Lectures on distributed systems

Distributed File Systems Design

Paul Krzyzanowski

Introduction

Presently, our most common exposure to distributed systems that exemplify some degree of transparency is through distributed file systems. We'd like remote files to look and feel just like local ones.

A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files. File systems provide directory services, which convert a file name (possibly a hierarchical one) into an internal identifier (e.g. inode, FAT index). They contain a representation of the file data itself and methods for accessing it (read/write). The file system is responsible for controlling access to the data and for performing low-level operations such as buffering frequently-used data and issuing disk I/O requests.

Our goals in designing a distributed file system are to present certain degrees of transparency to the user and the system:

access transparency
: Clients are unaware that files are distributed and can access them in the same way as local files are accessed.

location transparency
: A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.

concurrency transparency
: All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.

failure transparency
: The client and client programs should operate correctly after a server failure.

heterogeneity
: File service should be provided across different hardware and operating system platforms.

scalability
: The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).

replication transparency
: To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.

migration transparency
: Files should be able to move around without the client's knowledge.

support fine-grained distribution of data

To optimize performance, we may wish to locate individual objects near the processes that use them.

tolerance for network partitioning
The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

Distributed file system concepts

A **file service** is a specification of what the file system offers to clients. A **file server** is the implementation of a file service and runs on one or more machines.

A **file** itself contains a name, data, and attributes (such as owner, size, creation time, access rights). An **immutable file** is one that, once created, cannot be changed. Immutable files are easy to cache and to replicate across servers since their contents are guaranteed to remain unchanged.

Two forms of protection are generally used in distributed file systems, and they are essentially the same techniques that are used in single-processor non-networked systems:

capabilities
Each user is granted a ticket (capability) from some trusted source for each object to which it has access. The capability specifies what kinds of access are allowed.

access control lists
Each file has a list of users associated with it and access permissions per user. Multiple users may be organized into an entity known as a group.

File service types

To provide a remote system with file service, we will have to select one of two models of operation. One of these is the **upload/download model**. In this model, there are two fundamental operations: *read file* transfers an entire file from the server to the requesting client, and *write file* copies the file back to the server. It is a simple model and efficient in that it provides local access to the file when it is being used. Three problems are evident. It can be wasteful if the client needs access to only a small amount of the file data. It can be problematic if the client doesn't have enough space to cache the entire file. Finally, what happens if others need to modify the same file? The second model is a **remote access model**. The file service provides remote operations such as *open*, *close*, *read bytes*, *write bytes*, *get attributes*, etc. The file system itself runs on servers. The drawback in this approach is the servers are accessed for the duration of file access rather than once to download the file and again to upload it.

Another important distinction in providing file service is that of understanding the difference between *directory service* and *file service*. A **directory service**, in the context of file systems, maps human-friendly textual names for files to their internal locations, which can be used by the file service. The **file service** itself provides the file interface (this is mentioned above). Another component of file distributed file systems is the **client module**. This is the client-side interface for file and directory service. It provides a local file system interface to client software (for example, the vnode file system layer of a UNIX kernel).

Naming issues

In designing a distributed file service, we should consider whether all machines (and processes) should have the exact same view of the directory hierarchy. We might also wish to consider whether the name space on all machines should have a global root directory (a.k.a. super root) so that files can be accessed as, for example, `//server/path`. This is a model that was adopted by the Apollo Domain System, an early distributed file system, and more recently by the web community in the construction of a uniform resource locator (URL).

In considering our goals in name resolution, we must distinguish between *location transparency* and *location independence*. By **location transparency** we mean that the path name of a file gives no hint to where the file is located. For instance, we may refer to a file as `//server1/dir/file`. The server (`server`) can move anywhere without the client caring, so we have location transparency. However, if the file moves to `server2` things will not work. If we have **location independence**, the files can be moved without their names changing. Hence, if machine or server names are embedded into path names we do not achieve location independence.

It is desirable to have **access transparency**, so that applications and users can access remote files just as they access local files. To facilitate this, the remote file system name space should be syntactically consistent with the local name space. One way of accomplishing this is by redefining the way files are named and require an explicit syntax for identifying remote files. This can cause legacy applications to fail and user discontent (users will have to learn a new way of naming their files). An alternate solution is to use a file system mounting mechanism to overlay portions of another file system over a node in a local directory structure. Mounting is used in the local environment to construct a uniform name space from separate file systems (which reside on different disks or partitions) as well as incorporating special-purpose file systems into the name space (e.g. `/proc` on many UNIX systems allows file system access to processes). A remote file system can be mounted at a particular point in the local directory tree. Attempts to access files and directories under that node will be directed to the driver for that file system.

To summarize, our naming options are:

- machine and path naming (`machine:path`, `./machine/path`).
- mount remote file systems onto the local directory hierarchy (merging the two name spaces).
- provide a single name space which looks the same on all machines.

The first two of these options are relatively easy to implement.

Types of names

When we talk about file names, we refer to **symbolic names** (for example, `server.c`). These names are used by people (users or programmers) to refer to files. Another "name" is the identifier used by the system internally to refer to a file. We can think of this as a **binary name** (more precisely, as an address). On most UNIX file systems, this would be the device number and inode number. On MS-DOS systems, this would be the drive letter and FAT index.

