Distributed File Systems

G. Lettieri

10 Nov. 2016

In a distributed file system, users sitting at a machine can access files stored on other machines connected via a Local Area Network (LAN). In the typical setup a central, powerful machine acts as a file server for a set of clients. Centralizing the file system on the server has two main benefits: users may access their files from any client, and client file system management is simplified (e.g., the operating system may be installed and upgraded only once on the server, than used by all the clients). The latter benefit also applies to cloud data centers, where the virtual machines play the role of the clients. In this enviroment the central file server simplies file system management, exactly as in the non virtual case. In a virtualized environment, however, remotely accessed storage brings an additional benefit, namely that is simplifies the implementation of *virtual machine migration* (i.e., moving a virtual machine from one server to another, for load balancing or server maintainance reason). Indeed, if the virtual machine file system is stored in a local file, then this (possibly very big) file must be moved along with the machine during migration. If, instead, the VM file system is stored on a remote server and accessed through the network, we only need to make sure that the VM is able to reach the server from its new location after the migration, and no big file transfer is needed.

Many distributed file systems have been designed and built. Here, we are going to examine the Network File System (NFS) developed by Sun Microsystems in the '80s. NFS has gone through several revisions (the current one being NFSv4), but we will focus only on the original implementation.

1 NFS

Figure 1 shows the NFS architecture. NFS is defined as a *protocol* between a client and a server. The client issues Remote Procedure Calls (RPC) to the server. The NFS protocol defines the set of RPCs that the server must implement, including their parameters and the possible replies. NFS has been developed on BSD 2.2 and is now available on all Unix variants, including Linux. However, the protocol is independent of the operating system and it is also available on Windows. Moreover, the messages exchanged between the client and the server must use the XDR (eXternal Data Representation) format, which makes the protocol independent also from the physical architecture of the

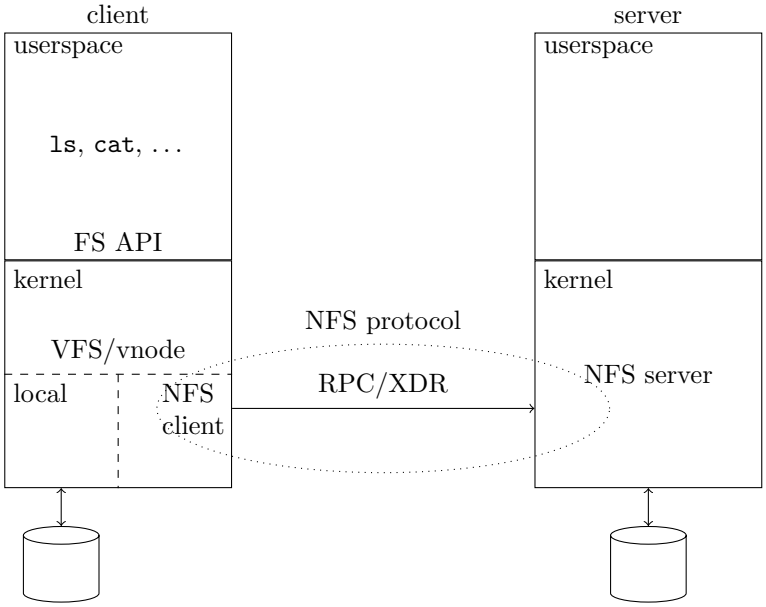

Figure 1: NFS architecture

machines (e.g., little-endian versus big-endian).

One of the design goals of NFS was to let the clients access local and remote files as uniformly as possible. If the client is a Unix machine, the user may mount a remote file system on a directory and then use the remote files (almost) exactly like the local ones. In particular, no userspace program (like `ls`, `cat`, etc.) needs to be modified in order to use the remote files. This is typically implemented by defining an abstract interface inside the kernel (called VFS/vnode in the original Sun implementation) through which all file operations are performed. The kernel provides at least two implementations of the interface: one is used to access local files by accessing the local storage in the usual way; the second one is used to access remote files by translating the operations into RPCs sent to the NFS server. Sun used synchronous RPCs (the clients blocks until it receives a reply from the sever) which further simplified the implementation, since synchronous RPCs behave much like normal function calls. On the server part, the NFS server runs in a kernel thread and listens and replies to the client's RPCs.

