Servers
A client operates on files during a session with a file server. A session is an established connection for a sequence of requests and responses between the client and server. During this session a number of items of data may be maintained by the parties such as the set of open files and their clients, file descriptors and handles, current position pointers, mounting information, lock status, session keys, caches and buffers. This information is required partly to avoid repeated authentication protocols for each request and repeated directory lookup operations for satisfying each file access during the session. The information may reside partly in clients or servers or both.

How state information is divided affects the performance and management of the distributed file system. For example, file position pointers, mounting information (as hints) and caches are more naturally maintained by the client. The session key is shared between both client and server. Locks represent global information shared by all clients, so it seems most suitable for the server to manage this information.

A file server is said to be *stateful* if it maintains internally some of the state information relating to current sessions and *stateless* if it maintains none at all. The more state information known about a connection means the more flexibility there is in exercising control over it. However, stateful servers are to be avoided generally in distributed systems. Stateless servers make it much easier to implement fault tolerant services then one that has to keep track of and recover state information. Client crashes have no effect on stateless servers. Crashes of stateless servers are easy to recover from and can be perceived by clients as longer response delays (or unresponsiveness) without disrupting their sessions.

Each file request to a stateless server must contain full information about the file handle, position pointer, session key and other necessary information. In addition, the following issues must be addressed:-

> *Idempotency:* How does the server deal with possible repeated client requests caused by the previous failures (nonresponsiveness) of the server? One solution is to make all requests idempotent meaning that a request can be executed any number of times with the same effect. For example, NFS provides only idempotent services like read-a-block or write-value-to-block but not append-a-block. Not all services can be made idempotent, for example lock management. In this case the client attaches sequence numbers to each request so that the server can detect duplicate or out of sequence requests. These sequence numbers must be maintained by the server. The need for sequence numbers at the client and server level may be eliminated if RPC communication is used on a reliable connection oriented TCP transport service. RPC call mechanism may offer *at-least-once* or *at-most-once* or *maybe* semantics.
>
> *File Locking:* How can sharing clients implement a file locking mechanism on top of stateless file servers?
>
> *Session Key Management:* How is the session key generated and maintained between the client and stateless file server? Can a one time session key be used for each file access?
>
> *Cache Consistency:* Is the file server responsible for controlling cache consistency among clients? What sharing semantics are supported? Ideally, we like updates of a file to be completed instantaneously and the consistent results visible to other clients immediately but practically this is difficult to achieve and is often relaxed. Unix semantics are to propagate a write to a file and its copies immediately so that reads will return the latest value. Subsequent

accesses from the client that issued the write must wait for the write to complete. The primary objective is to maintain currency of the data. Transaction semantics are to store writes tentatively which are committed only when consistency constraints are met at the end of the transaction. The primary objective is to maintain consistency of the data. Session semantics are to perform writes to a cached working copy of a file which is made permanent at the end of the session. The primary objective is to maintain efficiency of data access.

For practical purposes, it seems some minimal state information may have to be maintained by the server.