Introduction

The Andrew File System was developed at CMU 1 in the 1980s. This project's principal goal was simple: scale. How can a distributed file system be designed to support as many clients as possible?

Oddly enough, many factors of design and implementation affect scalability. The protocol between clients and servers is crucial. Because each check takes server resources (including CPU and network bandwidth), frequent checks like this will limit the number of customers a server can reply to, limiting scalability.

Unlike NFS, AFS has always prioritized sensible user-visible behavior. Client-side cache timeout intervals, for example, are challenging to describe in NFS cache consistency. AFS cache consistency is straightforward: when a file is opened, the client receives the server's most recent consistent copy.

AFS Version 1

The first system was dubbed the ITC distributed file system, which led to a redesign and the final protocol (called AFSv2, or just AFS) Now we'll look at the first.

EXAMPLE CODE

Whole-file caching on the local disk of the client system is a basic concept of all AFS versions. When you open() a file, the server retrieves the complete file (if it exists) and stores it locally. Read and write activities are diverted to the local file system, requiring no network communication and being quick. Finally, close() flushes the changed file back to the server. Contrast this with NFS, which caches blocks (not files, but it might cache every block of a file) in client memory (not local disk).

Let's dig a little deeper. To start, the AFS client-side code (dubbed Venus by the AFS creators) sends a Fetch protocol message to the server. For example, /home/remzi/notes.txt would be sent to the file server (named Vice) which would traverse the pathname, discover the requested file, and return it to the client. The client-side function would then cache the file locally (by writing it to local disk). As stated previously, AFS read() and write() system calls are entirely local (no contact with the server) and simply redirect to the local copy of the file. Because read() and write() behave like local file system calls, a block may be cached in client memory. So AFS uses client RAM to cache blocks from the local disk. Finally, the AFS client checks if the file has been modified (opened for writing); if so, it sends the modified file and pathname to the server via a Store protocol message.

When the file is accessed again, AFSv1 is substantially more efficient. This code initially contacts the server (using the TestAuth protocol message) to see if the file has changed. If not, the client uses the locally cached copy, avoiding a network transfer. AFSv1 protocol messages are seen in the graphic above. Earlier versions of the protocol just cached file contents, leaving directories on the server.

Problems with Version 1

Lesser known AFS issues prompted the designers to revise their file system. The AFS designers spent a lot of time measuring their existing prototype to find out what was incorrect. Getting good data is essential to understanding how systems work and how to improve them. The authors observed two major issues with AFSv1:

When using the FetchorStore protocol, the client sends the whole pathname (e.g., /home/ remzi/notes.txt) to the server. To access the file, the server must first seek in the root directory for home, then in home for remzi, and so on until the file is found. Many clients visiting the server at once meant the server was spending a lot of time simply walking directory paths.

Too many TestAuth protocol messages: AFSv1 generated a lot of traffic to check if a local file (or its stats) was valid with the TestAuth protocol message, just like NFS does. So servers spent a lot of time telling clients whether they may use cached copies of files. Most of the time, the file had not changed.

Another flaw in AFSv1 was that it didn't load balance and used a single process per client, causing context switching and other overheads. Load

AFSv2 solves the context-switch problem by constructing the server with threads instead of processes. For the sake of space, we will just discuss the two key protocol issues that hampered system scalability.

Improving Protocol

The server CPU became the system's bottleneck, and each server could only serve 20 clients without being overloaded. Servers were receiving too many TestAuth messages and spent too much time traversing the directory structure. So the AFS designers had a problem:

THE CRUX: SCALABLE FILE PROTOCOL DESIGN

How could the protocol be redesigned to limit the number of server interactions, i.e., TestAuth messages?
How could they build the protocol to make server interactions more efficient? A new protocol addressing these difficulties would result in a far more scalable AFS.

AFS Version 2

AFSv2 introduced callbacks to decrease client/server interactions. A callback is essentially a promise from the server to notify the client when a cached file is updated. The client no longer needs to contact the server to check the validity of a cached file. Like polling, it assumes the file is legitimate until the server notifies it otherwise.

As with NFS, AFSv2 introduces a file identifier (FID) instead of pathnames to identify a file. AFS FIDs have three parts: volume, file, and “uniquifier” (to enable reuse of the volume and file IDs when a file is deleted). As a result, instead of sending whole pathnames to the server, the client would walk them one by one, storing the results and hopefully lowering server demand.