Another design goal of NFS was to simplify crash recovery. In a distributed system one must always assume that machines need to be rebooted for whatever reason (e.g., a crash). When this will eventually happen, the system must be able to recover, possibly as quickly as possible and avoiding any data loss. Crash recovery in NFS is simplified by adopting the following design decisions:

1. The server is *stateless*: its volatile memory does not store any information

about clients between two different RPCs;

2. RPCs are *idempotent*: the effect of issuing the same RPC several times in a row is the same as issuing it only once.

Since there is no state to recover, a crashed server does not need to do anything after a reboot. Idempotency of the RPCs is useful for the following reason: clients waiting for a reply from the server have no way to know if the server has crashed, or if it is simply slow, or if there is some problem in the network. This means that clients do not know whether a request that is taking a lot of time has actually been lost, or is going to be completed. However, since the RPCs are idempotent, they may simply send the request again after a timeout: no problem may arise from the possible duplication of the request.

1.1 The protocol

RFC1094 defines 16 RPC procedures that any NFS server must implement (plus a couple that came from NFSv1 and were already declared obsolete by v2). These procedures refer to files and directories using *handles*, which are identifiers whose content is only meaningful for the server and must be considered opaque by the clients.

An NFS server may *export* (i.e., make available to clients via the NFS protocol) any number of its local file-system subtrees. A client must first *mount* an NFS-exported file-system using a *mount protocol*. Since the details of this operation are highly system-dependent, the mount protocol is not part of the NFS specification, so that each system may define its own. The purpose of the mount protocol is to check any credentials and permissions and then give the client a first handle for the root of the exported file-system.

An handle for a directory d can be used, for example, in a READDIR RPC call to obtain a list of the contents of the directory, or in a LOOKUP RPC call, together with a filename, to obtain an handle for one of the files (or subdirectories) directly contained in d. Note that the filename passed sent in the LOOKUP RCP must be a single path component. This limitation was introduced to overcome the differences in path-syntax among existing operating systems (e.g., Unix slash vs DOS backslash). Therefore, to obatin an handle for a file with path `/home/foo/bar` the client must issue three LOOKUP RPC calls: a first one, using the root file handle, to obtain the handle for `home`; a second one, using the `home` handle, to obtain the handle for `foo`; finally, a last call, using the handle of `foo`, to obtain the handle for `bar`. Caching handles helps in mitigating the overhead. Once the client has obtained a file handle, it can issues the READ and WRITE RPC calls to obtain/update the file-contents. Other RPC calls are made available for renaming, unlinking, creating and deleting directories, getting and setting file attributes (e.g., access and modification times).

1.2 Implementation details

Several problems must be solved in order to let the (Unix) client applications access the remote file system with the same semantics as a traditional one, while keeping the NFS server stateless. The following problems arise from the fact that the Unix API is not stateless.

- In Unix you `open()` a file and obtain a file descriptor which you then may use to indirectly refer to the open file in all subsequent operation. The file descriptor is an index in small per-process table stored in the kernel. The NFS server, being stateless, has no such table, and therefore has no use for file descriptors. The LOOKUP RPC, instead, returns a file handle which contains all the information needed to find the file, and which must be sent by the client at each request. The NFS file handle has three parts: a file system id (to identify the file system on the server), an inode number (to identify the file in the file system) and a *generation number*. The latter number is needed since inode numbers are reused, and therefore there is no guarantee that the inode number passed to the client at open time still refers to the the same file when the client later tries to access it. With the generation number this ambiguity can be removed: the server stores one such number for each inode, and increments it whenever the inode is reused. When the server sends a file handle for an inode, the handle will contain the current generation number of that inode. If the inode is later reused, clients that still want to use a file handle for the old file will be notified of the error.

- Consider the `read()` system call: each time you call `read()` the kernel moves the read pointer, so that two identical read calls issued one after the other return different parts of the file. To implement this behaviour with a stateless server, the read pointer must be remembered by the NFS client and passed to the NFS server at each READ RCP call. The problem is the same for `write()`: the WRITE RPC must send to the client the bytes to be written and the starting offset in the file. Note that, in this way, the READ and WRITE RPC become automatically idempotent: if the same RPC is reissued, the final outcome is the same.

