50

The Andrew File System (AFS)

The Andrew File System was introduced at Carnegie-Mellon University (CMU) [1] 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, there are numerous aspects of design and implementation 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 (including CPU and network bandwidth), frequent checks like this will limit the number of clients a server can respond to and thus limit scalability.

AFS also differs from NFS in that from the beginning, reasonable user-visible behavior was a first-class concern. In NFS, cache consistency is hard to describe because it depends directly on low-level implementation details, including client-side cache timeout intervals. In AFS, cache consistency is simple and readily understood: when the file is opened, a client will generally receive the latest consistent copy from the server.

50.1 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.

---

[1] Though originally referred to as "Carnegie-Mellon University", CMU later dropped the hyphen, and thus was born the modern form, "Carnegie Mellon University." As AFS derived from work in the early 80's, we refer to CMU in its original fully-hyphenated form. See `https://www.quora.com/When-did-Carnegie-Mellon-University-remove-the-hyphen-in-the-university-name` for more details, if you are into really boring minutiae.

`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 file`
`Store`	`Store this file on the server`
`SetFileStat`	`Set the stat info for a file`
`ListDir`	`List the contents of a directory`

Figure 50.1: **AFSv1 Protocol Highlights**

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 protocol message would pass the entire pathname of the desired file (for example, `/home/remzi/notes.txt`) 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.

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.

TIP: MEASURE THEN BUILD (PATTERSON'S LAW)

One of our advisors, David Patterson (of RISC and RAID fame), used to always encourage us to measure a system and demonstrate a problem *before* building a new system to fix said problem. By using experimental evidence, rather than gut instinct, you can turn the process of system building into a more scientific endeavor. Doing so also has the fringe benefit of making you think about how exactly to measure the system before your improved version is developed. When you do finally get around to building the new system, two things are better as a result: first, you have evidence that shows you are solving a real problem; second, you now have a way to measure your new system in place, to show that it actually improves upon the state of the art. And thus we call this **Patterson's Law**.

50.2 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, because **measurement** is the key to understanding how systems work and how to improve them; obtaining concrete, good data is thus a necessary part of systems construction. In their study, the authors found two main problems with AFSv1:

- **Path-traversal costs are too high**: When performing a Fetch or Store protocol request, the client passes the entire pathname (e.g., `/home/remzi/notes.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 CPU time simply walking down directory paths.

- **The client issues too many TestAuth protocol messages**: 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 much of their time telling clients whether it was OK to use their cached copies of a file. Most of the time, the answer was that the file had not changed.

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.

50.3 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 SCALABLE FILE PROTOCOL

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.

50.4 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 system, 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; notice the analogy to **polling** versus **interrupts**.

AFSv2 also introduced the notion of a **file identifier** (**FID**) (similar to the NFS **file handle**) instead of pathnames to specify which file a client was interested in. An FID in AFS consists of a volume identifier, a file identifier, and a "uniquifier" (to enable reuse of the volume and file IDs when a file is deleted). 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 the file `/home/remzi/notes.txt`, and `home` was the AFS directory mounted onto / (i.e., / 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 set up a callback on `home`. Then, the client would Fetch the directory

Client (C_1)	**Server**
fd = open("/home/remzi/notes.txt", ...); Send Fetch (home FID, "remzi")	
	Receive Fetch request look for remzi in home dir setup callback(C_1) on remzi return remzi's content/FID
Receive Fetch reply write remzi to local disk cache record callback status of remzi Send Fetch (remzi FID, "notes.txt")	
	Receive Fetch request look for notes.txt in remzi dir setup callback(C_1) on notes.txt return notes.txt's content/FID
Receive Fetch reply write notes.txt to local disk cache record callback status of notes.txt local `open()` of cached notes.txt return file descriptor to application	
read(fd, buffer, MAX); perform local `read()` on cached copy	
close(fd); do local `close()` on cached copy if file has changed, flush to server	

fd = open("/home/remzi/notes.txt", ...);
Foreach dir (home, remzi)
if (callback(dir) == VALID)
use local copy for lookup(dir)
else
Fetch (as above)
if (callback(notes.txt) == VALID)
open local cached copy
return file descriptor to it
else
Fetch (as above) then open and return fd	