INSERT TABLE

In this case, the client would first Fetch the contents of home, cache them locally, and then set up a callback on home. The client would then fetch remzi, store it locally, and set up a callback on it. This file descriptor is returned to the calling application. Figure 50.2 summarizes.

Unlike NFS, the AFS client would establish a callback with the server with each directory or file fetch.

the server would tell the client if its cached state changed. In addition to establishing callbacks for all folders and the file notes.txt, the first access to /home/remzi/notes.txt creates multiple client-server messages (as explained above). So, when a file is cached locally, AFS acts virtually equivalent to a disk-based file system. If a file is accessed twice, the second access should be as fast as accessing it locally.

Cache Consistency

When discussing NFS, we examined update visibility and cache staleness. With update visibility, the question is when will the server be updated? With cache staleness, how long before clients see the updated version instead of an older cached copy?

AFS cache consistency is straightforward to define and comprehend because to callbacks and whole-file caching. It is crucial to consider consistency between processes on various machines and consistency inside the same machine.

AFS makes updates visible on the server and invalidates cached copies at the same time, when the modified file closes. A client opens and writes to a file (perhaps repeatedly). When it’s done, the server flushes the new file (and thus visible). The server then “breaks” callbacks for any clients with cached copies by contacting each client and alerting them that their callback is no longer valid. Clients will no longer be able to read stale copies of the file; further opens will require a re-fetch of the latest version from the server (and will also serve to reestablish a callback on the new version of the file).

INSERT TABLE

AFS deviates from this fundamental concept inside a single process. Other local processes can observe the file writes in this instance (i.e., a process does not have to wait until a file is closed to see its latest updates). Using a single machine is exactly as expected, as this behavior is based on standard UNIX semantics. You’d only notice the AFS consistency mechanism if you switched machines.

One cross-machine situation warrants additional discussion. In the unusual scenario that many processes on different machines alter a file at the same time, AFS uses the last writer wins technique (which perhaps should be called last closer wins). Specifically, whatever client calls close() last will update the complete file on the server last, thus becoming the “winning” file. The result is a file created entirely by one client or the other. Unlike a block-based protocol like NFS, where individual block updates are flushed to the server, the final file on the server may contain updates from both clients. Imagine a JPEG image being modified by two clients in portions; the resulting mix of writes would not likely be a valid JPEG.

Figure 50.3 depicts a timeline of many events. They show two processes (P1 and P2) on Client1 and their cache states, one process (P3) on Client2 and their cache states, and the server (Server) all working on a single file (F).

For the server, the figure merely shows the file’s contents following the left operation. Read it and try to understand why each read produces the consequences it does. If you get stuck, the right commentary field will help.

Crash Recovery

As you can see, crash recovery is more involved than NFS. You are correct. Imagine a situation when a server (S) cannot access a client (C1) for a brief time, say while the client is rebooting. Suppose C1 had file F cached on its local disk, and then C2 (another client) modified F, leading S to send messages to all clients caching the file to remove it from their local caches. Because C1 may miss vital communications after rebooting, C1 should suspect all cache contents when rejoining the system. If the server's cached copy of file F is still valid, C1 can use it; if not, C1 should fetch the latest version from the server.

INSERT TABLE

Server recovery is also more difficult. Because callbacks are retained in memory, a server cannot tell which client machine has particular files when it restarts. Thus, upon server restart, each client must recognize that the server has crashed, suspect all cache contents, and verify the legitimacy of files before utilizing them. The importance of notifying clients of server failures is underscored by the risk of stale files being accessed. There are numerous ways to implement such recovery, such as having the server send a message to each client saying "don't trust your cache contents!" or having clients periodically check that the server is still alive (with a heartbeat message, as it is called). Building a scalable and appropriate caching architecture is not free; with NFS, clients barely noticed a server crash.

Scale and Performance of Version 2

With the new protocol, AFSv2 was proven to be far more scalable than AFSv1. Each server could handle around 50 clients (instead of just 20). Also, because most file accesses were local, file reads frequently went to the local disk cache, therefore client-side performance was often comparable to local performance (and potentially, local memory). Only when a client produced a new file or modified an existing one did a Store message need to be sent to the server.

