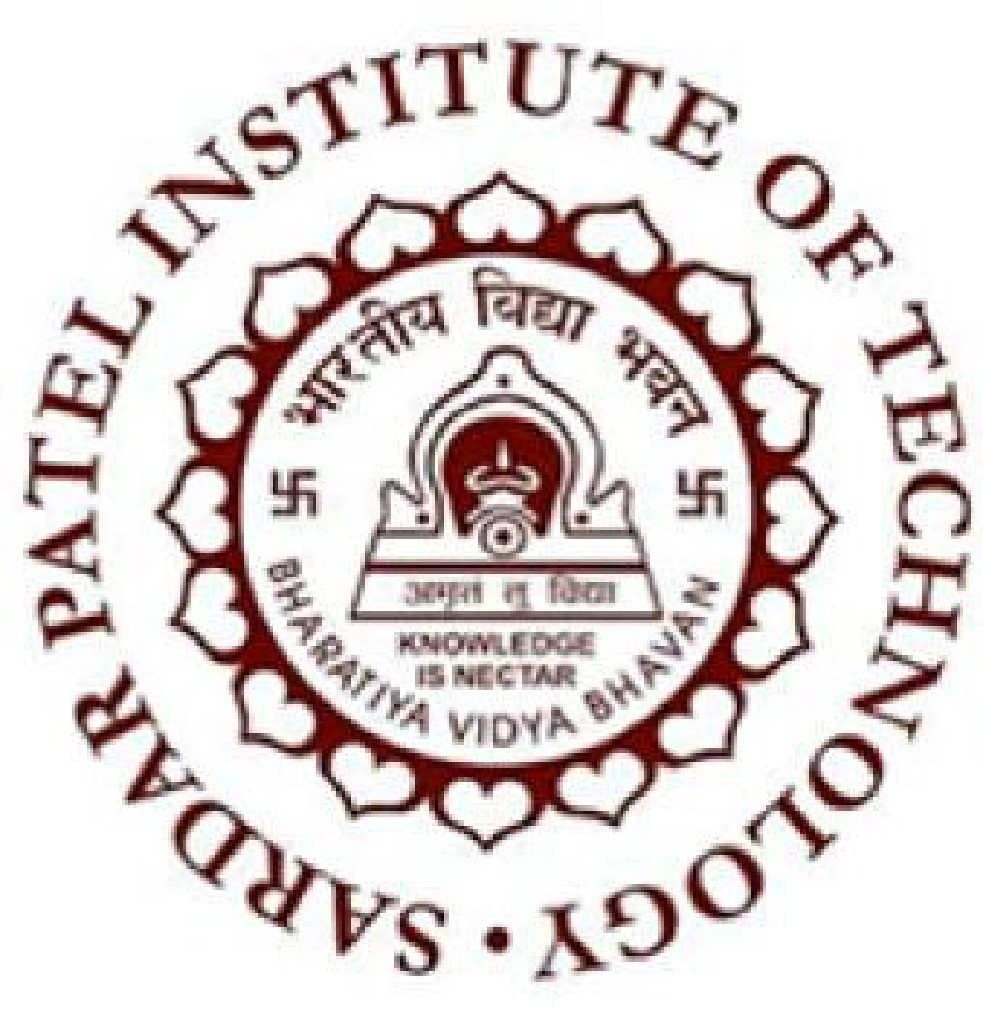
Sardar Patel Institute of Technology

Bhavan’s Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India (Autonomous College

Affiliated to University of Mumbai)



