48

The Andrew File System (AFS)

The Andrew File System was introduced by researchers at Carnegie-Mellon University (CMU) in the 1980's [H+88]. Led by the well-known Professor M. Satyanarayanan of Carnegie-Mellon University ("Satya" for short), the main goal of this project was simple: **scale**. Specifically, how can one design a distributed file system such that a server can support as many clients as possible?

Interestingly, as we will see, there are numerous design and implementation components that affect scalability. Most important is the design of the *protocol* between clients and servers. In NFS, for example, the protocol forces clients to check with the server periodically to determine if cached contents have changed; because each check uses server resources (e.g., CPU, network bandwidth, etc.), frequent checks like this will limit the number of clients a server can respond to and thus limit scalability.

48.1 AFS Version 1

We will discuss two versions of AFS [H+88,S+85]. The first version (which we will call AFSv1, but actually the original system was called the ITC distributed file system [S+85]) had some of the basic design in place, but didn't scale as desired, which led to a re-design and the final protocol (which we will call AFSv2, or just AFS) [H+88]. We now discuss the first version.

One of the basic tenets of all versions of AFS is **whole-file caching** on the **local disk** of the client machine that is accessing a file. When you open() a file, the entire file (if it exists) is fetched from the server

and stored in a file on your local disk. Subsequent application read() and write() operations are redirected to the local file system where the file is stored; thus, these operations require no network communication and are fast. Finally, upon close(), the file (if it has been modified) is flushed back to the server. Note the obvious contrasts with NFS, which caches **blocks** (not whole files, although NFS could of course cache every block of an entire file) and does so in client **memory** (not local disk).

Let's get into the details a bit more. When a client application first calls open(), the AFS client-side code (which the AFS designers call **Venus**) would send a Fetch protocol message to the server. The Fetch message would pass the entire pathname (e.g., `/home/remzi/notes.txt`) of the desired file to the file server (the group of which they called **Vice**), which would then traverse the pathname, find the desired file, and ship the entire file back to the client. The client-side code would then cache the file on the local disk of the client (by writing it to local disk). As we said above, subsequent read() and write() system calls are strictly *local* in AFS (no communication with the server occurs); they are just redirected to the local copy of the file. Because the read() and write() calls act just like calls to a local file system, once a block is accessed, it also may be cached in client memory. Thus, AFS also uses client memory to cache copies of blocks that it has in its local disk. Finally, when finished, the AFS client checks if the file has been modified (i.e., that it has been opened for writing); if so, it flushes the new version back to the server with a Store protocol message, sending the entire file and pathname to the server for permanent storage.

```
TestAuth      Test whether a file has changed (used to validate cached entries)
GetFileStat   Get the stat info for a file
Fetch         Fetch the contents of an entire file from the server
Store         Store this file on the server
SetFileStat   Set the stat info for a file
ListDir       List the contents of a directory
```

Figure 48.1: AFSv1 Protocol Highlights

The next time the file is accessed, AFSv1 does so much more efficiently. Specifically, the client-side code first contacts the server (using the TestAuth protocol message) in order to determine whether the file has changed. If not, the client would use the locally-cached copy, thus improving performance by avoiding a network transfer. The figure above shows some of the protocol messages in AFSv1.

Note that this early version of the protocol only cached file contents; directories, for example, were only kept at the server.

48.2 Problems with Version 1

A few key problems with this first version of AFS motivated the designers to rethink their file system. To study the problems in detail, the designers of AFS spent a great deal of time measuring their existing prototype to find what was wrong. Such experimentation is a good thing; **measurement** is the key to understanding how systems work and how to improve them. Hard data helps take intuition and make into a concrete science of deconstructing systems. In their study, the authors found two main problems with AFSv1:

- **Path-traversal costs are too high**: When performing a Fetch or Store, the client passes the entire file name (e.g., the file `/home/remzi/grades.txt`) to the server. The server, in order to access the file, must perform a full pathname traversal, first looking in the root directory to find `home`, then in `home` to find `remzi`, and so forth, all the way down the path until finally the desired file is located. With many clients accessing the server at once, the designers of AFS found that the server was spending much of its time simply walking down directory paths!
- **The client issues too many TestAuths to the server**: Much like NFS and its overabundance of GetAttr protocol messages, AFSv1 generated a large amount of traffic to check whether a local file (or its stat information) was valid with the TestAuth protocol message. Thus, servers spent a great deal of time telling clients whether it was OK to used their cached copies of a file. Most of the time, it was OK (of course), and thus the protocol was leading to high server overheads again.