Figure 50.2: **Reading A File: Client-side And File Server Actions**

`remzi`, put it in the local-disk cache, and set up a callback on `remzi`. Finally, the client would Fetch `notes.txt`, cache this regular file in the local disk, set up a callback, and finally return a file descriptor to the calling application. See Figure 50.2 for a summary.

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,

ASIDE: CACHE CONSISTENCY IS NOT A PANACEA

When discussing distributed file systems, much is made of the cache consistency the file systems provide. However, this baseline consistency does not solve all problems with regards to file access from multiple clients. For example, if you are building a code repository, with multiple clients performing check-ins and check-outs of code, you can't simply rely on the underlying file system to do all of the work for you; rather, you have to use explicit **file-level locking** in order to ensure that the "right" thing happens when such concurrent accesses take place. Indeed, any application that truly cares about concurrent updates will add extra machinery to handle conflicts. The baseline consistency described in this chapter and the previous one are useful primarily for casual usage, i.e., when a user logs into a different client, they expect some reasonable version of their files to show up there. Expecting more from these protocols is setting yourself up for failure, disappointment, and tear-filled frustration.

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.

50.5 Cache Consistency

When we discussed NFS, there were two aspects of cache consistency we considered: **update visibility** and **cache staleness**. With update visibility, the question is: when will the server be updated with a new version of a file? With cache staleness, the question is: once the server has a new version, how long before clients see the new version instead of an older cached copy?

Because of callbacks and whole-file caching, the cache consistency provided by AFS is easy to describe and understand. There are two important cases to consider: consistency between processes on *different* machines, and consistency between processes on the *same* machine.

Between different machines, AFS makes updates visible at the server and invalidates cached copies at the exact same time, which is when the updated file is closed. A client opens a file, and then writes to it (perhaps repeatedly). When it is finally closed, the new file is flushed to the server (and thus visible). At this point, the server then "breaks" callbacks for any clients with cached copies; the break is accomplished by contacting each client and informing it that the callback it has on the file is no longer

	Client$_1$		**Client$_2$**		**Server**	*Comments*
P$_1$	**P$_2$**	**Cache**	**P$_3$**	**Cache**	**Disk**	
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → A	A	A	do not see writes...
		B	close()	A	A	
close()		B		~~A~~	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() → B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		~~C~~	D	
		D	open(F)	D	D	Unfortunately for P$_3$
		D	read() → D	D	D	the last writer wins
		D	close()	D	D	

Figure 50.3: **Cache Consistency Timeline**

valid. This step ensures that clients will no longer read stale copies of the file; subsequent opens on those clients will require a re-fetch of the new version of the file from the server (and will also serve to reestablish a callback on the new version of the file).

AFS makes an exception to this simple model between processes on the same machine. In this case, writes to a file are immediately visible to other local processes (i.e., a process does not have to wait until a file is closed to see its latest updates). This makes using a single machine behave exactly as you would expect, as this behavior is based upon typical UNIX semantics. Only when switching to a different machine would you be able to detect the more general AFS consistency mechanism.

There is one interesting cross-machine case that is worthy of further discussion. Specifically, in the rare case that processes on different machines are modifying a file at the same time, AFS naturally employs what is known as a **last writer wins** approach (which perhaps should be called **last closer wins**). 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 was generated in its entirety either by one client or the other. Note the difference from a block-based protocol like NFS: in NFS, 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 not likely constitute a valid JPEG.