Directories provide a mapping from symbolic names to file addresses (binary names). Typically, one symbolic name maps to one file address. If multiple symbolic names map onto one binary name, these are called **hard links**. On inode-based file systems (e.g., most UNIX systems), hard links must exist within the same device since the address (inode) is unique only on that device. On MS-DOS systems, they are not supported because file attributes are stored with the name

of the file. Having two symbolic names refer to the same data will cause problems in synchronizing file attributes (how would you locate other files that point to this data?). A hack to allow multiple names to refer to the same file (whether its on the same device or a different device) is to have the symbolic name refer to a single file address but that file may have an attribute to tell the system that *its* contents contain a symbolic file name that should be dereferenced. Essentially, this adds a level of indirection: access a file which contains another file name, which references the file attributes and data. These files are known as **symbolic links**. Finally, it is possible for one symbolic name to refer to multiple file addresses. This doesn't make much sense on a local system[1], but can be useful on a networked file system to provide fault tolerance or enable the system to use the file address which is most efficient.

Semantics of file sharing

The analysis of file sharing semantics is that of understanding how files behave. For instance, on most systems, if a *read* follows a *write*, the *read* of that location will return the values just written. If two *write*s occur in succession, the following *read* will return the results of the *last* write. File systems that behave this way are said to observe **sequential semantics**.

Sequential semantics can be achieved in a distributed system if there is only one server and clients do not cache data. This can cause performance problems since clients will be going to the server for every file operation (such as single-byte reads). The performance problems can be alleviated with client caching. However, now if the client modifies its cache and another client reads data from the server, it will get obsolete data. Sequential semantics no longer hold.

One solution is to make all the writes write-through to the server. This is inefficient and does not solve the problem of clients having invalid copies in their cache. To solve this, the server would have to notify all clients holding copies of the data.

Another solution is to relax the semantics. We will simply tell the users that things do not work the same way on the distributed file system as they did on the local file system. The new rule can be "changes to an open file are initially visible only to the process (or machine) that modified it." These are known as **session semantics**.

Yet another solution is to make all the files immutable[2]. That is, a file cannot be open for modification, only for reading or creating. If we need to modify a file, we'll create a completely new file under the old name. Immutable files are an aid to replication but they do not help with changes to the file's contents (or, more precisely, that the old file is obsolete because a new one with modified contents succeeded it). We still have to contend with the issue that there may be another process reading the old file. It's possible to detect that a file has changed and start failing requests from other processes.

A final alternative is to use atomic transactions. To access a file or a group of files, a process first executes a *begin transaction* primitive to signal that all future operations will be executed indivisibly. When the work is completed, an *end transaction* primitive is executed. If two or more

[1] It really does make sense in a way. In the late 1980's, David Korn created a file system that allowed multiple directories to be mounted over the same directory node. The Plan 9 operating system later adopted this technique and called it *union mounts*. One thing it is useful for is getting rid of the PATH environment variable in searching for executables; the executables are always found in `/bin`. The name is resolved by searching through the file systems (directories) mounted on that node in a last-mounted, first-searched order.

[2] The Bullet server on the Amœba operating system is an example of a system that uses immutable files.

transactions start at the same time, the system ensures that the end result is as if they were run in some sequential order. All changes have an all or nothing property.

File usage patterns

It's clear that we cannot have the best of all worlds and that compromises will have to be made between the desired semantics and efficiency (efficiency encompasses increasing client performance, reducing network traffic, and minimizing server load). To design or select a suitable distributed file system, it is important to understand the usage patterns within a file system. A study was made by Satyanarayanan in 1981 which exhibited the following:

- Most files are under 10K bytes in size. This suggests that it may be feasible to transfer entire files (a simpler design). However, a file system should still be able to support large files.
- Most files have short lifetimes (many files are temporary files created by editors and compilers). It may be a good idea to keep these files local and see if they will be deleted soon.
- Few files are shared. While sharing is a big issue theoretically, in practice it's hardly done. We may choose to accept session semantics and not worry too much about the consequences.
- Files can be grouped into different classes, with each class exhibiting different properties:

  system binaries:
  read/only, widely distributed. These are good candidates for replication.

  compiler and editor temporary files :
  short, unshared, disappear quickly. We'd like to keep these local if possible.

  mailboxes (email):
  not shared, frequently updated. We don't want to replicate these.

  ordinary data files:
  these may be shared.

System design issues

Name resolution

In looking up the pathname of a file (e.g. via the *namei* function in the UNIX kernel), we may choose to evaluate a pathname a component at a time. For example, for a pathname `aaa/bbb/ccc`, we would perform a remote lookup of `aaa`, then another one of `bbb`, and finally one of `ccc`) . Alternatively, we may pass the rest of the pathname to the remote machine as one lookup request once we find that a component is remote. The drawback of the latter scheme is (a) the remote server may be asked to walk up the tree by processing `..` (parent node) components and reveal more of its file system than it wants and (b) other components cannot be mounted underneath the remote tree on the local system. Because of this, component at a time evaluation is

generally favored but it has performance problems (a lot more messages). We may choose to keep a local cache of component resolutions.

Should servers maintain state?

This issue is a topic of passionate debate. A **stateless system** is one in which the client sends a request to a server, the server carries it out, and returns the result. Between these requests, *no client-specific information is stored on the server*. A **stateful system** is one where information about client connections is maintained on the server.

In a stateless system:

- Each request must be complete – the file has to be fully identified and any offsets specified.
- Fault tolerance: if a server crashes and then recovers, no state was lost about client connections because there was no state to maintain.
- No remote open/close calls are needed (they only serve to establish state).
- No wasted server space per client.
- No limit on the number of open files on the server; they aren't "open" – the server maintains no per-client state.
- No problems if the client crashes. The server does not have any state to clean up.

On a stateful system:

- requests are shorter (less info to send).
- better performance in processing the requests.
- idempotency works; cache coherence is possible.
- file locking is possible; the server can keep state that a certain client is locking a file (or portion thereof).

