Sun's Network File System (NFS)

One of the first uses of distributed client/server computing was in the realm of distributed file systems. In such an environment, there are a number of client machines and one server (or a few); the server stores the data on its disks, and clients request data through well-formed protocol messages. Figure 49.1 depicts the basic setup.
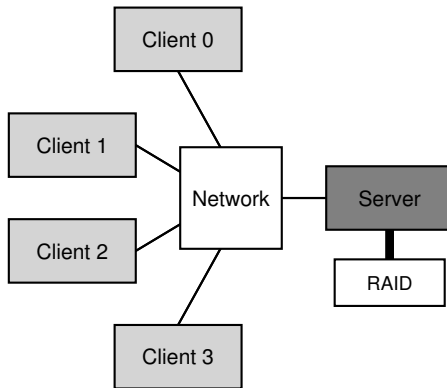

Figure 49.1: **A Generic Client/Server System**

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 **centralized administration**; for example, backing up files can be done from the few server machines instead of from the multitude of clients. Another advantage could be **security**; having all servers in a locked machine room prevents certain types of problems from arising.

CRUX: HOW TO BUILD A DISTRIBUTED FILE SYSTEM
How do you build a distributed file system? What are the key aspects to think about? What is easy to get wrong? What can we learn from existing systems?

49.1 A Basic Distributed File System

We now will study the architecture of a simplified distributed file system. A simple client/server distributed file system has more components than the file systems we have studied so far. On the client side, there are client applications which access files and directories through the **client-side file system**. A client application issues **system calls** to the client-side file system (such as open(), read(), write(), close(), mkdir(), etc.) in order to access files which are stored on the server. Thus, to client applications, the file system does not appear to be any different than a local (disk-based) file system, except perhaps for performance; in this way, distributed file systems provide **transparent** access to files, an obvious goal; after all, who would want to use a file system that required a different set of APIs or otherwise was a pain to use?

The role of the client-side file system is to execute the actions needed to service those system calls. For example, if the client issues a read() request, the client-side file system may send a message to the **server-side file system** (or, as it is commonly called, the **file server**) to read a particular block; the file server will then read the block from disk (or its own in-memory cache), and send a message back to the client with the requested data. The client-side file system will then copy the data into the user buffer supplied to the read() system call and thus the request will complete. Note that a subsequent read() of the same block on the client may be **cached** in client memory or on the client's disk even; in the best such case, no network traffic need be generated.
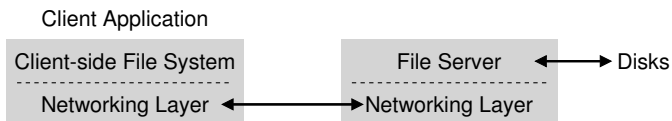

Figure 49.2: **Distributed File System Architecture**

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. Now it's time to study one particular system: Sun's Network File System (NFS).

ASIDE: WHY SERVERS CRASH

Before getting into the details of the NFSv2 protocol, you might be wondering: why do servers crash? Well, as you might guess, there are plenty of reasons. Servers may simply suffer from a **power outage** (temporarily); only when power is restored can the machines be restarted. Servers are often comprised of hundreds of thousands or even millions of lines of code; thus, they have **bugs** (even good software has a few bugs per hundred or thousand lines of code), and thus they eventually will trigger a bug that will cause them to crash. They also have memory leaks; even a small memory leak will cause a system to run out of memory and crash. And, finally, in distributed systems, there is a network between the client and the server; if the network acts strangely (for example, if it becomes **partitioned** and clients and servers are working but cannot communicate), it may appear as if a remote machine has crashed, but in reality it is just not currently reachable through the network.

49.2 On To NFS

One of the earliest and quite successful distributed systems was developed by Sun Microsystems, and is known as the Sun Network File System (or NFS) [S86]. In defining NFS, Sun took an unusual approach: instead of building a proprietary and closed system, Sun instead developed an **open protocol** which simply specified the exact message formats that clients and servers would use to communicate. Different groups could develop their own NFS servers and thus compete in an NFS marketplace while preserving interoperability. It worked: today there are many companies that sell NFS servers (including Oracle/Sun, NetApp [HLM94], EMC, IBM, and others), and the widespread success of NFS is likely attributed to this "open market" approach.

49.3 Focus: Simple And Fast Server Crash Recovery

In this chapter, we will discuss the classic NFS protocol (version 2, a.k.a. NFSv2), which was the standard for many years; small changes were made in moving to NFSv3, and larger-scale protocol changes were made in moving to NFSv4. However, NFSv2 is both wonderful and frustrating and thus serves as our focus.

