Operating Systems

Network File Systems

By Paul Krzyzanowski
December 1, 2010 [updated April 20, 2011]

Introduction

Sockets gave us the ability to send and receive messages between processes on different (or the same) machines. Remote procedure calls gave us a layer of software that uses sockets and a compiler to provide a functional interface to communication between processes. Using sockets, we created programs that allow us to log onto remote machines, execute commands (*rsh*, *telnet*, *rlogin*), and copy files among among them (*rcp*, *ftp*). These programs, however, have to be specifically run by the user and require the user to be fully aware of the remote machine and remote file name. Wouldn't it be great to create a system where remote files can look and feel just like a local file system?

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 (often a hierarchical one) into an internal identifier (e.g. inode, FAT index). They also provide **data access** services that allow one to query metadata (attributes) of a file and read and write the file's data. The file system is responsible for controlling access to files and for performing low-level operations such as buffering frequently-used data and issuing disk I/O requests. Our goal in designing a network-accessible file system is to create something that fits into the operating system and offers users transparent access to files that live on remote machines. Such file systems are referred to as **network attached storage** (**NAS**), networked/network file systems, or distributed file systems.

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.

File service types

To provide a remote system with file service, we will have to select one of two approaches to design. 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 (the local copy is accessed). 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. A file server (the machine that hosts the files) services these requests. 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.

When we look at file access as a set of network services, one important distinction is that of a **directory service** and a **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 (get attributes, read bytes, write bytes, etc).

A key component of network 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. On POSIX file systems (various variants of Unix, BSD, OS X, Linux), for example, the client module is implemented as a file system driver underneath the VFS layer of the

kernel. Instead of reading and writing disk blocks, it sends and receives network messages to/from file servers.

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 *writes* 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**. It seems like common sense and one rarely thinks about this.

Sequential semantics can be achieved with networked file systems 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 users that things do not work the same way on the network 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."* This behavior is known as **session semantics**.

Yet another solution is to make all the files **immutable**. That is, a file can never be opened 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. This is generally not a useful approach to dealing with changing files in a file system.

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 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.

Stateful or stateless design?

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. *State* may refer to any information that a server stores about a client: whether a file is open, whether a file is being modified, cached data on the client, etc.

In a stateless system:

- Each request must be complete — the file has to be fully identified and any offsets specified.
- If a server crashes and then recovers, no state was lost about client connections because there was no state to maintain. This creates a higher degree of fault tolerance.
- No remote open/close calls are needed (they only serve to establish state).
- There is no server memory devoted to storing per-client data.
- There is no limit on the number of open files on the server; they aren't "open" since the server maintains no per-client state.
- There are no problems if the client crashes. The server does not have any state to clean up.

In a stateful file system:

- Requests are shorter (there is less information to send).
- Cache coherence is poosible; the server can know which clients are caching which blocks of a file.
- With shorter requests and caching, one will generally see better performance in processing the requests.
- File locking is possible; the server can keep state that a certain client is locking a file (or portion thereof).

Although the list of stateless advantages is longer, history shows us that the clear winner is the stateful approach. The ability to maintain better cache coherence, lock files, and know whether files are open by remote clients are all incredibly compelling advantages.

Caching

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

- On the server's disk
- In a cache in the server's memory
- In the client's memory
- 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

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

Examples/Case studies

Network File System (NFS)

Sun's NFS is one of the most earliest but still one of the most widely used network 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 (that is, how seamless remote operations appear compared to local ones), NFS offers:

access transparency

Remote (NFS) files are accessed through normal system calls; the client protocol is implemented under the VFS layer.

location transparency

The client adds remote file systems to its local name space via the mount mechanism. File systems must be exported at the server. After the remote file system is mounted, the user is unaware of whether a file or directory is local or 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 is 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 file access 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 RPC) using two families of 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 that may have different byte ordering or word sizes. 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, although it 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 SunOS/System V release 4 versions of Unix, and `/etc/exports` on most 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-kernel VFS inode at the mount point. This is a pointer to an NFS *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

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 lists on the server. After a *lookup*, 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 (more were added later). These are:

null	no operation but ensure that one can access the server
lookup	look up 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
readdir	read from a directory
getattr	get attributes a bout a file or directory (type, access and modification times, access permissions)
setattr	set file attributes
statfs	get information about the remote file system

Accessing files

Because the NFS client is implemented as a file system type under VFS, no new system calls are needed to use NFS. Files are accessed through conventional system calls (thus providing access transparency). A hierarchical pathname is dereferenced to the file location with a kernel function called *namei*. This function maintains a reference to a current directory, 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 `..` (the parent directory) and to symbolic links become local if necessary. Each component is retrieved via a remote procedure call to the NFS *lookup* function. This function returns a file handle. An in-memory rnode is created and the VFS layer in the file system creates an in-memory VFS inode 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-memory inode at the VFS layer, which in turn will reference the 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 partially remedied by turning on RPC encryption. However, the NFS assumes that user and group numbers are the same among all machines.

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** operations to avoid unexpected data loss if the server dies. That is, the date must be actually written to the disk before the RPC funciton returns.

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 to the server 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. File offsets were extended to 64-bit values 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)

AFS

The goal of the Andrew File System (from Carnegie Mellon University, now available via the IBM public license) was to support information sharing on a *large* scale (thousands to 10,000+ users). There were several incarnations of AFS, with the first version being available around 194, AFS-2 in 1986, and AFS-3 in 1989). To focus on scalability, AFS made some assumptions about file usage:

- 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 goal of AFS was to use **whole file serving** on the server (send an entire file when it is opened by the client) 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 that is devoted to AFS. If a file is modified on the client, it would be

written back to the server when the application performs a *close*. The local copy would remain cached at the client until the server invalidates it.