There were actually two other problems with AFSv1: load was not balanced across servers, and the server used a single distinct process per client thus inducing context switching and other overheads. The load imbalance problem was solved by introducing **volumes**, which an administrator could move across servers to balance load; the context-switch problem was solved in AFSv2 by building

the server with threads instead of processes. However, for the sake of space, we focus here on the main two protocol problems above that limited the scale of the system.

48.3 Improving the Protocol

The two problems above limited the scalability of AFS; the server CPU became the bottleneck of the system, and each server could only service 20 clients without becoming overloaded. Servers were receiving too many TestAuth messages, and when they received Fetch or Store messages, were spending too much time traversing the directory hierarchy. Thus, the AFS designers were faced with a problem:

THE CRUX: HOW TO DESIGN A PROTOCOL FOR SCALABILITY
How should one redesign the protocol to minimize the number of server interactions, i.e., how could they reduce the number of TestAuth messages? Further, how could they design the protocol to make these server interactions efficient? By attacking both of these issues, a new protocol would result in a much more scalable version AFS.

48.4 AFS Version 2

AFSv2 introduced the notion of a **callback** to reduce the number of client/server interactions. A callback is simply a promise from the server to the client that the server will inform the client when a file that the client is caching has been modified. By adding this **state** to the server, the client no longer needs to contact the server to find out if a cached file is still valid; rather, it assumes that the file is valid until the server tells it otherwise.

AFSv2 also introduced the notion of a **file handle** (very similar to NFS) instead of pathnames to specify which file a client was interested in. A file handle in AFS consisted of a volume identifier, a file identifier, and a generation number. Thus, instead of sending whole pathnames to the server and letting the server walk the pathname to find the desired file, the client would walk the pathname, one piece at a time, caching the results and thus hopefully reducing the load

on the server.

For example, if a client accessed `/home/remzi/notes.txt`, and home was the AFS directory mounted onto `/` (in other words, `/` was the local root directory, but `home` and its children were in AFS), the client would first Fetch the directory contents of `home`, put them in the local-disk cache, and setup a callback on `home`. Then, the client would Fetch the directory `remzi`, put it in the local-disk cache, and setup a callback on the server on `remzi`. Finally, the client would Fetch `notes.txt`, cache this regular file in the local disk, setup a callback, and finally return a file descriptor to the calling application.

The key difference, however, from NFS, is that with each fetch of a directory or file, the AFS client would establish a callback with the server, thus ensuring that the server would notify the client of a change in its cached state. The benefit is obvious: although the first access to `/home/remzi/notes.txt` generates many client-server messages (as described above), it also establishes callbacks for all the directories as well as the file notes.txt, and thus subsequent accesses are entirely local and require no server interaction at all. Thus, in the common case where a file is cached at the client, AFS behaves nearly identically to a local disk-based file system. If one accesses a file more than once, the second access should be just as fast as accessing a file locally.

48.5 Cache Consistency

Because of callbacks and whole-file caching, the cache consistency model provided by AFS is easy to describe and understand. When a client (C1) opens a file, it will fetch it from the server. Any updates it makes to the file are entirely local, and thus only visible to other applications on that same client (C1); if an application on another client (C2) opens the file at this point, it will just get the version that is stored at the server which does not yet reflect the changes being made at C1. When the application at C1 finishes updating the file, it calls close() which flushes the entire file to the server. At that point, any clients caching the file (such as C2) would be informed that their callbacks are broken and thus they should not use cached versions of the file because the server has a newer version.