In NFSv2, the main goal in the design of the protocol was *simple and fast server crash recovery*. In a multiple-client, single-server environment, this goal makes a great deal of sense; any minute that the server is down (or unavailable) makes *all* the client machines (and their users) unhappy and unproductive. Thus, as the server goes, so goes the entire system.

THREE
EASY
PIECES

49.4 Key To Fast Crash Recovery: Statelessness

This simple goal is realized in NFSv2 by designing what we refer to as a **stateless** protocol. The server, by design, does not keep track of anything about what is happening at each client. For example, the server does not know which clients are caching which blocks, or which files are currently open at each client, or the current file pointer position for a file, etc. Simply put, the server does not track anything about what clients are doing; rather, the protocol is designed to deliver in each protocol request *all the information* that is needed in order to complete the request. If it doesn't now, this stateless approach will make more sense as we discuss the protocol in more detail below.

For an example of a **stateful** (not stateless) protocol, consider the `open()` system call. Given a pathname, `open()` returns a file descriptor (an integer). This descriptor is used on subsequent `read()` or `write()` requests to access various file blocks, as in this application code (note that proper error checking of the system calls is omitted for space reasons):

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor "fd"
read(fd, buffer, MAX);   // read MAX from foo via "fd"
read(fd, buffer, MAX);   // read MAX again
...
read(fd, buffer, MAX);   // read MAX again
close(fd);               // close file
```

Figure 49.3: **Client Code: Reading From A File**

Now imagine that the client-side file system opens the file by sending a protocol message to the server saying "open the file 'foo' and give me back a descriptor". The file server then opens the file locally on its side and sends the descriptor back to the client. On subsequent reads, the client application uses that descriptor to call the `read()` system call; the client-side file system then passes the descriptor in a message to the file server, saying "read some bytes from the file that is referred to by the descriptor I am passing you here".

In this example, the file descriptor is a piece of **shared state** between the client and the server (Ousterhout calls this **distributed state** [O91]). Shared state, as we hinted above, complicates crash recovery. Imagine the server crashes after the first read completes, but before the client has issued the second one. After the server is up and running again, the client then issues the second read. Unfortunately, the server has no idea to which file `fd` is referring; that information was ephemeral (i.e., in memory) and thus lost when the server crashed. To handle this situation, the client and server would have to engage in some kind of **recovery protocol**, where the client would make sure to keep enough information around in its memory to be able to tell the server what it needs to know (in this case, that file descriptor `fd` refers to file `foo`).

It gets even worse when you consider the fact that a stateful server has to deal with client crashes. Imagine, for example, a client that opens a file and then crashes. The `open()` uses up a file descriptor on the server; how can the server know it is OK to close a given file? In normal operation, a client would eventually call `close()` and thus inform the server that the file should be closed. However, when a client crashes, the server never receives a `close()`, and thus has to notice the client has crashed in order to close the file.

For these reasons, the designers of NFS decided to pursue a stateless approach: each client operation contains all the information needed to complete the request. No fancy crash recovery is needed; the server just starts running again, and a client, at worst, might have to retry a request.

49.5 The NFSv2 Protocol

We thus arrive at the NFSv2 protocol definition. Our problem statement is simple:

THE CRUX: HOW TO DEFINE A STATELESS FILE PROTOCOL
How can we define the network protocol to enable stateless operation? Clearly, stateful calls like `open()` can't be a part of the discussion (as it would require the server to track open files); however, the client application will want to call `open()`, `read()`, `write()`, `close()` and other standard API calls to access files and directories. Thus, as a refined question, how do we define the protocol to both be stateless *and* support the POSIX file system API?

One key to understanding the design of the NFS protocol is understanding the **file handle**. File handles are used to uniquely describe the file or directory a particular operation is going to operate upon; thus, many of the protocol requests include a file handle.