Caching

We can employ caching to improve system performance. There are four places in a distributed system where we can hold data:

1. on the server's disk
2. in a cache in the server's memory
3. in the client's memory
4. on the client's disk

The first two places are not an issue since any interface to the server can check the centralized cache. It is in the last two places that problems arise and we have to consider the issue of **cache consistency**. Several approaches may be taken:

write-through

What if another client reads its own cached copy? All accesses would require checking with the server first (adds network congestion) or require the server to maintain state on who has what files cached. Write-through also does not alleviate congestion on writes.

delayed writes

Data can be buffered locally (where consistency suffers) but files can be updated periodically. A single bulk write is far more efficient than lots of little writes every time any file contents are modified. Unfortunately the semantics become ambiguous.

write on close
: This is admitting that the file system uses session semantics.

centralized control
: Server keeps track of who has what open in which mode. We would have to support a stateful system and deal with signaling traffic.

Distributed File Systems: case studies

It is clear that compromises have to be made in a practical design. For example, we may trade-off file consistency for decreased network traffic. We may choose to use a connectionless protocol to enable clients to survive a server crash gracefully but sacrifice file locking. This section examines a few distributed file systems.

Network File System (NFS)

Sun's NFS is one of the most popular and widespread distributed file systems in use today. The design goals of NFS were:

- Any machine can be a client and/or a server.
- NFS must support diskless workstations (that are booted from the network). Diskless workstations were Sun's major product line.
- Heterogeneous systems should be supported: clients and servers may have different hardware and/or operating systems. Interfaces for NFS were published to encourage the widespread adoption of NFS.
- high performance: try to make remote access as comparable to local access through caching and read-ahead.

From a transparency point of view NFS offers:

access transparency
: Remote (NFS) files are accessed through normal system calls; the protocol is implemented under the VFS (vnode) layer in UNIX.

location transparency
: The client adds remote file systems to its local name space via *mount*. File systems must be exported at the server. The user is unaware of which directories are local and which are remote. The location of the mount point in the local system is up to the client's administrator.

failure transparency
: NFS is stateless; UDP is used as a transport. If a server fails, the client retries.

performance transparency
: Caching at the client will be used to improve performance

no migration transparency
: The client mounts machines from a server. If the resource moves to another server, the client must know about the move.

no support for Unix semantics
: NFS is stateless, so stateful operations such as file locking are a problem. All UNIX file system controls may not be available.

devices?
: Since NFS had to support diskless workstations, where *every* file is remote, remote device files had to refer to the client's local devices. Otherwise there would be no way to access local devices in a diskless environment.

NFS protocols

The NFS client and server communicate over remote procedure calls (Sun's RPC) using two protocols: the *mounting protocol* and the *directory and file access protocol*. The mounting protocol is used to request a access to an exported directory (and the files and directories within that file system under that directory). The directory and file access protocol is used for accessing the files and directories (e.g. read/write bytes, create files, etc.). The use of RPC's external data representation (XDR) allows NFS to communicate with heterogeneous machines. The initial design of NFS ran only with remote procedure calls over UDP. This was done for two reasons. The first reason is that UDP is somewhat faster than TCP but does not provide error correction (the UDP header provides a checksum of the data and headers). The second reason is that UDP does not require a connection to be present. This means that the server does not need to keep per-client connection state and there is no need to reestablish a connection if a server was rebooted.

The lack of UDP error correction is remedied in the fact that remote procedure calls have built-in retry logic. The client can specify the maximum number of retries (default is 5) and a timeout period. If a valid response is not received within the timeout period the request is re-sent. To avoid server overload, the timeout period is then doubled. The retry continues until the limit has been reached. This same logic keeps NFS clients fault-tolerant in the presence of server failures: a client will keep retrying until the server responds.

mounting protocol

The client sends the pathname to the server and requests permission to access the contents of that directory. If the name is valid and exported (listed in `/etc/dfs/sharetab` on System V release 4 versions of UNIX, and `/etc/exports` on many other versions) the server returns a **file handle** to the client. This file handle contains all the information needed to identify the file on the server: {*file system type*, *disk ID*, *inode number*, *security info*}.

Mounting an NFS file system is accomplished by parsing the path name, contacting the remote machine for a file handle, and creating an in-core vnode at the mount point. A vnode points to an inode for a local UNIX file or, in the case of NFS, an rnode. The rnode contains specific information about the state of the file from the point of view of the client. Two forms of mounting are supported:

static
: In this case, file systems are mounted with the *mount* command (generally during system boot).

automounting
: One problem with static mounting is that if a client has a lot of remote resources mounted, boot-time can be excessive, particularly if any of the remote systems are not responding and the client keeps retrying. Another problem is that each machine has to maintain its own name space. If an administrator wants all machines to have the same name space, this can be an administrative headache. To combat these problems the *automounter* was introduced.

The automounter allows mounts and unmounts to be performed in response to client requests. A set of remote directories is associated with a local directory. None are mounted initially. the first time any of these is referenced, the operating system sends a message to *each* of the servers. The first reply wins and that file system gets mounted (it is up to the administrator to ensure that all file systems are the same). To configure this, the automounter relies on mapping files that provide a mapping of client pathname to the server file system. These maps can be shared to facilitate providing a uniform naming space to a number of clients.

directory and file access protocol

Clients send RPC messages to the server to manipulate files and directories. A file is accessed by performing a *lookup* remote procedure call. This returns a file handle and attributes. It is *not* like an *open* in that no information is stored in any system tables on the server. After that, the handle may be passed as a parameter for other functions. For example, a *read*(*handle*, *offset*, *count*) function will read *count* bytes from location *offset* in the file referred to by *handle*.