Distributed Computing ISE 1

Topic: Sun Network Distributed File System

Submitted by

Shivalik Pandita: 2020300049

Pratik Pujari: 2020300054

Divyesh Shah: 2020300060

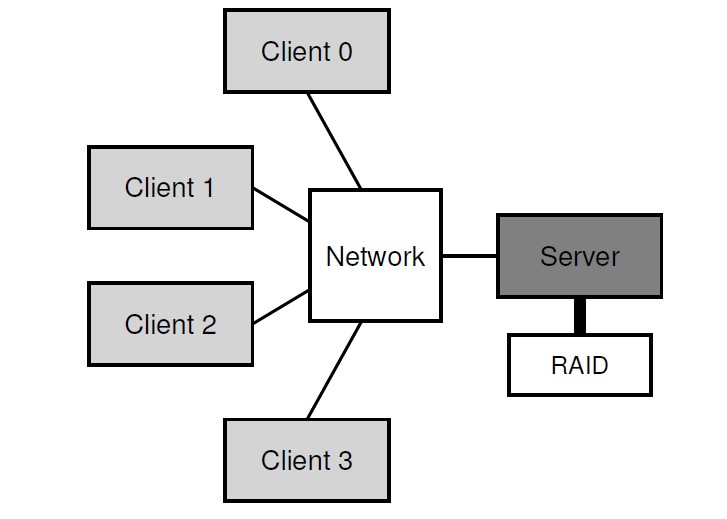
**Introduction:**

The Sun Network Filesystem (NFS) provides transparent, remote access to filesystems. Unlike many other remote file system implementations under UNIX, NFS is designed to be easily portable to other operating systems and machine architectures. It uses an External Data Representation (XDR) specification to describe protocols in a machine and system-independent way. NFS is implemented on top of a Remote Procedure Call package (RPC) to help simplify protocol definition, implementation, and maintenance.

In order to build NFS into the UNIX kernel in a way that is transparent to applications, we decided to add a new interface to the kernel which separates generic filesystem operations from specific file system implementations. The “filesystem interface” consists of two parts: the Virtual File System (VFS) interface defines the functions that can be done on a filesystem, while the virtual node (vnode) interface defines the operations that can be done on a file within that filesystem.

This new interface allows us to implement and install new filesystems in the same way as new device drivers are added to the kernel. This paper discusses the design and implementation of the file system interface in the UNIX kernel and the NFS virtual filesystem. We compare NFS to other remote file system implementations and describe some interesting NFS ports that have been done, including the IBM PC implementation under MS/DOS and the VMS server implementation. We also describe the user-level NFS server implementation which allows simple server ports without modification to the underlying operating system.

We conclude with some ideas for future enhancements. In this paper we use the term server to refer to a machine that provides resources to the network; a client is a machine that accesses resources over the network; a user is a person “logged in” at a client; an application is a program that executes on a client, and a workstation is a client machine that typically supports one user at a time.

