Introduction

Distributed file systems were one of the first uses of distributed client/server computing. The server stores data on its disks, while clients request data via well-formed protocol messages. 49.1 shows the basic setup.

For example, to access their directories and files, clients transmit messages over a network to the server. Why do we bother? Can't we just let clients use their own disks? Basically, this configuration facilitates client-to-client data transfer. It is possible to browse the file system in the same way on two different machines (Client 0 and Client 2). Your data is automatically transferred between machines. Secondary benefits include centralized administration, such as backing up files from a few server machines rather than many clients. Another benefit is security; keeping all servers in a locked machine room prevents certain issues.

DISTRIBUTED FILE SYSTEM DESIGN

How do you build a distributed file system?
What are the main considerations?
What is easy to get wrong?
What can existing systems teach us?

Basic Distributed File System

Now we'll look at a simplified distributed file system. It contains more components than the other file systems we've researched. Client applications use the client-side file system to access files and directories. To access files stored on the server, a client program uses the client-side file system (open(), read(), write(), close(), mkdir(), etc.). In this approach, distributed file systems enable transparent access to files, which is an apparent goal; after all, who would want to use a file system that required a new set of APIs or was otherwise difficult to use?

The client-side file system's role is to service system calls. For example, a read() request from the client causes the client-side file system to send a message to the server-side file system (or file server) to read a certain block from disk (or its own in-memory cache). It will then copy data into the user buffer supplied by read(), completing your request. Client read() of the same block may be cached in memory or disk; in this scenario, no network traffic is generated.

From this brief introduction, it should be clear that a client/server distributed file system has two key components: the client-side file system and the file server. Their joint behavior determines the distributed file system's functioning. Now we'll focus on one system: Sun's Network File System (NFS).

On To NFS

Sun Microsystems created one of the first and most successful distributed systems, the Sun Network File System (or NFS). Instead of creating a proprietary and closed system, Sun created an open protocol that merely defined the message formats that clients and servers would use to interact. Different parties could design their own NFS servers, preserving interoperability. It worked: today numerous companies sell NFS servers (Oracle/Sun, NetApp, EMC, IBM, and others), a testament to NFS's widespread success.

Focus: Simple and Fast Server Crash Recovery

This chapter will cover the classic NFS protocol (version 2, sometimes known as NFSv2), which was the standard for many years. But NFSv2 is fantastic and frustrating, so that's our emphasis.

The protocol design goal for NFSv2 was easy and rapid server recovery. Any minute the server is down (or unavailable) makes all the client machines (and their users) angry and unproductive. So, if the server fails, the system fails.

Statelessness

NFSv2 achieves this goal by designing a stateless protocol. The server does not keep track of what happens at each client by design. The server, for example, has no idea which clients are caching which blocks, which files are open at each client, or where a file's reference is located. Simply said, the server does not track what clients do; rather, the protocol is designed to convey all necessary information in each protocol request. If it doesn't now, it will when we cover the protocol in greater depth below.

Consider the open() system call as an example of a stateful protocol. open() returns a file descriptor (an inte- ger). Notice that proper error checking of the system calls is removed for space reasons in this application code (note that proper error checking of the system calls is omitted):

Pretend that the client-side file system opens the file by sending a protocol message to the server saying, "open the file 'foo' and return a descriptor." This opens the file locally and sends the descriptor to the client. After that, the client application uses the descriptor to use read(), and the client-side file system sends the descriptor along with the request to the file server.

The file descriptor is a shared state between the client and server (Ousterhout calls this distributed state). Shared state, as stated above, impedes recovery. Assume the server crashes after the first read but before the client sends the second. After the server restarts, the client issues the second read. Since such information was ephemeral (i.e., in memory), it was lost when the server crashed. To tackle this situation, the client and server would need to establish a recovery protocol, where the client would maintain enough information in its memory to inform the server of the situation (in this case, that file descriptor fd refers to file foo).