The entire directory and file access protocol is encapsulated in sixteen functions[3]. These are:

null	no-operation but ensure that connectivity exists
lookup	lookup the file name in a directory
create	create a file or a symbolic link
remove	remove a file from a directory
rename	rename a file or directory
read	read bytes from a file
write	write bytes to a file
link	create a link to a file
symlink	create a symbolic link to a file
readlink	read the data in a symbolic link (do not follow the link)
mkdir	create a directory
rmdir	remove a directory
readdur	read from a directory
getattr	get attributes about a file or directory (type, access and modify times, and access permissions)
setattr	set file attributes
statfs	get information about the remote file system

Accessing files

Files are accessed through conventional system calls (thus providing access transparency). If you recall conventional UNIX systems, a hierarchical pathname is dereferenced to the file location with a kernel function called *namei*. This function maintains a reference to a current directory,

[3] These functions are present in versions 2 and 3 of the NFS protocol. Version 3 added six more functions.

looks at one component and finds it in the directory, changes the reference to that directory, and continues until the entire path is resolved. At each point in traversing this pathname, it checks to see whether the component is a *mount point*, meaning that name resolution should continue on another file system. In the case of NFS, it continues with remote procedure calls to the server hosting that file system.

Upon realizing that the rest of the pathname is remote, *namei* will continue to parse one component of the pathname at a time to ensure that references to `..` and to symbolic links become local if necessary. Each component is retrieved via a remote procedure call which performs an NFS *lookup*. This procedure returns a file handle. An in-core *rnode* is created and the VFS layer in the file system creates a *vnode* to point to it.

The application can now issue *read* and *write* system calls. The file descriptor in the user's process will reference the in-core vnode at the VFS layer, which in turn will reference the in-core rnode at the NFS level which contains NFS-specific information, such as the file handle. At the NFS level, NFS *read*, *write*, etc. operations may now be performed, passing the file handle and *local* state (such as file offset) as parameters. No information is maintained on the server between requests; it is a *stateless* system.

The RPC requests have the user ID and group ID number sent with them. This is a security hole that may be stopped by turning on RPC encryption.

Performance

NFS performance was generally found to be slower than accessing local files because of the network overhead. To improve performance, reduce network congestion, and reduce server load, file data is cached at the client. Entire pathnames are also cached at the client to improve performance for directory lookups.

server caching

Server caching is automatic at the server in that the same buffer cache is used as for all other files on the server. The difference for NFS-related writes is that they are all *write-through* to avoid unexpected data loss if the server dies.

client caching

The goal of client caching is to reduce the amount of remote operations. Three forms of information are cached at the client: file data, file attribute information, and pathname bindings. We cache the results of *read*, *readlink*, *getattr*, *lookup*, and *readdir* operations. The danger with caching is that inconsistencies may arise. NFS *tries* to avoid inconsistencies (and/or increase performance) with:

- *validation* - if caching one or more blocks of a file, save a time stamp. When a file is opened or if the server is contacted for a *new* data block, compare the last modification time. If the remote modification time is more recent, invalidate the cache.
- Validation is performed every three seconds on open files.
- Cached data blocks are assumed to be valid for three seconds.
- Cached directory blocks are assumed to be valid for thirty seconds.
- Whenever a page is modified, it is marked *dirty* and scheduled to be written (asynchronously). The page is flushed when the file is closed.

Transfers of data are done in large chunks; the default is 8K bytes. As soon as a chunk is received, the client immediately requests the next 8K-byte chunk. This is known as read-ahead. The assumption is that most file accesses are sequential and we might as well fetch the next block of data while we're working on our current block, anticipating that we'll likely need it. This way, by the time we do, it will either be there or we don't have to wait too long for it since it's on its way.

Problems

The biggest problem with NFS is file consistency. The caching and validation policies do not guarantee session semantics.

NFS assumes that clocks between machines are synchronized and performs no clock synchronization between client and server. One place where this hurts is in distributed software development environments. A program such as *make,* which compares times of files (such as object and source) to determine whether to regenerate them, can either fail or give confusing results.

Because of its stateless design, open with append mode cannot be guaranteed to work. You can open a file, get the attributes (size), and then write at that offset, but you'll have no assurance that somebody else did not write to that location after you received the attributes. In that case your write will overwrite the other once since it will go to the old end-of-file byte offset.

Also because of its stateless nature, file locking cannot work. File locking implies that the server keeps track of which processes have locks on the file. Sun's solution to this was to provide a separate process (a lock manager) that *does* keep state.

One common programming practice under UNIX file systems for manipulating temporary data in files is to open a temporary file and then remove it from the directory. The name is gone, but the data persists because you still have the file open. Under NFS, the server maintains no state about remotely opened files and removing a file will cause the file to disappear. Since legacy applications depended on this, Sun's solution was to create a special hack for UNIX: if the same process that has a file open attempts to delete it, it is instead moved to a temporary name and deleted on close. It's not a perfect solution, but it works well.

Permission bits might change on the server and disallow future access to a file. Since NFS is stateless, it has to check access permissions each time it receives an NFS request. With local file systems, once access is granted initially, a process can continue accessing the file even if permissions change.

By default, no data is encrypted and Unix-style authentication is used (used ID, group ID). NFS supports two additional forms of authentication: Diffie-Hellman and Kerberos. However, data is never encrypted and user-level software should be used to encrypt files if this is necessary.