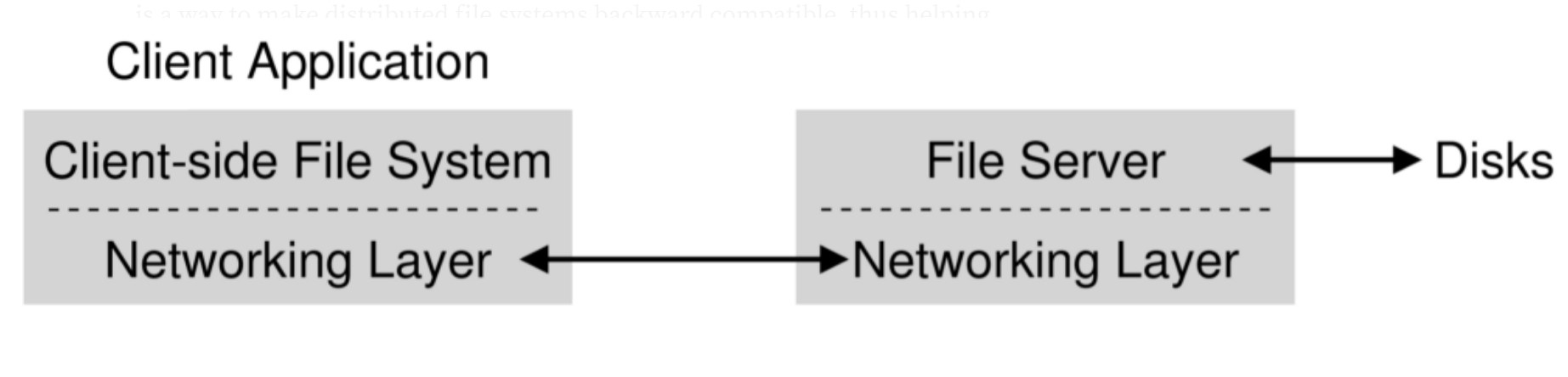


As you can see from the picture, the server has the disks, and clients send messages across a network to access their directories and files on those disks. Why do we bother with this arrangement? (i.e., why don’t we just let clients use their local disks?) Well, primarily this setup allows for easy sharing of data across clients. Thus, if you access a file on one machine (Client 0) and then later use another (Client 2), you will have the same view of the file system. Your data is naturally shared across these different machines. A secondary benefit is a centralized administration; for example, backing up files can be done from a few server machines instead of from a multitude of clients. Another advantage could be security; having all servers in a locked machine room prevents certain types of problems from arising.

**The Need for Distributed File System:**

The primary motivation for a distributed file system is sharing, i.e., the ability to access a file from multiple machines called clients. These clients access the files that are stored on one (or a few) servers. Another advantage of storing files on a centralized server (as opposed to on each individual client) is the ease of administration. Administration includes things like backing up the data, enforcing quotas across users, and security (the chapter notes security as a different aspect than administration, but I consider it to be a subset of administration in general).

A primary requirement for any distributed file system (like NFS) is to make its usage transparent to applications. Transparency implies that the client applications should be able to access data from the distributed file system in the same way as they access it from their local file system. One way to think about the requirement is that local file systems (and applications using local file systems) existed before distributed file systems and having transparency is a way to make distributed file systems backward compatible, thus helping their adoption.



From this simple overview, you should get a sense that there are two important pieces of software in a client/server distributed file system: the client-side file system and the file server. Together their behavior determines the behavior of the distributed file system.

To enable transparent access to data from a distributed file system, NFS (and other distributed file systems) employs a client-side file system that acts as a go-between the client applications and the server-side file system. The client-side file system translates the applications’ local-file-system-like requests to the appropriate calls to the server-side file system. Sun’s NFS defined an open communication protocol for the server and client file systems. In addition to Sun’s implementation of their client and server file systems (together, with the distributed file system), the open protocol enabled other vendors to implement their distributed file systems. The cross-vendor interoperability, thanks to the open protocol, was one of the reasons for NFS’s success and wide adoption.

**NFS Protocol**

The NFS protocol uses the Sun Remote Procedure Call (RPC) mechanism 1. For the same reasons that procedure calls simplify programs, RPC helps simplify the definition, organization, and implementation of remote services. The NFS protocol is defined in terms of a set of procedures, their arguments and results, and their effects. Remote procedure calls are synchronous, that is, the client application blocks until the server has completed the call and returned the results. This makes RPC very easy to use and understand because it behaves like a local procedure call. NFS uses a stateless protocol. The parameters to each procedure call contain all of the information necessary to complete the call, and the server does not keep track of any past requests. This makes crash recovery very easy; when a server crashes, the client resends NFS requests until a response is received, and the server does no crash recovery at all.

If the state is maintained on the server, on the other hand, recovery is much harder. Both client and server need to reliably detect crashes. The server needs to detect client crashes so that it can discard any state it is holding for the client, and the client must detect server crashes so that it can rebuild the server’s state.