- The WRITE RPC must wait until the data has actually been written on the server hard disk before sending the reply to the client. The server cannot write the data into its buffer cache and immediately notify the completion to the client, since a crash at this point would cause a data loss. Essentially, the written data is part of the client state that the server is not allowed to remember in volatile memory between two RPCs.

- Deleting a file, a directory or a symlink in Unix is also inherently non idempotent, since trying to delete something that does not exist should return an error, and therefore two delete operations in a row are not equivalent to a single one. The RFC suggests to mitigate this problem by caching RPC replies in the server, on the assumption that retrasmissions

due to temporary netwkork failures are more frequent than server crashes. When the server receives a REMOVE RPC, say, it shoul first lookup its cache for a recent reply to the same request, and send the cached one if available.

- The Unix kernel only deletes a file when it is no longer open in any process. This means that a process may, e.g., create a file with `open()`, delete it with `unlink()`, and still be able to read and write from the deleted file. `unlink()` removes the file name from the file system, but it does not actually remove the file (i.e., it does not release the inode and the file blocks) until the file is `close()`ed or the process terminates (in which case the kernel closes all the process files). This feature has actually been used to implement temporary files which must be deleted when an application terminates: in this way the cleanup is done by the kernel, even if the application crashes. This feature is difficult to implement with a stateless server which does not know which files are opened by the clients. The solution is in fact only partial and not particularly elegant: the client knows the open files of each local process; if a process wants to `unlink()` an open file, the client asks the server to *rename* it (so that it kind of disappears) using the RENAME RPC, and only issues the REMOVE RPC when the file is finally closed. The solution is only partial since the client does not know whether the file is opened by a process running on another client.

1.3 Performance

The original implementation used UDP/IP to send the RPC messages, since that was faster than TCP/IP. The fact that UDP may lose messaged is not a problem, since the recovery protocol of the clients (just send the request again after a timeout) already addresses this.

To improve performance, the buffer cache of each client is still used: the result of READ RPCs are cached locally; local writes first go to the local buffer cache and only go to the server on flush (this reduces the number of WRITE RPCs and somehow mitigates the fact that this RPC is slow, since it must wait for the server to flush the data to disk). Of course, we must now consider the fact that a client caches may become stale if cached files are updated by another client. The solution implemented in NFS is to use the cached data for a limited amount of time (a few seconds) before asking the server for an updated version.

1.4 Limitations

Some problems are related to the distributed nature of the file systems. The user and group ids belong to processes running in the clients, but must be checked on the server. A simple solution for this is to adopt a global namespace for uids and gids for all the machines in the LAN. Another problem is related to the meaning of root on the clients and the server: is root on the client also root on the server?

In the simplest case, this is not true: root on the client is mapped to user nobody on the server. Note that this is only meant to prevent unintentional errors, and is not an effective measure against malicious clients. Indeed, the original NFS protocol is weak to malicious clients and servers, since there is essentially no authentication and file handles can be easily forged. Most of these problems have been addressed in the subsequent revisions of the protocol.

1.5 Extensions

The most notable change in NFSv3 was the possibility to have *unsafe* WRITE RPCs, where the server replies to the client before having flushed the buffers to stable storage. The client must later issue a COMMIT RCP when it wants the server to perform the flush. In this way, many WRITEs may be amortiezed over a single COMMIT. To prevent data loss, the client should keep the unsafely written blocks in its own memory (e.g., by not resetting the dirty bits of the corresponding blocks in its buffer cache) and ask for a COMMIT before removing them. The client must also detect when the server has been rebooted and, therefore, may have lost all the uncommitted blocks. For this purpose, the server sends a "version number" in each reply to the WRITE RPCs. The version number must change whenever the server reboots (it may be implemented by sampling the system time during boot). If the client sees that it still has uncommited blocks that were sent when the server had a different version number, it must re-send all of them.

NVFv4 introduced the possibility to have COMPOUND RPCs, where a single RPC call is made up of a sequence of RPCs, thus reducing the latency of typical sequences of operations like many LOOKUPs followed by a READ.