More fixes

The original version of NFS was released in 1985, with version 2 released around 1988. In 1992, NFS was enhanced to version 3 (SunOS ≥5.5). Several changes were added to enhance the performance of the system:

1. NFS was enhanced to support TCP. UDP caused more problems over wide-area networks than it did over LANs because of errors. To combat that and to support larger data transfer sizes, NFS was modified to support TCP as well as UDP. To minimize connection setup, all traffic can be multiplexed over one TCP connection.
2. NFS always relied on the system buffer cache for caching file data. The buffer cache is often not very large and useful data was getting flushed because of the size of the cache. Sun introduced a caching file system, *CacheFS*, that provides more caching capability by using the disk.

Memory is still used as before, but a local disk is used if more cache space is needed. Data can be cached in chunks as large as 64K bytes and entire directories can be cached.

3. NFS was modified to support asynchronous writes. If a client needed to send several write requests to a server, it would send them one after another. The server would respond to a request only *after* the data was flushed to the disk. Now multiple writes can be collected and sent as an aggregate request to the server. The server does not have to ensure that the data is on stable storage (disk) until it receives a *commit* request from the client.
4. File attributes are returned with each remote procedure call now. The overhead is slight and saves clients from having to request file attributes separately (which was a common operation).
5. Version 3 allows 64-bit rather than the old 32-bit file offsets (supporting file sizes over 18 million terabytes).
6. An enhanced lock manager was added to provide *monitored locks*. A status monitor monitors hosts with locks and informs a lock manager of a system crash. If a server crashes, the status monitor reinstates locks on recovery. If a client crashes, all locks from that client are freed on the server.
7. A few more NFS functions were added:

access	check access permissions for a server
mknod	create a special device file on a server
readdirplus	extended read from a directory
fsinfo	get file system state information (static, as opposed to *fsstat*, which returns dynamic information)
commit	commit the cached data on the server to stable storage (e.g. disk)

Remote File Sharing (RFS)

At the time that Sun was developing NFS for SunOS, AT&T was developing its own distributed file system for UNIX System V. This file system was almost everything that NFS was not. NFS had a very rudimentary and generic interface (the 16 remote procedure calls). One of its goals was that NFS servers could be written on many operating systems. RFS was committed to preserving UNIX System V file system semantics *exactly*. It would not provide just a remote file system, but rather a remote UNIX System V file system. As such, it would not be an easy feat to build an RFS server on non-UNIX machines.

To provide UNIX System V file semantics, RFS was designed to be both stateful and connection-oriented. A client would establish a connection with a server upon mounting a remote file system and the server would maintain state on which files are open and which data blocks are in use. The server also detects client crashes, so cache consistency is guaranteed.

Resource access on RFS is different from NFS. With NFS, a client machine has to know which directories are exported on which machines. This information is obtained only by going to that machine and examining the file containing exported file systems (or asking an administrator or someone who knows). RFS chose to add a level of indirection by employing a *name server*. A machine could *advertise* resources to the name server with some symbolic name. Any other machine could then query the name server for resources advertised within that *domain* (group) of machines. For fault tolerance, the several name servers could run concurrently with one designated as a primary. Should the primary name server die, the remaining name servers would elect a new acting primary name server. This adds a degree of transparency in that a client need not know the server's name or the location of the resource on the server.

Another major difference between RFS and NFS is the handling of device files. In UNIX systems, the file name space is used for access to devices: the inode simply has a flag stating that

the file is really a device and contains numbers to identify the kernel drivers. If one is accessing a remote file system and accesses a remote device file, should the access be to a local or to a remote device? AT&T's response was that it would be useful if access to a device on a remote file system would access the remote device. After all, you can always access your local devices through a local file system. Accessing remote devices allows for easy sharing of devices such as backup tape drives and printers.

While Sun may have thought that accessing remote devices would be convenient, it had more important things to think about, namely its hardware. Sun's initial claim to fame was the development of the diskless workstation. A diskless workstation loads the operating system from some server and, upon booting, mounts *all* its file systems as remote (NFS) file systems since it does not have a local disk. In this environment it is important that local hardware could be accessed (the keyboard, mouse, display, etc.). The only way this could be done is by making a rule that remote device files really refer to local devices. While this solves the diskless problem, it does not allow for sharing devices through the file system. It also has the problem that if a client's kernel device numbers for a particular device differ from the server's, the wrong device may be accessed.

RFS or NFS? Which is better? The answer is *it depends*. If support for heterogeneous operating systems is desired, NFS is the winner. On the other hand, if support for UNIX file system semantics is important (cache coherency, file locking that works) then RFS wins. If complete support of every feature in a UNIX System V file system is required, RFS provides it; NFS caters to the lowest-common-denominator of file system operations. It support of remote devices is required, RFS can provide it while NFS will attempt to access the corresponding local device. RFS, because of its connection-oriented nature, is sensitive to server crashes. If an application has a remote file open and the server crashes, the mounted remote file system is unmounted and any open or closed files are immediately inaccessible. With NFS, the NFS client simply keeps issuing NFS RPC requests until the server comes up again. Looking at the marketplace, NFS has completely overshadowed RFS, primarily because of its easy portability, the popularity of Suns in the late 1980s over machines running UNIX, System V, and the fault tolerance of NFS.

Andrew File System (AFS)

The goal of the Andrew File System (from Carnegie Mellon University, then a product of Transarc Corp., and now part of the Transarc division of IBM and available via the IBM public license) was to support information sharing on a large scale (thousands to 10000+ users). There were several incarnations of AFS, with the first version being available around 194, AFS-2 in 1986, and AFS-3 in 1989).