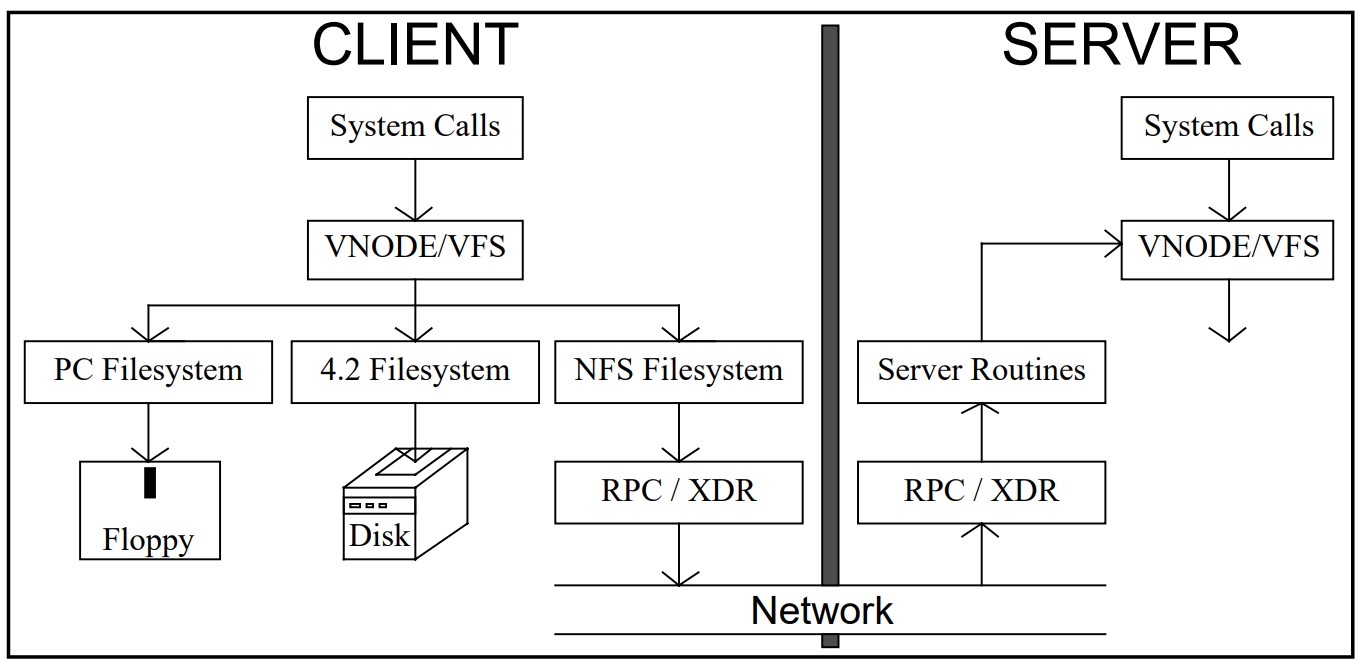
A stateless protocol avoids complex crash recovery. If a client just resends requests until a response is received, data will never be lost due to a server crash. In fact, the client cannot tell the difference between a server that has crashed and recovered, and a server that is slow.

# Server Side

Because the NFS server is stateless, when servicing an NFS request it must commit any modified data to stable storage before returning results. The implication for UNIX-based servers is that requests which modify the filesystem must flush all modified data to disk before returning from the call. For example, on a written request, not only the data block but also any modified indirect blocks and the block containing the inode must be flushed if they have been modified. Another modification to UNIX necessary for our server implementation is the addition of a generation number in the inode, and a filesystem id in the superblock. These extra numbers make it possible for the server to use the inode number, inode generation number, and filesystem id together as the fhandle for a file.

## Client Side

The Sun implementation of the client side provides an interface to NFS which is transparent to applications. To make transparent access to remote files work we had to use a method of locating remote - 4 - files that do not change the structure of path names. Some UNIX-based remote file access methods use pathnames like host: path or /../host/path to name remote files. This does not allow real transparent access since existing programs that parse path names have to be modified. Rather than doing a “late binding” of file address, we decided to do the hostname lookup and file address binding once per filesystem by allowing the client to attach a remote filesystem to a directory with the mount command. This method has the advantage that the client only has to deal with hostnames once, at mount time. It also allows the server to limit access to file systems by checking client credentials. The disadvantage is that remote files are not available to the client until a mount is done.