A timeline showing a few of these different scenarios can be seen in Figure 50.3. The columns show the behavior of two processes (P_1 and P_2) on Client_1 and its cache state, one process (P_3) on Client_2 and its cache state, and the server (Server), all operating on a single file called, imaginatively, `F`. For the server, the figure simply shows the contents of the file after the operation on the left has completed. Read through it and see if you can understand why each read returns the results that it does. A commentary field on the right will help you if you get stuck.

50.6 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 also 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, or by having clients check that the server is alive periodically (with a **heartbeat** message, as it is called). 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.

Workload	**NFS**	**AFS**	$\frac{\textbf{AFS}}{\textbf{NFS}}$
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	L_{net}	$N_L \cdot L_{net}$	N_L
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

Figure 50.4: **Comparison: AFS vs. NFS**

50.7 Scale And Performance Of AFSv2

With the new protocol in place, AFSv2 was measured and found to be much more scalable than 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.

Let us also gain some perspective on AFS performance by comparing common file-system access scenarios with NFS. Figure 50.4 (page 9) shows the results of our qualitative comparison.

In the figure, we examine typical read and write patterns analytically, for files of different sizes. Small files have N_s blocks in them; medium files have N_m blocks; large files have N_L blocks. We assume that small and medium files fit into the memory of a client; large files fit on a local disk but not in client memory.

We also assume, for the sake of analysis, that an access across the network to the remote server for a file block takes L_{net} time units. Access to local memory takes L_{mem}, and access to local disk takes L_{disk}. The general assumption is that $L_{net} > L_{disk} > L_{mem}$.

Finally, we assume that the first access to a file does not hit in any caches. Subsequent file accesses (i.e., "re-reads") we assume will hit in caches, if the relevant cache has enough capacity to hold the file.

The columns of the figure show the time a particular operation (e.g., a small file sequential read) roughly takes on either NFS or AFS. The rightmost column displays the ratio of AFS to NFS.

We make the following observations. First, in many cases, the performance of each system is roughly equivalent. For example, when first reading a file (e.g., Workloads 1, 3, 5), the time to fetch the file from the remote server dominates, and is similar on both systems. You might think AFS would be slower in this case, as it has to write the file to local disk; however, those writes are buffered by the local (client-side) file system cache and thus said costs are likely hidden. Similarly, you might think that AFS reads from the local cached copy would be slower, again because AFS stores the cached copy on disk. However, AFS again benefits here from local file system caching; reads on AFS would likely hit in the client-side memory cache, and performance would be similar to NFS.

Second, an interesting difference arises during a large-file sequential re-read (Workload 6). Because AFS has a large local disk cache, it will access the file from there when the file is accessed again. NFS, in contrast, only can cache blocks in client memory; as a result, if a large file (i.e., a file bigger than local memory) is re-read, the NFS client will have to re-fetch the entire file from the remote server. Thus, AFS is faster than NFS in this case by a factor of $\frac{L_{net}}{L_{disk}}$, assuming that remote access is indeed slower than local disk. We also note that NFS in this case increases server load, which has an impact on scale as well.

Third, we note that sequential writes (of new files) should perform similarly on both systems (Workloads 8, 9). AFS, in this case, will write the file to the local cached copy; when the file is closed, the AFS client will force the writes to the server, as per the protocol. NFS will buffer writes in client memory, perhaps forcing some blocks to the server due to client-side memory pressure, but definitely writing them to the server when the file is closed, to preserve NFS flush-on-close consistency. You might think AFS would be slower here, because it writes all data to local disk. However, realize that it is writing to a local file system; those writes are first committed to the page cache, and only later (in the background) to disk, and thus AFS reaps the benefits of the client-side OS memory caching infrastructure to improve performance.

Fourth, we note that AFS performs worse on a sequential file overwrite (Workload 10). Thus far, we have assumed that the workloads that write are also creating a new file; in this case, the file exists, and is then over-written. Overwrite can be a particularly bad case for AFS, because the client first fetches the old file in its entirety, only to subsequently overwrite it. NFS, in contrast, will simply overwrite blocks and thus avoid the initial (useless) read[2].