In the rare case that two clients are modifying a file at the same time, AFS naturally employs what is known as a **last writer wins**

approach. Specifically, whichever client calls close() last will update the entire file on the server last and thus will be the winning file, i.e., the file that remains on the server for others to see. The result is a file that is either one client's or the other client's. Note the difference from a block-based protocol like NFS: in such a block-based protocol, writes of individual blocks may be flushed out to the server as each client is updating the file, and thus the final file on the server could end up as a mix of updates from both clients; in many cases, such a mixed file output would not make much sense (i.e., imagine a JPEG image getting modified by two clients in pieces; the resulting mix of writes would hardly make much sense).

48.6 Crash Recovery

From the description above, you might sense that crash recovery is more involved than with NFS. You would be right. For example, imagine there is a short period of time where a server (S) is not able to contact a client (C1), for example, while the client C1 is rebooting. While C1 is not available, S may have tried to send it one or more callback recall messages; for example, imagine C1 had file F cached on its local disk, and then C2 (another client) updated F, thus causing S to send messages to all clients caching the file to remove it from their local caches. Because C1 may miss those critical messages when it is rebooting, upon rejoining the system, C1 should treat all of its cache contents as suspect. Thus, upon the next access to file F, C1 should first ask the server (with a TestAuth protocol message) whether its cached copy of file F is still valid; if so, C1 can use it; if not, C1 should fetch the newer version from the server.

Server recovery after a crash is more complicated. The problem that arises is that callbacks are kept in-memory; thus, when a server reboots, it has no idea which client machine has which files. Thus, upon server restart, each client of the server must realize that the server has crashed and treat all of their cache contents as suspect, and (as above) reestablish the validity of a file before using it. Thus, a server crash is a big event, as one must ensure that each client is aware of the crash in a timely manner, or risk a client accessing a stale file. There are many ways to implement such recovery; for example, by having the server send a message (saying "don't trust your cache contents!") to each client when it is up and running again. As

you can see, there is a cost to building a more scalable and sensible caching model; with NFS, clients hardly noticed a server crash.

48.7 Scale of AFSv2

With the new protocol in place, AFSv2 was measured and found to be much more scalable that the original version. Indeed, each server could support about 50 clients (instead of just 20). A further benefit was that client-side performance often came quite close to local performance, because in the common case, all file accesses were local; file reads usually went to the local disk cache (and potentially, local memory). Only when a client created a new file or wrote to an existing one was there need to send a Store message to the server and thus update the file with new contents.

48.8 Other Improvements: Namespaces, Security, Etc.

AFS added a number of other improvements beyond scale. It provided a true global namespace to clients, thus ensuring that all files were named the same way on all client machines; NFS, in contrast, allowed each client to mount NFS servers in any way that they pleased, and thus only by convention (and great administrative effort) would files be named similarly across clients.

AFS also took security seriously, and incorporated mechanisms to authenticate users and ensure that a set of files could be kept private if a user so desired. NFS, in contrast, still has quite primitive support for security.

Finally, AFS also included facilities for flexible user-managed access control. Thus, when using AFS, a user has a great deal of control over who exactly can access which files. NFS, like most UNIX file systems, has much more primitive support for this type of sharing.

48.9 Summary

AFS shows us how distributed file systems can be built quite differently than what we saw with NFS. The protocol design of AFS is particularly important; by minimizing server interactions (through whole-file caching and callbacks), each server can support many clients

and thus reduce the number of servers needed to manage a particular site. Many other features, including the single namespace, security, and access-control lists, make AFS quite nice to use. Finally, the consistency model provided by AFS is simple to understand and reason about, and does not lead to the occasional weird behavior as one sometimes observes in NFS.

Perhaps unfortunately, AFS is likely on the decline. Because NFS became an open standard, many different vendors supported it, and, along with CIFS (the Windows-based distributed file system protocol), NFS dominates the marketplace. Although one still sees AFS installations from time to time (such as in various educational institutions, including Wisconsin), the only lasting influence will likely be from the ideas of AFS rather than the actual system itself. Indeed, NFSv4 now adds server state (e.g., an "open" protocol message), and thus bears more similarity to AFS than it used to.

References

[H+88] "Scale and Performance in a Distributed File System"
John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West.
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988.

[S+85] "The ITC Distributed File System: Principles and Design"
M. Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A. Spector, M.J. West.
SOSP '85. pages 35-50.