Even worse, a stateful server must cope with client crashes. Assume a client opens a file and then crashes. A file descriptor is used by open(); how does the server know when to terminate a file? Normally, a client would call close() to tell the server to close the file. However, when a client crashes, the server never receives a close(), thus it must be notified to close the file.

In order to address these issues, NFS designers chose a stateless approach: each client operation contains all necessary information. The server just restarts, and a client may have to retry a request.

The NFSv2 Protocol

So here's the NFSv2 protocol. Our issue is clear:

STATELESS FILE PROTOCOL DEFINITION

How can we design a stateless network protocol? But the client application will want to use open(), read(), write(), close(), and other normal API functions to access files and directories (as the server would need to track open files, for example). So, how can we define the protocol to be stateless while also supporting the POSIX file system API.

The file handle is important in understanding the NFS protocol design. Many protocol requests include a file handle to uniquely identify the file or directory being operated on.
One might consider of a file handle as having three key components: volume identification, inode number, and the client's generation number. Inode numbers notify the server which file system the request is for (an NFS server can export many file systems). A client with an old file handle can't unintentionally access the newly-allocated file since the generation number is incremented every time an inode number is reused, therefore the server increments it every time.

The whole protocol (with a good and extensive description of NFS) is available elsewhere.

In order to access files, the LOOKUP protocol message first obtains a file handle. The client sends the server a directory file handle and a file name to lookup, and the server sends back the handle and associated attributes.

Assume the client already has a handle to the file system's root directory (/). (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). To find the file handle (and characteristics) for the file foo.txt, a client-side program opens the file /foo.txt.

The file system tracks fields such file creation time, last modification time, size, ownership and permissions information, etc., in attributes.

Following this, the client can use the READ and WRITE protocol messages to read and write files. The READ protocol message requires the protocol to send the file handle, offset within the file, and the quantity of bytes to read, along with the file name. The server may then issue the read (because the handle informs it which volume and inode to read from, and offset and

count tell it which bytes of the file to read) and provide the data to the client (or an error if there was a failure). Writing data to the server is done similarly, except that no data is sent back.

The GETATTR request simply collects the file's attributes, including the file's last updated time, given a file handle. Caching will be discussed in NFSv2 shortly.

From Protocol To Distributed File System

Hopefully you can see how this protocol is translated into a file system between the client and the file server. The client-side file system keeps track of open files and converts application requests into protocol messages. The server merely responds to protocol messages, which contain all necessary information.

Consider a basic application that reads a file. The diagram (Figure 49.5) shows the application's system calls and how the client-side file system and file server respond.

Observations on the figure For starters, note how the client keeps track of the integer file descriptor's NFS file handle mapping as well as the current file pointer. A properly designed read protocol message tells the server exactly which bytes from the file to read. Successions to a successful read are provided with the same file handle but a different offset.

Second, look for server interactions. When a file is first opened, the client-side file system sends a LOOKUP request. /home/remzi/foo.txt would require three LOOKUPs: one for home, one for remzi, and finally one for foo.txt in remzi.

Third, you may have noticed that each server request has all the information needed to complete. This design point ensures that the server does not need state to react to the request, which we shall examine in more depth later.

Handling Server Failure With Idempotent Operations

A client's message to the server is sometimes ignored. This lack of response could be for numerous causes. In some situations, the network may lose the communication, resulting in the client never receiving a response.

It's also conceivable that the server is down and not responding to messages. The server will be reset later, but all requests will be lost. In each situation, clients are left wondering what to do when the server does not respond promptly.

In NFSv2, a client simply retries the request if any of these fail. After sending the request, the client sets a timer to expire. If you react before the countdown goes off, the timer is canceled. If the timer expires without a response, the client assumes the request was not processed and resends it. If the server responds, the client has solved the issue.

The client can simply retry the request (regardless of the reason for the failure) because most NFS requests are idempotent. An operation is idempotent if the consequence of doing it several times is the same as performing it once. For example, storing a value three times in memory is the same as storing it once, making it an idempotent action. However, incrementing a counter three times has a different result than incrementing it once, therefore it is not idempotent. Generally, any operation that only reads data is idempotent; however, an operation that updates data must be carefully examined to determine if it is.