You can think of a file handle as having three important components: a *volume identifier*, an *inode number*, and a *generation number*; together, these three items comprise a unique identifier for a file or directory that a client wishes to access. The volume identifier informs the server which file system the request refers to (an NFS server can export more than one file system); the inode number tells the server which file within that partition the request is accessing. Finally, the generation number is needed when reusing an inode number; by incrementing it whenever an inode number is reused, the server ensures that a client with an old file handle can't accidentally access the newly-allocated file.

Here is a summary of some of the important pieces of the protocol; the full protocol is available elsewhere (see Callaghan's book for an excellent and detailed overview of NFS [C00]).

NFSPROC_GETATTR	file handle
	returns: attributes
NFSPROC_SETATTR	file handle, attributes
	returns: attributes
NFSPROC_LOOKUP	directory file handle, name of file/dir to look up
	returns: file handle, attributes
NFSPROC_READ	file handle, offset, count
	data, attributes
NFSPROC_WRITE	file handle, offset, count, data
	attributes
NFSPROC_CREATE	directory file handle, name of file, attributes
	file handle, attributes
NFSPROC_REMOVE	directory file handle, name of file to be removed
	–
NFSPROC_MKDIR	directory file handle, name of directory, attributes
	file handle, attributes
NFSPROC_RMDIR	directory file handle, name of directory to be removed
	–
NFSPROC_READDIR	directory handle, count of bytes to read, cookie
	returns: directory entries, cookie (to get more entries)

Figure 49.4: **The NFS Protocol: Examples**

We briefly highlight the important components of the protocol. First, the LOOKUP protocol message is used to obtain a file handle, which is then subsequently used to access file data. The client passes a directory file handle and name of a file to look up, and the handle to that file (or directory) plus its attributes are passed back to the client from the server.

For example, assume the client already has a directory file handle for the root directory of a file system (/) (indeed, this would be obtained through the NFS **mount protocol**, which is how clients and servers first are connected together; we do not discuss the mount protocol here for sake of brevity). If an application running on the client opens the file /foo.txt, the client-side file system sends a lookup request to the server, passing it the root file handle and the name foo.txt; if successful, the file handle (and attributes) for foo.txt will be returned.

In case you are wondering, attributes are just the metadata that the file system tracks about each file, including fields such as file creation time, last modification time, size, ownership and permissions information, and so forth, i.e., the same type of information that you would get back if you called stat() on a file.

Once a file handle is available, the client can issue READ and WRITE protocol messages on a file to read or write the file, respectively. The READ protocol message requires the protocol to pass along the file handle of the file along with the offset within the file and number of bytes to read. The server then will be able to issue the read (after all, the handle tells the server which volume and which inode to read from, and the offset and count tells it which bytes of the file to read) and return the data (and up-

to-date attributes) to the client (or an error if there was a failure). WRITE is handled similarly, except the data is passed from the client to the server, and just a success code (and up-to-date attributes) is returned.

One last interesting protocol message is the GETATTR request; given a file handle, it simply fetches the attributes for that file, including the last modified time of the file. We will see why this protocol request is important in NFSv2 below when we discuss caching (can you guess why?).

49.6 From Protocol To Distributed File System

Hopefully you are now getting some sense of how this protocol is turned into a file system across the client-side file system and the file server. The client-side file system tracks open files, and generally translates application requests into the relevant set of protocol messages. The server simply responds to protocol messages, each of which contains all of the information needed to complete the request.

For example, let us consider a simple application which reads a file. In the diagram (Figure 49.5), we show what system calls the application makes, and what the client-side file system and file server do in responding to such calls.

A few comments about the figure. First, notice how the client tracks all relevant **state** for the file access, including the mapping of the integer file descriptor to an NFS file handle as well as the current file pointer. This enables the client to turn each read request (which you may have noticed do *not* specify the offset to read from explicitly) into a properly-formatted read protocol message which tells the server exactly which bytes from the file to read. Upon a successful read, the client updates the current file position; subsequent reads are issued with the same file handle but a different offset.