The assumptions about file usage were:

- most files are small
- reads are much more common than writes
- most files are read/written by one user
- files are referenced in bursts (locality principle). Once referenced, a file will probably be referenced again.

From these assumptions, the original goal of AFS was to use **whole file serving** on the server (send an entire file when it is opened) and **whole file caching** on the client (save the entire file onto a local disk). To enable this mode of operation, the user would have a cache partition on a local disk devoted to AFS. If a file was updated then the file would be written back to the server when the application performs a *close*. The local copy would remain cached at the client.

Implementation

The client's machine has one disk partition devoted to the AFS cache (for example, 100M bytes, or whatever the client can spare). The client software manages this cache in an LRU (least recently used) manner and the clients communicate with a set of trusted servers. Each server presents a **location-transparent** UNIX (hierarchical) file name space to its clients. On the server, each physical disk partition contains files and directories that can be grouped into one or more **volumes**. A volume is nothing mmore than an administrative unit of organization (e.g., a user's home directory, a local source tree). Each volume has a directory structure (a rooted hierarchy of files and directories) and is given a name and ID. Servers are grouped into administrative entities called **cells**. A cell is a collection of servers, administrators, clients, and users. Each cell is autonomous but cells may cooperate and present users with one *uniform name space*. The goal is that *every client* will see the same name space (by convention, under a directory `/afs`). Listing the directory `/afs` shows the participating cells (e.g., `/afs/mit.edu`).

Each file and directory is identified by three 32-bit numbers:

volume ID:
: This identifies the volume to which the object belongs. The client caches the binding between volume ID and server, but the server is responsible for maintaining the bindings.

vnode ID:
: This is the "handle" (vnode number) that refers to the file on a particular server and disk partition (volume).

uniquifier:
: This is a unique number to ensure that the same vnode IDs are not reused.

Each server maintains a copy of a database that maps a volume number to its server. If the client request is incorrect (because a volume moved to a different server), the server forwards the request. This provides AFS with migration transparence: volumes may be moved between servers without disrupting access.

Communication in AFS is with RPCs via UDP. Access control lists are used for protection; UNIX file permissions are ignored. The granularity of access control is directory based; the access rights apply to all files in the directory. Users may be members of groups and access rights specified for a group. Kerberos is used for authentication.

Cache coherence

The server copies a file to the client and provides a **callback promise**: *it will notify the client when any other process modifies the file*.

When a server gets an update from a client, it notifies all the clients by sending a ***callback*** (via RPC). Each client that receives the callback then invalidates the cached file. If a client that had a file cached was down, on restart, it contacts the server with the timestamps of each cached file to decide whether to invalidate the file. Note that if an process as a file open, it can continue using it, even if it has been invalidated in the cache. Upon close, the contents will still be propagated to the server. There is no further mechanism for coherency. AFS abides by **session semantics**.

Under AFS, read-only files may be replicated on multiple servers.

Whole file caching isn't feasible for very large files, so AFS caches files in 64K byte chunks (by default) and directories in their entirety. File modifications are propagated only on close. Directory modifications are propagated immediately.

AFS does not support byte-range file locking. Advisory file locking (query to see whether a file has a lock on it) is supported.

AFS Summary

AFS demonstrates that whole file (or large chunk) caching offers dramatically reduced loads on servers, creating an environment that scales well. The AFS file system provides a uniform name space from all workstations, unlike NFS, where the client mount each NFS file system at a client-specific location (the name space is uniform only under the `/afs` directory, however). Establishing the same view of the file name space from each client is easier than with NFS. This enables users to move to different workstations and see the same view of the file system.

Access permission is handled through control lists per directory, but there is no per-file access control. Workstation/user authentication is performed via the Kerberos authentication protocol using a trusted third party (more on this in the security section).

A limited form of replication is supported. Replicating read-only (and read-mostly at your own risk) files can alleviate some performance bottlenecks for commonly accessed files (e.g. password files, system binaries).

Coda

Coda is a descendent of AFS, also created at CMU (c. 1990-1992). Its goals are:

- Provide better support for replication of file volumes than offered by AFS. AFS' limited form (read-only volumes) of replication will be a limiting factor in scaling the system. We would like to support widely shared read/write files, such as those found in bulletin-board systems.
- Provide *constant* data availability in disconnected environments through hoarding (user-directed caching). This requires logging updates on the client and reintegration when the client is reconnected to the network. Such a scheme will support the mobility of PCs.
- Improve fault tolerance. Failed servers and network problems shouldn't seriously inconvenience users.

To achieve these goals, AFS was modified in two substantial ways:

1. File volumes can be replicated to achieve higher throughput of file access operations and improve fault tolerance.
2. The caching mechanism was extended to enable disconnected clients to operate.

Volumes can be replicated to group of servers. The set of servers that can host a particular volume is the **volume storage group** (VSG) for that volume. In identifying files and directories, a client no longer uses a volume ID as AFS did, but instead uses a **replicated volume ID**. The client performs a one-time lookup to map the replicated volume ID to a list of servers and local volume IDs. This list is cached for efficiency. Read operations can take place from any of these servers to distribute the load. A write operation has to be multicast to all *available* servers. Since some servers

may be inaccessible at a particular point in time, a client may be able to access only a subset of the VSG. This subset is known as the **Available Volume Storage Group**, or **AVSG**.

Since some volume servers may be inaccessible, special treatment is needed to ensure that clients do not read obsolete data. Each file copy has a version stamp. Before fetching a file, a client requests version stamps for that file from *all* available servers. If some servers are found to have old versions, the client initiates a **resolution process** which tries to automatically resolve differences (administrative intervention may be required if the process finds problems that it cannot fix). Resolution is only initiated by the client. The process is handled entirely by the servers.