The idempotency of most common operations lies at the basis of NFS crash recovery. In the absence of an update, LOOKUP and READ queries are trivially idempotent. WRITE requests are also idempotent. If a WRITE fails, the client can simply try again. Data, count, and (most critically) offset are all contained in the WRITE message. It can be repeated knowing that the consequence of several writes is the same as a single write.

So the client can handle all timeouts uniformly. Case 1: A WRITE request is simply lost, but the client retries it, and the server writes it. Similarly, if the server is down when the first request is submitted, but up and running when the second request is sent, everything functions as expected (Case 2). Finally, the server may receive a WRITE request, write to its disk, and respond. Case 3: The client must resend the request if the reply is lost. On re-request, the server simply writes the data to disk and replies that it has done so. Succeeding clients have handled both message loss and server failure uniformly. Neat!

Aside: some operations are difficult to idempotent. For example, when trying to create a directory that already exists, the mkdir request fails. The file server receives an MKDIR protocol message and executes it properly, but the reply is lost. The client may repeat the action and face that failure. So, life isn't ideal.

Client-side Caching

Distributed file systems are useful for many reasons, but transmitting all read and write requests across the network introduces a major performance issue: the network is slower than local memory or disk. So, how can we measure a distributed file system's performance?

Client-side caching, as the huge bold words in the sub-heading above suggest. NFS caches file data (and metadata) read from the server in client memory. While the first access is costly (requiring network connectivity), subsequent accesses are inexpensive (served from client memory).

The cache also acts as a write buffer. Before writing to a file, a client application buffers the data in client memory (in the same cache as the data retrieved from the file server). The application's write() function succeeds instantly (and only puts the data in the client-side file system's cache); the data is only written to the file server afterwards.

So NFS clients cache data and perform well, right? Sadly, no. A system with many client caches creates a huge and interesting challenge we'll call the cache consistency problem.

The Cache Consistency Problem

Problem of cache consistency with two clients and one server. Consider client C1 reading file F and caching it locally. To distinguish the two versions of the file F, let's call the new version (version 2) F[v2] and the old version (version 1) F[v1] (but of course the file has the same name, just different contents). Finally, C3 has not yet accessed file F.

You can probably see the issue coming (Figure 49.7). In reality, there are two. In this example, while F[v2] is in C2's memory, any access to F from another client (say C3) will fetch the old version of the file (F[v1]). Imagine logging into machine C2, updating F, and then logging into C3 and trying to read the file, only to get the old copy! This may be vexing. So, when do modifications from one client become available to other clients?

In this situation, C2 has finally flushed its writes to the file server, so the server has the latest version (F[v2]). However, C1 still contains F[v1] in its cache, so a program running on C1 will obtain the old version (F[v1]) and not the newer version (F[v2]).

NFSv2 solves cache consistency issues in two ways. When a client program writes to and then closes a file, the client flushes all updates (i.e., dirty pages in the cache) to the server. A subsequent open from another node will view the latest file version with flush-on-close consistency.

Second, to avoid stale cache, NFSv2 clients first check for changes in a file before accessing its cached data. For each cached block, the client-side file system issues a GETATTR request to the server. It is crucial to note that if the server time-of-modification is more current than the client cache time-of-modification, then the client invalidates the file. This removes it from the client cache and ensures that subsequent reads will travel to the server and receive the newest version. If the client has the latest version of the file, it will utilise the cached data, improving performance.

An unexpected side effect of the stale cache approach was that the NFS server became overloaded with GETATTR queries. Although the normal occurrence was that a file was accessed only by one client (perhaps frequently), the client had to send GETATTR queries to the server to ensure no one else had altered the file. So a client continuously asks the server "has anyone modified this file?" when no one has.