Second, you may notice where server interactions occur. When the file is opened for the first time, the client-side file system sends a LOOKUP request message. Indeed, if a long pathname must be traversed (e.g., `/home/remzi/foo.txt`), the client would send three LOOKUPs: one to look up `home` in the directory `/`, one to look up `remzi` in `home`, and finally one to look up `foo.txt` in `remzi`.

Third, you may notice how each server request has all the information needed to complete the request in its entirety. This design point is critical to be able to gracefully recover from server failure, as we will now discuss in more detail; it ensures that the server does not need state to be able to respond to the request.

Client	Server
fd = open("/foo", ...);	
Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request
	look for "foo" in root dir
	return foo's FH + attributes
Receive LOOKUP reply	
allocate file desc in open file table	
store foo's FH in table	
store current file position (0)	
return file descriptor to application	

Client	Server
read(fd, buffer, MAX);	
Index into open file table with fd	
get NFS file handle (FH)	
use current file position as offset	
Send READ (FH, offset=0, count=MAX)	
	Receive READ request
	use FH to get volume/inode num
	read inode from disk (or cache)
	compute block location (using offset)
	read data from disk (or cache)
	return data to client
Receive READ reply	
update file position (+bytes read)	
set current file position = MAX	
return data/error code to app	

Client	Server
read(fd, buffer, MAX);	
Same except offset=MAX and set current file position = 2*MAX	

Client	Server
read(fd, buffer, MAX);	
Same except offset=2*MAX and set current file position = 3*MAX	

Client	Server
close(fd);	
Just need to clean up local structures	
Free descriptor "fd" in open file table	
(No need to talk to server)	

Figure 49.5: **Reading A File: Client-side And File Server Actions**

TIP: IDEMPOTENCY IS POWERFUL

Idempotency is a useful property when building reliable systems. When an operation can be issued more than once, it is much easier to handle failure of the operation; you can just retry it. If an operation is *not* idempotent, life becomes more difficult.

49.7 Handling Server Failure With Idempotent Operations

When a client sends a message to the server, it sometimes does not receive a reply. There are many possible reasons for this failure to respond. In some cases, the message may be dropped by the network; networks do lose messages, and thus either the request or the reply could be lost and thus the client would never receive a response.

It is also possible that the server has crashed, and thus is not currently responding to messages. After a bit, the server will be rebooted and start running again, but in the meanwhile all requests have been lost. In all of these cases, clients are left with a question: what should they do when the server does not reply in a timely manner?

In NFSv2, a client handles all of these failures in a single, uniform, and elegant way: it simply *retries* the request. Specifically, after sending the request, the client sets a timer to go off after a specified time period. If a reply is received before the timer goes off, the timer is canceled and all is well. If, however, the timer goes off *before* any reply is received, the client assumes the request has not been processed and resends it. If the server replies, all is well and the client has neatly handled the problem.

The ability of the client to simply retry the request (regardless of what caused the failure) is due to an important property of most NFS requests: they are **idempotent**. An operation is called idempotent when the effect of performing the operation multiple times is equivalent to the effect of performing the operation a single time. For example, if you store a value to a memory location three times, it is the same as doing so once; thus "store value to memory" is an idempotent operation. If, however, you increment a counter three times, it results in a different amount than doing so just once; thus, "increment counter" is not idempotent. More generally, any operation that just reads data is obviously idempotent; an operation that updates data must be more carefully considered to determine if it has this property.

The heart of the design of crash recovery in NFS is the idempotency of most common operations. LOOKUP and READ requests are trivially idempotent, as they only read information from the file server and do not update it. More interestingly, WRITE requests are also idempotent. If, for example, a WRITE fails, the client can simply retry it. The WRITE message contains the data, the count, and (importantly) the exact offset to write the data to. Thus, it can be repeated with the knowledge that the outcome of multiple writes is the same as the outcome of a single one.

Case 1: Request Lost

Client   Server
[send request] ────────────────➤✕

                                                          (no mesg)

- -

Case 2: Server Down

Client   Server
[send request] ──────────────────────➤✕   (down)

- -

Case 3: Reply lost on way back from Server

Client   Server
[send request] ──────────────────────➤ [recv request]
                                                          [handle request]
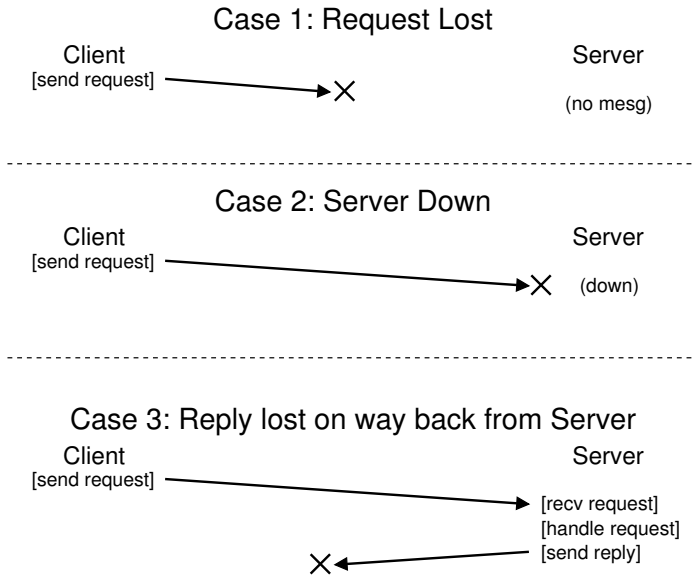   ✕◄─────────────── [send reply]