## Implementation

The first step in the implementation was modification of the 4.2 kernel to include the filesystem interface. Only a few of the filesystem routines in the kernel had to be completely rewritten to use vnodes. Namei, the routine that does pathname lookup, was changed to use the vnode lookup operation, and cleaned up so that it doesn’t use global state. The direnter routine, which adds new directory entries (used by create, rename, etc.), was fixed because it depended on the global state from namei.

Direnter was also modified to do directory locking during directory rename operations because inode locking is no longer available at this level, and vnodes are never locked. Beginning in March 1984, the user level RPC and XDR libraries were ported from the user-level library to the kernel, and we were able to make kernel to user and kernel to kernel RPC calls in June. We worked on RPC performance for about a month until the round trip time for a kernel to kernel null RPC call was 8.8 milliseconds on a Sun-2 (68010). The performance tuning included several speed ups to the UDP and IP code in the kernel On the client side, the mount command was modified to take additional arguments including a filesystem type and options string.

The filesystem type allows one mount command to mount any type of filesystem. The options string is used to pass optional flags to the different filesystem types at mount time.

For example, NFS allows two flavors of mount, soft and hard. A hard mounted filesystem will retry NFS requests forever if the server goes down, while a soft mount gives up after a while and returns an error. The problem with soft mounts is that most UNIX programs are not very good about checking return status from system calls so you can get some strange behavior when servers go down.

### Filesystem Naming

Servers export whole filesystems, but clients can mount any sub-directory of a remote filesystem on top of a local filesystem, or on top of another remote filesystem.

To alleviate some of the confusion we use a set of basic mounted filesystems on each machine and then let users add other filesystems on top of that. Remember that this is policy, there is no mechanism in NFS to enforce this. User home directories are mounted on /usr/servername. This may seem like a violation of our goals because hostnames are now part of pathnames but in fact the directories could have been called /usr/1, /usr/2, etc

To avoid the problems of loop detection and dynamic filesystem access checking, servers do not cross mount points on remote lookup requests. This means that in order to see the same filesystem layout as a server, a client has to remote mount each of the server’s exported filesystems.