Each client now has an attribute cache. Clients still check files before accessing them, but they mostly only search for the characteristics in the attribute cache. When a file was first accessed, its characteristics were

cached and timed out (say 3 seconds). All file accesses would then assess that it was safe to use the cached file and proceed without contacting the server.

Assessing NFS Cache Consistency

Final thoughts on NFS cache consistency. While the flush-on-close behavior made sense, it caused a performance issue. The server would still be required to accept a temporary or short-lived file created on a client and subsequently erased. A better method would hold such temporary files in memory until they are erased, reducing server contact and possibly improving performance.

Adding an attribute cache to NFS made it difficult to determine what version of a file one was getting. Because your attribute cache had not yet timed out, the client was delighted to offer you whatever was in client memory. This was great most of the time, but occasionally led to weird behavior.

So that is NFS client caching. It's a fascinating case where implementation details define user-observable semantics, rather than the other way around.

Implications On Server-Side Write Buffering

So far, we've focused on client caching, which is where the intriguing challenges arise. However, NFS servers are often powerful machines with lots of memory, so caching issues arise. When data (or metadata) is read from disk, NFS servers hold it in memory, so subsequent reads don't go to disk, potentially improving performance.

Write buffering is more intriguing. NFS servers must compel a write to stable storage before returning success (e.g., to disk or some other persistent device). Can you figure out why returning success on a WRITE protocol request to the client could result in wrong behavior?

The answer is in our assumptions about client behavior. Consider a client's write sequence:

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 a file's three blocks with a block of a, b, and c. So, assuming the file started out like this:

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz

The x's, y's, and z's would be overwritten with a's, b's, and c's, accordingly.

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccc

Consider for a moment that these three client writes were sent to the server as three separate WRITE protocol messages. Assume the server receives the first WRITE message, writes it to the disk, and informs the client. Assume the second write is only buffered in memory, and the server communicates success to the client before forcing it to disk. The server restarts promptly and receives a third successful write request.

So, the client's requests were successful, but the file contents are surprising:

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy <— oops
ccccccccccccccccccccccccccccccccccccccccccccccccc

Yikes! Because the server informed the client that the second write was successful before committing it to disk, an old chunk remains in the file, potentially causing problems.

To avoid this issue, NFS servers must commit each write to stable (permanent) storage before telling the client of success. So we won't get file con- tents mixed up like in the prior example.

The issue with this requirement in NFS server implementation is that write performance might be a substantial bottleneck. The first trick is to put writes in a battery-backed memory, thereby enabling quick responses to WRITE requests without fear of losing data or incurring the cost of writing to disk immediately; the second trick uses a file system design specifically designed to write to disk quickly when one fina

Summary

The NFS distributed file system was introduced. NFS is designed to provide simple and fast server recovery in the event of server failure. Because a client can safely replay a failed operation, it is OK whether or not the server has performed the request.

We've also seen how caching can complicate a multi-client, single-server system. To be reasonable, the system must tackle the cache consistency issue, which NFS does in an ad hoc manner, resulting in occasionally odd behavior. Finally, we learned about challenging server caching: forcing writes to stable storage before returning success (otherwise data can be lost).

We haven't discussed other aspects, most notably security. Early NFS security was inadequate; any user on a client could easily masquerade as another user and so access nearly any file.

- An NFS stateless protocol is the key to achieving the core goal of rapid and simple crash recovery. After a server crash, clients just retry requests until they succeed.
- Making requests idempotent is a central aspect of the NFS protocol. An operation is idempotent if repeating it has the same result as doing it once. In NFS, idempotency unifies client lost-message retransmission and client server crash handling.
- Client-side caching and write buffering are required for performance, but pose a cache consistency issue.
- NFS implementations engineer cache integrity in different ways: When a file is closed, its contents are forced to the server, allowing other clients to see the updates. An attribute cache decreases the frequency of file change checks with the server (via GETATTR requests).
- Failure to commit writes to persistent media might result in data loss.
- Sun introduced the VFS/Vnode interface to provide NFS integration into the operating system, allowing multiple file system implementations to coexist.