Figure 49.6: **The Three Types Of Loss**

In this way, the client can handle all timeouts in a unified way. If a
WRITE request was simply lost (Case 1 above), the client will retry it, the
server will perform the write, and all will be well. The same will happen
if the server happened to be down while the request was sent, but back
up and running when the second request is sent, and again all works
as desired (Case 2). Finally, the server may in fact receive the WRITE
request, issue the write to its disk, and send a reply. This reply may get
lost (Case 3), again causing the client to re-send the request. When the
server receives the request again, it will simply do the exact same thing:
write the data to disk and reply that it has done so. If the client this time
receives the reply, all is again well, and thus the client has handled both
message loss and server failure in a uniform manner. Neat!

A small aside: some operations are hard to make idempotent. For
example, when you try to make a directory that already exists, you are
informed that the mkdir request has failed. Thus, in NFS, if the file server
receives a MKDIR protocol message and executes it successfully but the
reply is lost, the client may repeat it and encounter that failure when in
fact the operation at first succeeded and then only failed on the retry.
Thus, life is not perfect.

TIP: PERFECT IS THE ENEMY OF THE GOOD (VOLTAIRE'S LAW)
Even when you design a beautiful system, sometimes all the corner cases
don't work out exactly as you might like. Take the mkdir example above;
one could redesign mkdir to have different semantics, thus making it
idempotent (think about how you might do so); however, why bother?
The NFS design philosophy covers most of the important cases, and over-
all makes the system design clean and simple with regards to failure.
Thus, accepting that life isn't perfect and still building the system is a sign
of good engineering. Apparently, this wisdom is attributed to Voltaire,
for saying "... a wise Italian says that the best is the enemy of the good"
[V72], and thus we call it **Voltaire's Law**.

49.8 Improving Performance: Client-side Caching

Distributed file systems are good for a number of reasons, but sending
all read and write requests across the network can lead to a big perfor-
mance problem: the network generally isn't that fast, especially as com-
pared to local memory or disk. Thus, another problem: how can we im-
prove the performance of a distributed file system?

The answer, as you might guess from reading the big bold words in
the sub-heading above, is client-side **caching**. The NFS client-side file
system caches file data (and metadata) that it has read from the server in
client memory. Thus, while the first access is expensive (i.e., it requires
network communication), subsequent accesses are serviced quite quickly
out of client memory.

The cache also serves as a temporary buffer for writes. When a client
application first writes to a file, the client buffers the data in client mem-
ory (in the same cache as the data it read from the file server) before writ-
ing the data out to the server. Such **write buffering** is useful because it de-
couples application `write()` latency from actual write performance, i.e.,
the application's call to `write()` succeeds immediately (and just puts
the data in the client-side file system's cache); only later does the data get
written out to the file server.

Thus, NFS clients cache data and performance is usually great and
we are done, right? Unfortunately, not quite. Adding caching into any
sort of system with multiple client caches introduces a big and interesting
challenge which we will refer to as the **cache consistency problem**.

49.9 The Cache Consistency Problem

The cache consistency problem is best illustrated with three clients and
a single server. Imagine client C1 reads a file F, and keeps a copy of the
file in its local cache. Now imagine a different client, C2, overwrites the
file F, thus changing its contents; let's call the new version of the file F
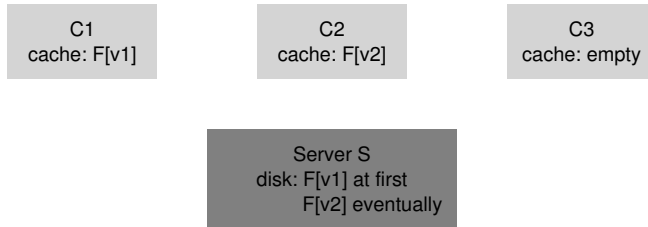
C1	C2	C3
cache: F[v1]	cache: F[v2]	cache: empty

Server S
disk: F[v1] at first
F[v2] eventually

Figure 49.7: **The Cache Consistency Problem**

(version 2), or F[v2] and the old version F[v1] so we can keep the two distinct (but of course the file has the same name, just different contents). Finally, there is a third client, C3, which has not yet accessed the file F.

You can probably see the problem that is upcoming (Figure 49.7). In fact, there are two subproblems. The first subproblem is that the client C2 may buffer its writes in its cache for a time before propagating them to the server; in this case, while F[v2] sits in C2's memory, any access of F from another client (say C3) will fetch the old version of the file (F[v1]). Thus, by buffering writes at the client, other clients may get stale versions of the file, which may be undesirable; indeed, imagine the case where you log into machine C2, update F, and then log into C3 and try to read the file, only to get the old copy! Certainly this could be frustrating. Thus, let us call this aspect of the cache consistency problem **update visibility**; when do updates from one client become visible at other clients?