Let's compare AFS performance to NFS performance in common file-system access scenarios. Our qualitative comparison is shown in Figure 50.4.

The image shows typical read and write patterns for files of various sizes. Medium and big files have NL blocks while small files have Ns blocks. Large files can be stored on a local disk but not in client memory.

For analysis purposes, we additionally assume that a Lnet time unit access to a distant server for a file block. Local memory requires Lmem, and local disk requires Ldisk. Lnet > Ldisk > Lmem.
For our final assumption, we don't cache the first file access. All subsequent file visits (re-reads) are assumed to reach caches, if available.

The figure's columns illustrate how long an operation (such a small file sequential read) takes on NFS or AFS. The right column shows the AFS/NFS ratio.

We note the following. First, the performance of each system is often comparable. On both systems, the time to fetch a file from a remote server dominates when reading a file (Workloads 1, 3, 5). It's possible that AFS is slower since it has to write the file locally, but that's probably because the local (client-side) file system cache buffers such writes. Also, because AFS keeps the cached copy on disk, you might expect AFS reads to be slower. However, AFS reads would likely hit the client-side memory cache, and speed would be equivalent to NFS.

In a huge file sequential re-read, an intriguing distinction occurs (Workload 6). Because AFS has a big local disk cache, it will re-access the file from there. Because NFS can only cache blocks in client memory, re-reading a large file (larger than local memory) requires the NFS client to re-fetch the full file from the faraway server. In this scenario, AFS outperforms NFS by a factor of Lnet, assuming remote access is slower than local disk. In this scenario, NFS increases server load, which impacts scaling.

We should see identical results on both systems for sequential writes (new files) (Workloads 8, 9). When the file is closed, the AFS client will compel writes to the server as per protocol. Client-side memory pressure may force certain blocks to the server, but they will be written to the server when the file is closed to maintain NFS flush-on-close consistency. Because AFS writes data to local disk, you might expect it to be slower. To boost performance, keep in mind that AFS writes to a local file system, which is then committed to the page cache, and then (in the background) to disk.

AFS performs worse on a sequential file overwrite (Workload 10). So far, we've assumed that workloads that write create a new file also overwrite existing files. Overwrite is a terrible scenario for AFS because the client first downloads the old file, then overwrites it. To prevent the initial (useless) read2, NFS will just overwrite blocks.

NFS also outperforms AFS for workloads that access small subsets of data within huge files (Workloads 7, 11). In these circumstances, the AFS protocol opens the file and retrieves the complete file, but only performs a short read or write. Even worse, modifying the file requires rewriting it to the server, compounding the performance hit. NFS, being a block-based protocol, performs I/O proportional to the read or write size.

Overall, NFS and AFS make distinct assumptions, which leads to differing performance consequences. Workload determines whether these differences matter.

Other Improvements

Like Berkeley FFS (which included symbolic links and other capabilities), AFS designers added features to make the system easier to use and administer. By providing a genuine global namespace to clients, AFS ensures that all files are named the same way. Due to the fact that NFS permits each client to mount NFS servers differently, it is only by convention (and a lot of work) that files are called similarly across clients.

AFS also takes security seriously, incorporating means to authenticate users and keep files secret if requested. For many years, NFS had very limited security support.

AFS also allows for user-controlled access. Using AFS, a user may regulate who can access which files. NFS, like most UNIX file systems, lacks this feature.

Finally, AFS introduces tools to make server management easier for system administrators. AFS was much ahead of the curve in terms of system administration.

Summary

AFS shows us how to build distributed file systems in a different way than NFS. By decreasing server contacts (via whole-file caching and callbacks), each server may accommodate multiple clients, reducing the number of servers required to manage a site. AFS's single namespace, security, and access-control lists make it easy to use. The AFS consistency paradigm is easy to understand and reason about, and does not lead to the odd behavior seen in NFS.

Regrettably, AFS is expected to decline. Because NFS is an open standard, many companies support it, and it dominates the market together with CIFS (the Windows-based distributed file system protocol). While AFS installations are still seen from time to time (such as at Wisconsin educational institutions), the system's concepts are likely to be the only lasting influence. NFSv4 now includes server state (e.g., a "open" protocol message) and so resembles the fundamental AFS protocol.