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. Clients communicate with a set of trusted AFS file servers. Each server presents a location-transparent hierarchical file name space to its clients. On the server, each physical disk partition contains files and directories that can be grouped logically into one or more volumes. A **volume** is nothing more 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.

VFS inode

This is the *handle* that refers to the file on a particular server and disk partition (volume).

uniquifier

This is a unique number to ensure that the the identifier among different files is different even if the inode number gets reused at the server's file system.

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

Communication in AFS is with remote procedure calls 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. Note that even through remote procedure calls are used, the file system does not employ the *remote access model* of NFS, which employs frequent access to the server for even small read/write operations and does not use long-term caching.

The server allows the client to download a file when the client opens the file and provides a **callback promise**: *it will notify the client when any other process modifies the file*. This means that clients can feel confident using any files in their AFS cache (for years!) without fear of the data being invalid. When the file changes on the AFS server, the server goes through its list of clients that have the file cached and sends each of them a *callback promise*. Upon reciving this message, the client invalidates the file in the AFS cache.

If a client process has the file open when the AFS client receives an invalidation message, access continues even if the file is invalidated from the cache. This is no different than a removing a local file while a process is accessing it. The name is removed from the directory but the file is not removed until the last reference disappears. Moreover, if the process has been modifying the file, the modified file contents are always sent to the server when the file is closed. This means that, in the case of concurrent modifications, the last process to close the file wins: its version will be the one stored on the server and modifications from other clients will get lost. While this might be surprising when we're used to sequential semantics (which

we get on most local file systems), this is the expected behavior for **session semantics**. If processes want to guard against this, they need to ensure that they grab and check locks. If multiple processes really need to modify the same file concurrently, then AFS is not a solution.

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).

Microsoft SMB (CIFS)

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 (as well as mailslots, printers, and devices) from the servers. The protocol is connection-oriented and required either Microsoft's IPX/SPX or NetBIOS4 over either TCP/IP or NetBEUI (only TCP/IP is needed now).

Although SMBs aren't remote procedure calls in the general sense, the interface behaves exactly as a remote procedure call interface would. The file system uses a *remote access model* with limited caching on the clients. The top design priority was consistency and ensuring that the behavior of accessing remote files mimics that of accessing local files as closely as possible. Unlike NFS, SMB keeps state of remote open files. Unlike AFS, it does not support long-term caching and working strictly from a cache.

A typical client-server session proceeds as follows:

1. client sends a *negprot* SMB to the server. This is a protocol negotiation request.
2. The server responds 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 was restricted to 15-character NetBIOS names if NetBEUI or TCP/IP were

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 was the default under Windows 95. This model has been largely deprecated due to its low security.
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 presented for all future accesses.

Microsoft's file sharing implementation was not designed for interoperability among other system. The specifications were not disclosed publicly and, in fact, Microsoft never even came up with a name for their network file system. SMB is just the name of a data structure. The protocol has, of course, been reverse engineered years ago. For Linux/Unix systems, the effort has yielded Samba (http://samba.org/), a collection of software that allows machines to interact with Microsoft Windows clients and servers.

In the late 1990s, Microsoft partially specified some aspects of the protocol and renamed SMB to **Common Internet File System** (**CIFS**). Enhancements to the protocol included:

- 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. Microsoft wanted to move away from the need for its proprietary NetBIOS/NetBEUI API and run directly on TCP.

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 (and sequential 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. 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. This means that 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 writing 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 opera-tions 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 oplock	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.

The protocol still remains largely proprietary. The lastest version of CIFS (or SMB) is SMB 2.0, which was introduced with Microsof Vista in 2006.

References

- The NFS Distributed File Service: NFS White Paper (http://www.sun.com/software/white-papers/wp-nfs/), Sun Microsystems, March 1995
- RFC 1094: NFS: Network File System Protocol Specification (http://www.faqs.org/rfcs/rfc1094.html), Sun Microsystems, March 1989
- RFC 1813: NFS Version 3 Protocol Specification (http://www.faqs.org/rfcs/rfc1813.html), Sun Microsystems, March 1995
- RFC 3530: NFS version 4 Protocol (http://www.ietf.org/rfc/rfc3530.txt), Sun Microsystems & Network Appliance, Inc., April 2003
- RFC 2623: NFS Version 2 and Version 3 Security Issues and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5 (http://www.faqs.org/rfcs/rfc2623.html), Sun Micrososystems, 1999
- AFS distributed filesystem FAQ (http://www.faqs.org/faqs/afs-faq/), Internet FAQ Archives
- OpenAFS (http://www.openafs.org/)
- Andrew File System (http://en.wikipedia.org/wiki/Andrew_File_System), Wikipedia article.
- Samba.org project page (http://samba.org/)
- Server Message Block (http://en.wikipedia.org/wiki/Server_Message_Block), Wikipedia

article

- Just what is SMB? (http://anu.samba.org/cifs/docs/what-is-smb.html), Richard Sharpe, October 2002.
- Common Internet File System Technical Reference 1.00 (http://www.snia.org/tech_activities/CIFS), Storage Networking Industry Association (SNIA), March 2002.