### Statelessness

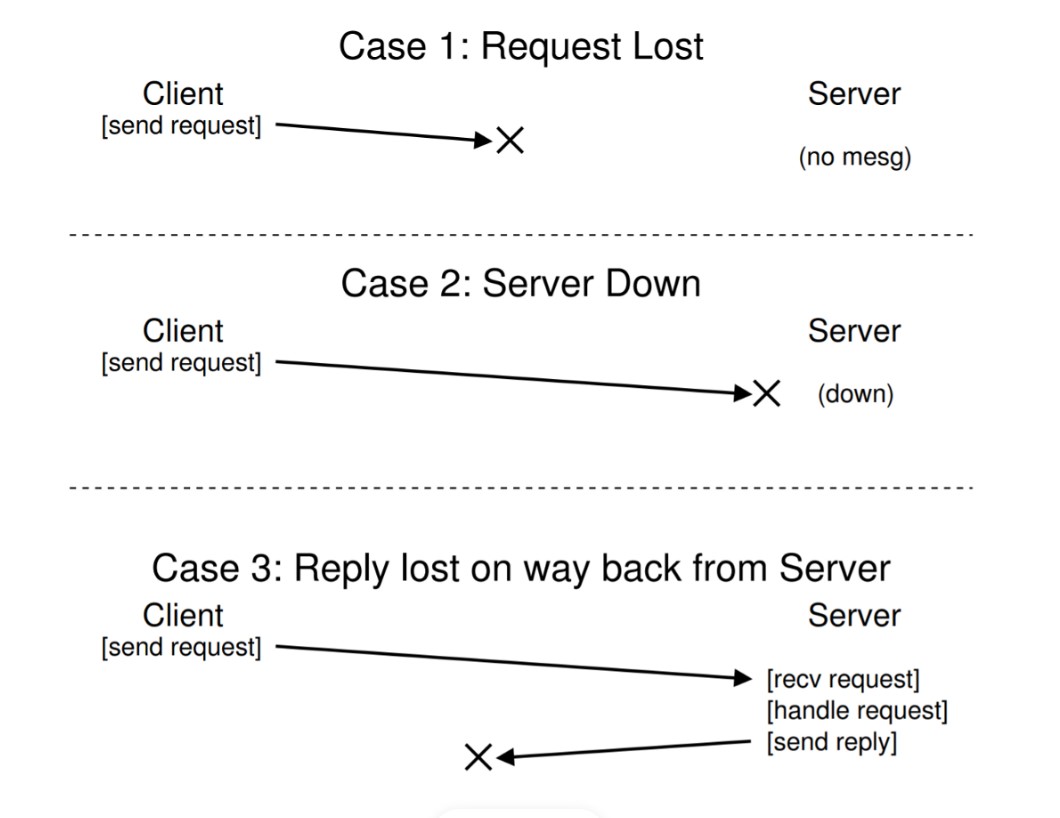
The key to a fast and simple crash recovery (for anything in general, but for the file server in this case) is statelessness. Statelessness, as the name suggests, refers to the property that the file server does not store any state. State refers to any information that needs to be stored durably, i.e., information that needs to be recovered after a failure. If the file server is stateless, it does not have to spend any time recovering any information (recall that ensuring data is recoverable requires crash-consistency, which is challenging) and can simply start processing requests from client-side file systems after recovering from a failure.

For a concrete example of a file system state, consider that a client application opens a file and reads 1KB of data from it. After processing the read data, the client reads the next 1KB of data from the file. The way this works in a local filesystem (like FFS or LFS) is that the application receives a file descriptor (basically an integer) when it opens the file. It uses the integer to make the first read request to the file system. The file system returns the read data and stores a file pointer (storing the offset into the file) at 1KB. Next, the application requests the next 1KB of data of the file using the file descriptor. The filesystem knows where to start reading the data from based on the file pointer. In this example, the file system stores (at least) two pieces of information as state — the mapping of the file description to the file, and the file pointer storing the offset within the file. This is because if the file system were to restart for some reason, it would need this information to be able to serve future requests from the application — the application would only specify the file descriptor and it would be up to the file system to recognize the file and the offset to read/write from/to the file.

In order to make the server-side file system stateless, NFS relies on the client-side file to maintain the required state. The client-side file system uses this state to extract the relevant information for any application request and passes it on to the server-side file system. In the case of the above example, the client-side file system stores the mapping for the file descriptor and the file pointer. When the application makes a read request, the client-side file system uses this state and generates a read request for the server-file system which includes the file and the offset to read from (in addition to the number of bytes to read which is already present in the application request).

### File handles

The next aspect of NFS design is how a file is identified. The client-side file system needs to maintain a mapping of the file descriptor to the file. In the case of a local file system, a file is identified by its inode number. NFS uses a file handle which is a tuple of three values — a volume number, an inode number, and a generation number. The volume number is used to identify the file system volume within which the file resides because one NFS file server can export multiple volumes (e.g., one file system volume per disk). The inode number identifies the file inode within the volume (similar to a local file system). The generation number is used to allow the reuse of inode numbers in the server-side file system. To understand the need for generation numbers consider the following example. A client obtains the file handler for a given file A. While the client holds the file handle for A, file A is deleted on the file server and the server-side file system uses its inode number for a new file B. If the file handle only consisted of the volume and inode number, a new request by the client would end up accessing data from the new file B. To avoid such situations, the server-side file system increments the generation number whenever it reuses an inode number. Note that this problem does not arise in the case of a local file system reusing an inode because there isn’t a separate client- and server-side file system for a local file system. NFS could have also avoided the use of a generation number if the client- and server-side file systems were tightly coupled (e.g., if the server-side file system knew which client-side file systems held which file systems). However, this would require the server-side file system to hold this state, making it stateful. Instead, NFS chooses to keep the server-side file system stateless by using generation numbers in the file handle.