Disconnected operation

If a client's AVSG is empty, then the client is operating in a **disconnected operation** mode. If a file is not cached locally and is needed, nothing can be done: the system simply retries access and fails. For writes, however, the client does not report a failure of an update. Instead, the client logs the update locally in a Client Modification Log (CML). The user is oblivious to this. On reconnection, a process of **reintegration** with the server(s) commences to bring the server up to date. The CML is played back (the log playback is optimized so that only the latest changes are sent). The system tries to resolve conflicts automatically. This is not always possible (for example, someone may have modified the same parts of the file on a server while our client was disconnected). In cases where conflicts arise, user intervention is required to reconcile the differences.

To further support disconnected operation, it is desirable to cache all the files that will be needed for work to proceed when disconnected *and* keep them up to date even if they are not being actively used. To do this, Coda supports a **hoard database** that contains a list of these "important" files. The hoard database is constructed both by monitoring a user's file activity and allowing a user to explicitly specify files and directories that should be present on the client. The client frequently asks the server to send updates if necessary (that is, when it receives a callback).

Distributed File System (DFS)

DFS is the file system that is part of the Open Group's (formerly the Open Software Foundation or OSF) Distributed Computing Environment (DCE) and is a direct descendant of AFS. Like AFS, it assumes that:

- most file accesses are sequential
- most file lifetimes are short
- the majority of accesses are whole-file transfers
- the majority of accesses are to small files

With these assumptions, the conclusion is that file caching can reduce network traffic and server load. Since the studies on file usage in the early and mid 1980's, it was noticed that file throughput per user has increased dramatically and that typical file sizes became much larger.

DFS implements a strong consistency model (unlike AFS) with Unix semantics supported. This means that a read will return the effects of all writes that precede it. Cache consistency under DFS is maintained by the use of *tokens*.

A **token** is a guarantee from the server that a client can perform certain operations on the cached file. The server will revoke a token if another client attempts a conflicting operation. A

server grants and revokes tokens. It will grant any number of *read* tokens to clients but as soon as one client requests write access, the server will revoke all outstanding *read* and *write* tokens and issue a single *write* token to the requestor. This token scheme makes long term caching possible (it is not under NFS). Caching is in units of chunk sizes that range from 8K to 256K bytes. Caching is both in client memory and on the disk. DFS also employs read-ahead (similar to NFS) to attempt to bring additional chunks off the file to the client before they are needed.

DFS is integrated with DCE security services. File protection is via access control lists (ACL) and all communication between client and server is via authenticated remote procedure calls.

Server Message Block (SMB)

SMB is a protocol for sharing files, devices, and communication abstractions (such as named pipes or mailslots). It was created by Microsoft and Intel in 1987 and evolved over the years.

SMB is a client-server request-response protocol. Servers make file systems and other resources available to clients and clients access the shared file systems (and printers …) from the servers. The protocol is connection-oriented and requires either Microsoft's IPX/SPX or NetBIOS[4] over either TCP/IP or NetBEUI. A typical session proceeds as follows:

1. client sends a *negprot* SMB to the server. This is a protocol negotiation request.
2. the server respons with a version number of the protocol (and version-specific information, such as a maximum buffer size and naming information).
3. the client logs on (if required) by sending a *sesssetupX* SMB, which includes a username and password. It receives an acknowledgement or failure from the server. If successful, the server sends back a user ID (UID) of the logged-on user. This UID must be submitted with future requests.
4. The client can now connect to a tree. It sends a *tcon* (or *tconX*) SMB with the network name of the shared resource to request access to the resource. The server responds with a tree identifier (TID) that the client will use for all future requests for that resource.
5. Now the client can send *open*, *read*, *write*, *close* SMBs.

Machine naming is restricted to 15-character NetBIOS names if NetBEUI or TCP/IP are used. Since SMB was designed to operate in a small local-area network, clients find out about resources either by being configured to know about the servers in their environment or by having each server periodically broadcast information about its presence. Clients would listen for these broadcasts and build lists of servers. This is fine in a LAN environment but does not scale to wide-area networks (e.g. a TCP/IP environment with multiple subnets or networks). To combat this deficiency, Microsoft introduced *browse servers* and the *Windows Internet Name Service* (*WINS*).

The SMB security model has two levels:

1. Share level — Protection is applied per "share" (resource). Each share can have a password. The client needs to know that password to be able to access all files under that share. This was the only security model in early versions of SMB and is the default under Windows 95.

2. User level — Protection is applied to individual files in each share based on user access rights. A user (client) must log into a server and be authenticated. The client is then provided with a UID which must be

[4] See RFC 1001, 1002 for a description of this protocol.

presented for all future accesses.

Microsoft, Compaq, SCO, and a number of other companies are currently developing a public version of the SMB protocol, called CIFS (Common Internet File System).

Common Internet File System (CIFS)

CIFS is an evolutionary step from SMB and attempts to provide a distributed file system that allows heterogeneous hardware and operating systems (as opposed to the Microsoft and Intel-centric SMB) to request file services over a network. It is based on the server message block protocol and draws from concepts in AFS and DFS).

CIFS supports:

- shared files
- byte-range locking
- coherent caching
- change notification
- replicated storage
- Unicode file names

For load sharing, replicated virtual volumes are supported. They appear as one volume from one server to the client but, in reality, may span multiple volumes and servers. Components can be moved to different servers without changing their name. To accomplish this, referrals are used as in AFS (a client request is redirected to the server that currently stores that data).