The second subproblem of cache consistency is a **stale cache**; in this case, C2 has finally flushed its writes to the file server, and thus the server has the latest version (F[v2]). However, C1 still has F[v1] in its cache; if a program running on C1 reads file F, it will get a stale version (F[v1]) and not the most recent copy (F[v2]), which is (often) undesirable.

NFSv2 implementations solve these cache consistency problems in two ways. First, to address update visibility, clients implement what is sometimes called **flush-on-close** (a.k.a., **close-to-open**) consistency semantics; specifically, when a file is written to and subsequently closed by a client application, the client flushes all updates (i.e., dirty pages in the cache) to the server. With flush-on-close consistency, NFS ensures that a subsequent open from another node will see the latest file version.

Second, to address the stale-cache problem, NFSv2 clients first check to see whether a file has changed before using its cached contents. Specifically, before using a cached block, the client-side file system will issue a GETATTR request to the server to fetch the file's attributes. The attributes, importantly, include information as to when the file was last modified on the server; if the time-of-modification is more recent than the time that the file was fetched into the client cache, the client **invalidates** the file, thus removing it from the client cache and ensuring that subsequent reads will

go to the server and retrieve the latest version of the file. If, on the other hand, the client sees that it has the latest version of the file, it will go ahead and use the cached contents, thus increasing performance.

When the original team at Sun implemented this solution to the stale-cache problem, they realized a new problem; suddenly, the NFS server was flooded with GETATTR requests. A good engineering principle to follow is to design for the **common case**, and to make it work well; here, although the common case was that a file was accessed only from a single client (perhaps repeatedly), the client always had to send GETATTR requests to the server to make sure no one else had changed the file. A client thus bombards the server, constantly asking "has anyone changed this file?", when most of the time no one had.

To remedy this situation (somewhat), an **attribute cache** was added to each client. A client would still validate a file before accessing it, but most often would just look in the attribute cache to fetch the attributes. The attributes for a particular file were placed in the cache when the file was first accessed, and then would timeout after a certain amount of time (say 3 seconds). Thus, during those three seconds, all file accesses would determine that it was OK to use the cached file and thus do so with no network communication with the server.

49.10 Assessing NFS Cache Consistency

A few final words about NFS cache consistency. The flush-on-close behavior was added to "make sense", but introduced a certain performance problem. Specifically, if a temporary or short-lived file was created on a client and then soon deleted, it would still be forced to the server. A more ideal implementation might keep such short-lived files in memory until they are deleted and thus remove the server interaction entirely, perhaps increasing performance.

More importantly, the addition of an attribute cache into NFS made it very hard to understand or reason about exactly what version of a file one was getting. Sometimes you would get the latest version; sometimes you would get an old version simply because your attribute cache hadn't yet timed out and thus the client was happy to give you what was in client memory. Although this was fine most of the time, it would (and still does!) occasionally lead to odd behavior.

And thus we have described the oddity that is NFS client caching. It serves as an interesting example where details of an implementation serve to define user-observable semantics, instead of the other way around.

49.11 Implications On Server-Side Write Buffering

Our focus so far has been on client caching, and that is where most of the interesting issues arise. However, NFS servers tend to be well-equipped machines with a lot of memory too, and thus they have caching

concerns as well. When data (and metadata) is read from disk, NFS
servers will keep it in memory, and subsequent reads of said data (and
metadata) will not go to disk, a potential (small) boost in performance.

More intriguing is the case of write buffering. An NFS server abso-
lutely may *not* return success on a WRITE protocol request until the write
has been forced to stable storage (e.g., to disk or some other persistent
device). While the server can place a copy of the data in its memory, re-
turning success to the client on a WRITE protocol request could result in
incorrect behavior; can you figure out why?

The answer lies in our assumptions about how clients handle server
failure. Imagine the following sequence of writes as issued by a client:

```
write(fd, a_buffer, size); // fill 1st block with a's
write(fd, b_buffer, size); // fill 2nd block with b's
write(fd, c_buffer, size); // fill 3rd block with c's
```

These writes overwrite the three blocks of a file with a block of a's,
then b's, and then c's. Thus, if the file initially looked like this:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz
```

We might expect the final result after these writes to be like this, with the
x's, y's, and z's, would be overwritten with a's, b's, and c's, respectively.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
cccccccccccccccccccccccccccccccccccccccccccccc
```

Now let's assume for the sake of the example that these three client
writes were issued to the server as three distinct WRITE protocol mes-
sages. Assume the first WRITE message is received by the server and
issued to the disk, and the client informed of its success. Now assume
the second write is just buffered in memory, and the server also reports
it success to the client *before* forcing it to disk; unfortunately, the server
crashes before writing it to disk. The server quickly restarts and receives
the third write request, which also succeeds.

Thus, to the client, all the requests succeeded, but we are surprised
that the file contents look like this:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <--- oops
cccccccccccccccccccccccccccccccccccccccccccccc
```

Yikes! Because the server told the client that the second write was
successful before committing it to disk, an old chunk is left in the file,
which, depending on the application, might be catastrophic.

ASIDE: INNOVATION BREEDS INNOVATION

As with many pioneering technologies, bringing NFS into the world also required other fundamental innovations to enable its success. Probably the most lasting is the **Virtual File System** (**VFS**) / **Virtual Node** (**vnode**) interface, introduced by Sun to allow different file systems to be readily plugged into the operating system [K86].

The VFS layer includes operations that are done to an entire file system, such as mounting and unmounting, getting file-system wide statistics, and forcing all dirty (not yet written) writes to disk. The vnode layer consists of all operations one can perform on a file, such as open, close, reads, writes, and so forth.

To build a new file system, one simply has to define these "methods"; the framework then handles the rest, connecting system calls to the particular file system implementation, performing generic functions common to all file systems (e.g., caching) in a centralized manner, and thus providing a way for multiple file system implementations to operate simultaneously within the same system.

Although some of the details have changed, many modern systems have some form of a VFS/vnode layer, including Linux, BSD variants, macOS, and even Windows (in the form of the Installable File System). Even if NFS becomes less relevant to the world, some of the necessary foundations beneath it will live on.

To avoid this problem, NFS servers *must* commit each write to stable (persistent) storage before informing the client of success; doing so enables the client to detect server failure during a write, and thus retry until it finally succeeds. Doing so ensures we will never end up with file contents intermingled as in the above example.

The problem that this requirement gives rise to in NFS server implementation is that write performance, without great care, can be *the* major performance bottleneck. Indeed, some companies (e.g., Network Appliance) came into existence with the simple objective of building an NFS server that can perform writes quickly; one trick they use is to first put writes in a battery-backed memory, thus enabling to quickly reply to WRITE requests without fear of losing the data and without the cost of having to write to disk right away; the second trick is to use a file system design specifically designed to write to disk quickly when one finally needs to do so [HLM94, RO91].

49.12 Summary

We have seen the introduction of the NFS distributed file system. NFS is centered around the idea of simple and fast recovery in the face of server failure, and achieves this end through careful protocol design. Idem-

---

ASIDE: KEY NFS TERMS

- The key to realizing the main goal of fast and simple crash recovery in NFS is in the design of a **stateless** protocol. After a crash, the server can quickly restart and begin serving requests again; clients just **retry** requests until they succeed.

- Making requests **idempotent** is a central aspect of the NFS protocol. An operation is idempotent when the effect of performing it multiple times is equivalent to performing it once. In NFS, idempotency enables client retry without worry, and unifies client lost-message retransmission and how the client handles server crashes.

- Performance concerns dictate the need for client-side **caching** and **write buffering**, but introduces a **cache consistency problem**.

- NFS implementations provide an engineering solution to cache consistency through multiple means: a **flush-on-close** (**close-to-open**) approach ensures that when a file is closed, its contents are forced to the server, enabling other clients to observe the updates to it. An attribute cache reduces the frequency of checking with the server whether a file has changed (via GETATTR requests).

- NFS servers must commit writes to persistent media before returning success; otherwise, data loss can arise.

- To support NFS integration into the operating system, Sun introduced the **VFS/Vnode** interface, enabling multiple file system implementations to coexist in the same operating system.

---

potency of operations is essential; because a client can safely replay a failed operation, it is OK to do so whether or not the server has executed the request.

We also have seen how the introduction of caching into a multiple-client, single-server system can complicate things. In particular, the system must resolve the cache consistency problem in order to behave reasonably; however, NFS does so in a slightly ad hoc fashion which can occasionally result in observably weird behavior. Finally, we saw how server caching can be tricky: writes to the server must be forced to stable storage before returning success (otherwise data can be lost).

We haven't talked about other issues which are certainly relevant, notably security. Security in early NFS implementations was remarkably lax; it was rather easy for any user on a client to masquerade as other users and thus gain access to virtually any file. Subsequent integration with more serious authentication services (e.g., Kerberos [NT94]) have addressed these obvious deficiencies.

References

[AKW88] "The AWK Programming Language" by Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger. Pearson, 1988 (1st edition). *A concise, wonderful book about awk. We once had the pleasure of meeting Peter Weinberger; when he introduced himself, he said "I'm Peter Weinberger, you know, the 'W' in awk?" As huge awk fans, this was a moment to savor. One of us (Remzi) then said, "I love awk! I particularly love the book, which makes everything so wonderfully clear." Weinberger replied (crestfallen), "Oh, Kernighan wrote the book."*

[C00] "NFS Illustrated" by Brent Callaghan. Addison-Wesley Professional Computing Series, 2000. *A great NFS reference; incredibly thorough and detailed per the protocol itself.*

[ES03] "New NFS Tracing Tools and Techniques for System Analysis" by Daniel Ellard and Margo Seltzer. LISA '03, San Diego, California. *An intricate, careful analysis of NFS done via passive tracing. By simply monitoring network traffic, the authors show how to derive a vast amount of file system understanding.*