**Idempotent operations**:

Another aspect of simplifying file servers crash-recovery is to have idempotent operations. An operation is called idempotent if it has the same end effect irrespective of the number of times it is performed. For example, reading data from a file is idempotent because whether you read it once or twice or 10 times, the result is the same. Writing data to a file at a fixed offset is also idempotent. However, appending data to a file is not idempotent — if you append ‘a’ to ‘b’ once, you get ‘ba’, but if you do it twice you get ‘baa’.

Idempotent operations offer a simple strategy for handling server failures — the client simply retries a request if it does not receive an acknowledgment of the successful completion of the request. The idempotency of operations ensures that this is safe. In addition to handling server failures (i.e., the server was not available to serve the request), the retry-if-not-acknowledged approach handles network failures as well (i.e., the request did not reach the server or the acknowledgment from the server did not reach the client because of a network packet loss). This simplicity offered by idempotent operations makes them a powerful tool for building distributed systems.

### Caching

Caching is used almost ubiquitously in systems (distributed or otherwise) to improve performance and NFS is no exception. NFS clients cache the file blocks they read so that they don’t have to request the same block again from the file server (and incur the network round trip latency) in case of the application requests it again. Further, when an application writes a block, the NFS client keeps it in its cache for some time before sending it to the file server. This is called write-back caching and it helps reduce the number of requests to the file system in case a block is updated repeatedly.

Although useful, caching often introduces the problem of cache-consistency. The cache-consistency problem is about identifying the most up-to-date data corresponding to a particular file or block when it can be write-back cached. For example in the NFS context, if a client (say client-A) updates a file and keeps it in its cache without updating the file server, a subsequent read from a different client (say client-B) gets a stale copy of the file because the most up-to-date copy of the file is not available with the file server. This is referred to as the update-visibility problem. As another example, consider a scenario in which a third client (client-C) had read and cached a copy of the file even before client-A modified it. Even if client-A writes back the updated file to the file server, client-C would continue to use outdated data because it will read the data from its cache. This is referred to as the stale-cache problem.

For the update-visibility problem, NFS chose the following design: whenever a file is closed on the client, it is flushed to the file server. This is referred to as the flush-on-close or close-to-open semantics. The flush-on-close semantic makes sense because closing a file is a reasonable logical demarcator. However, it can cause performance problems in the case of short-lived files. For example, consider the process of compiling a large binary. This typically involves the creation (and closing) of a bunch of temporary files that are discarded after a while. For such workloads, writing the temporary file to the file server is wasted work.

For the stale-cache problem, NFS chose the following design: before serving data from the cache, the client-side file system checks whether the file has been updated at the file server and serves the data only if it hasn’t been updated. However, doing this check for every read request on the client overwhelms the file server. To address this, NFS introduces (surprise surprise!) an attribute cache to store the last updated time for each file on the client. The client-side file system checks the attribute cache before contacting the file server. Although the attribute cache reduces the number of requests to the file server, it leads to rather unpredictable behavior — as each attribute cache entry has a time to live (TTL), applications could get old data rather arbitrarily depending on the TTL.

### Concluding thoughts

* NFS, as an example of an early distributed file system, showcases some key considerations for any distributed system. In particular, it showcases the power of idempotency and statelessness for simplifying crash recovery.
* It also highlights the cache-consistency problems in distributed file systems and the need to thoroughly reason about the solutions’ semantics and trade-offs.