To support wide-area (slow) networks, CIFS allows multiple requests to be combined into a single message to minimize round-trip latencies.

CIFS is transport-independent but requires a reliable connection-oriented message-stream transport. Sharing is in units of directory trees or individual devices. For access, the client sends authentication information to the server (name and password). The granularity of authorization is up to the server (e.g. individual files or an entire directory tree).

The caching mechanism is one of the more interesting aspects of CIFS. It is well-understood that network load is reduced if the amount of times that the client informs the server of changes is minimized. This also minimizes the load on servers. An extreme optimization leads to session semantics. Since a goal in CIFS is to provide coherent caching (UNIX semantics), the realization is that client caching is safe if any number of clients are reading data. Read-ahead operations (prefetch) are also safe as long as other clients are only reading the file. Write-behind (delayed writes) is safe only if a single client is accessing the file. None of these optimizations is safe if multiple clients are writing the file. In that case, operations will have to go directly to the server.

To support this behavior, the server grants **opportunistic locks** (*oplocks*) to a client for each file that it is accessing. This is a slight refinement of the token granting scheme used in DFS. An oplock takes one of the following forms:

exclusive oplock	tells the client that it is the only one with the file open (for write). Local caching , read-ahead, and write-behind are allowed. The server must receive an update upon file close. If someone else opens the file, the server has the previous client break its oplock. The client must send the server any *lock* and *write* data and acknowledge that it no longer has the lock.

level II oplock	allows multiple clients to have the same file open as long as none are writeing to the file. It tells the client that there are multiple concurrent clients, none of whom have modified the file (read access). Local caching of reads as well as read-ahead are allowed. All other operations must be sent directly to the server.
batch oplock	allows the client to keep the file open on the server even if a local process that was using it has closed the file. A client requests a batch oplock if it expects that programs may behave in a way that generates a lot of traffic (accessing the same file over and over). This oplock tells the client that it is the only one with the file open. All operations may be done on cached data and data may be cached indefinitely.
no oplocks	tells the client that other clients may be writing data to the file: all requests other than reads must be sent to the server. Read operations may work from a local cache *only* if the byte range was locked by the client.

The server has the right to asynchronously send a message to the client changing the oplock. For example, a client may be granted an exclusive oplock initially since nobody else was accessing the file. Later on, when another client opened the same file for read, the oplock was changed to a level II oplock. When another client opened the file for writing, both of the earlier clients were sent a message revoking their oplocks.

References

NFS

The NFS Distributed File Service: NFS White Paper, Sun Microsystems, March 1995
http://www.sun.com/software/white-papers/wp-nfs/

RFC 1094: NFS: Network File System Protocol Specification, Sun Microsystems, March 1989
One place to get this is http://sunsite.auc.dk/RFC/rfc/rfc1094.html

AFS

IBM's Transarc division provides the commercial AFS product. Some overview info on AFS as well as DFS can be found at
http://www.transarc.ibm.com/Product/EFS/index.html

Carneigie Mellon's AFS Reference Page:
http://www.cs.cmu.edu/afs/andrew.cmu.edu/usr/shadow/www/afs.html

The AFS File System in Distributed Computing Environments, Transarc Corporation, 1996. (This paper includes a comparison of AFS with NFS.)

http://www.transarc.ibm.com/Library/whitepapers/AFS/afswp.html

CIFS

A Common Internet File System (CIFS/1.0) Protocol, Paul J. Leach, Microsoft. Preliminary Draft, posted December 19, 1997
http://www.thursby.com/cifs/file/

Coda

Information about the Coda project at CMU can be found at
http://www.coda.cs.cmu.edu/

An index of Coda papers and project updates, including information about *Odyssey*, a follow-on project to Coda is available at:
http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html

Coda: A Highly Available File System for a Distributed Workstation Environment, Satyanarayanan, M., Proceeding of the Second IEEE Workshop on Workstation Operating Systems, Sept. 1989,
http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/wwos2.ps.Z

Coda: A Resilient Distributed File System, Satyanarayan, M., Kistler, J.J, Siegel, E.H., IEEE Workshop on Workstation Operating Systems,
http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/wwos1-fulltext.html

DCE

A Distributed Computing Environment Framework: An OSF Perspective, Brad Curtis Johnson, OSF, June, 1991
http://www.opengroup.org/dce/info/papers/dev-dce-tp6-1.ps

Towards a Worldwide Distributed File System:: The OSF DCE File System as an example, Norbert Leser, DCE Evaluation Team, Open Software Foundation, September, 1990
http://www.opengroup.org/dce/info/papers/dev-dce-tp4-1.ps

Performance Characteristics of the DCE Distributed File Service, by Agustin Mena III and Carl Burnett, IBM Corp.
http://www.networking.ibm.com/dce/dcedfspf.html

High level description of DFS and a whole bunch of benchmarks against Sun's NFS.

SMB

The main web reference for Samba (SMB) is (most likely):

http://anu.samba.org/

Just what is SMB?, Richard Sharpe, v1.1, 14 May 1998
http://anu.samba.org/cifs/docs/what-is-smb.html

Miscellany

Networking Applications on UNIX System V Release 4, Michael Padovano, © 1993 Prentice Hall.

This is a good reference for architectural discussions of RFS and NFS as well as administration and application programming.

UNIX Network Programming, W. Richard Stevens, © 1990 Orentice Hall.

Distributed Operating Systems, Andrew Tanenbaum, ©; 1995 Prentice Hall.

WebNFS: The Filesystem for the Internet, Sun Microsystems, © April 1997,
http://www.sun.com/webnfs/wp-webnfs/