[HLM94] "File System Design for an NFS File Server Appliance" by Dave Hitz, James Lau, Michael Malcolm. USENIX Winter 1994. San Francisco, California, 1994. *Hitz et al. were greatly influenced by previous work on log-structured file systems.*

[K86] "Vnodes: An Architecture for Multiple File System Types in Sun UNIX" by Steve R. Kleiman. USENIX Summer '86, Atlanta, Georgia. *This paper shows how to build a flexible file system architecture into an operating system, enabling multiple different file system implementations to coexist. Now used in virtually every modern operating system in some form.*

[NT94] "Kerberos: An Authentication Service for Computer Networks" by B. Clifford Neuman, Theodore Ts'o. IEEE Communications, 32(9):33-38, September 1994. *Kerberos is an early and hugely influential authentication service. We probably should write a book chapter about it sometime...*

[O91] "The Role of Distributed State" by John K. Ousterhout. 1991. Available at this site: `ftp://ftp.cs.berkeley.edu/ucb/sprite/papers/state.ps`. *A rarely referenced discussion of distributed state; a broader perspective on the problems and challenges.*

[P+94] "NFS Version 3: Design and Implementation" by Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, Dave Hitz. USENIX Summer 1994, pages 137-152. *The small modifications that underlie NFS version 3.*

[P+00] "The NFS version 4 protocol" by Brian Pawlowski, David Noveck, David Robinson, Robert Thurlow. 2nd International System Administration and Networking Conference (SANE 2000). *Undoubtedly the most literary paper on NFS ever written.*

[RO91] "The Design and Implementation of the Log-structured File System" by Mendel Rosenblum, John Ousterhout. Symposium on Operating Systems Principles (SOSP), 1991. *LFS again. No, you can never get enough LFS.*

[S86] "The Sun Network File System: Design, Implementation and Experience" by Russel Sandberg. USENIX Summer 1986. *The original NFS paper; though a bit of a challenging read, it is worthwhile to see the source of these wonderful ideas.*

[Sun89] "NFS: Network File System Protocol Specification" by Sun Microsystems, Inc. Request for Comments: 1094, March 1989. Available: `http://www.ietf.org/rfc/rfc1094.txt`. *The dreaded specification; read it if you must, i.e., you are getting paid to read it. Hopefully, paid a lot. Cash money!*

[V72] "La Begueule" by Francois-Marie Arouet a.k.a. Voltaire. Published in 1772. *Voltaire said a number of clever things, this being but one example. For example, Voltaire also said "If you have two religions in your land, the two will cut each other's throats; but if you have thirty religions, they will dwell in peace." What do you say to that, Democrats and Republicans?*

Homework (Measurement)

In this homework, you'll do a little bit of NFS trace analysis using real traces. The source of these traces is Ellard and Seltzer's effort [ES03]. Make sure to read the related README and download the relevant tarball from the OSTEP homework page (as usual) before starting.

Questions

1. A first question for your trace analysis: using the timestamps found in the first column, determine the period of time the traces were taken from. How long is the period? What day/week/month/year was it? (does this match the hint given in the file name?) Hint: Use the tools `head -1` and `tail -1` to extract the first and last lines of the file, and do the calculation.

2. Now, let's do some operation counts. How many of each type of operation occur in the trace? Sort these by frequency; which operation is most frequent? Does NFS live up to its reputation?

3. Now let's look at some particular operations in more detail. For example, the GETATTR request returns a lot of information about files, including which user ID the request is being performed for, the size of the file, and so forth. Make a distribution of file sizes accessed within the trace; what is the average file size? Also, how many different users access files in the trace? Do a few users dominate traffic, or is it more spread out? What other interesting information is found within GETATTR replies?

4. You can also look at requests to a given file and determine how files are being accessed. For example, is a given file being read or written sequentially? Or randomly? Look at the details of READ and WRITE requests/replies to compute the answer.

5. Traffic comes from many machines and goes to one server (in this trace). Compute a traffic matrix, which shows how many different clients there are in the trace, and how many requests/replies go to each. Do a few machines dominate, or is it more evenly balanced?

6. The timing information, and the per-request/reply unique ID, should allow you to compute the latency for a given request. Compute the latencies of all request/reply pairs, and plot them as a distribution. What is the average? Maximum? Minimum?

7. Sometimes requests are retried, as the request or its reply could be lost or dropped. Can you find any evidence of such retrying in the trace sample?

8. There are many other questions you could answer through more analysis. What questions do you think are important? Suggest them to us, and perhaps we'll add them here!