Finally, workloads that access a small subset of data within large files perform much better on NFS than AFS (Workloads 7, 11). In these cases, the AFS protocol fetches the entire file when the file is opened; unfortunately, only a small read or write is performed. Even worse, if the file is modified, the entire file is written back to the server, doubling the per-

---

[2] We assume here that NFS writes are block-sized and block-aligned; if they were not, the NFS client would also have to read the block first. We also assume the file was *not* opened with the O_TRUNC flag; if it had been, the initial open in AFS would not fetch the soon to be truncated file's contents.

ASIDE: THE IMPORTANCE OF WORKLOAD

One challenge of evaluating any system is the choice of **workload**. Because computer systems are used in so many different ways, there are a large variety of workloads to choose from. How should the storage system designer decide which workloads are important, in order to make reasonable design decisions?

The designers of AFS, given their experience in measuring how file systems were used, made certain workload assumptions; in particular, they assumed that most files were not frequently shared, and accessed sequentially in their entirety. Given those assumptions, the AFS design makes perfect sense.

However, these assumptions are not always correct. For example, imagine an application that appends information, periodically, to a log. These little log writes, which add small amounts of data to an existing large file, are quite problematic for AFS. Many other difficult workloads exist as well, e.g., random updates in a transaction database.

One place to get some information about what types of workloads are common are through various research studies that have been performed. See any of these studies for good examples of workload analysis [B+91, H+11, R+00, V99], including the AFS retrospective [H+88].

formance impact. NFS, as a block-based protocol, performs I/O that is proportional to the size of the read or write.

Overall, we see that NFS and AFS make different assumptions and not surprisingly realize different performance outcomes as a result. Whether these differences matter is, as always, a question of workload.

50.8 AFS: Other Improvements

Like we saw with the introduction of Berkeley FFS (which added symbolic links and a number of other features), the designers of AFS took the opportunity when building their system to add a number of features that made the system easier to use and manage. For example, AFS provides a true global namespace to clients, thus ensuring that all files were named the same way on all client machines. NFS, in contrast, allows each client to mount NFS servers in any way that they please, and thus only by convention (and great administrative effort) would files be named similarly across clients.

AFS also takes security seriously, and incorporates mechanisms to authenticate users and ensure that a set of files could be kept private if a user so desired. NFS, in contrast, had quite primitive support for security for many years.

AFS also includes 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 less support for this type of sharing.

Finally, as mentioned before, AFS adds tools to enable simpler management of servers for the administrators of the system. In thinking about system management, AFS was light years ahead of the field.

50.9 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. 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 an increasing similarity to the basic AFS protocol.

References

[B+91] "Measurements of a Distributed File System" by Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, John Ousterhout. SOSP '91, Pacific Grove, California, October 1991. *An early paper measuring how people use distributed file systems. Matches much of the intuition found in AFS.*

[H+11] "A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications" by Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. SOSP '11, New York, New York, October 2011. *Our own paper studying the behavior of Apple Desktop workloads; turns out they are a bit different than many of the server-based workloads the systems research community usually focuses upon. Also a good recent reference which points to a lot of related work.*

[H+88] "Scale and Performance in a Distributed File System" by 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), Volume 6:1, February 1988. *The long journal version of the famous AFS system, still in use in a number of places throughout the world, and also probably the earliest clear thinking on how to build distributed file systems. A wonderful combination of the science of measurement and principled engineering.*

[R+00] "A Comparison of File System Workloads" by Drew Roselli, Jacob R. Lorch, Thomas E. Anderson. USENIX '00, San Diego, California, June 2000. *A more recent set of traces as compared to the Baker paper [B+91], with some interesting twists.*

[S+85] "The ITC Distributed File System: Principles and Design" by M. Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A. Spector, M.J. West. SOSP '85, Orcas Island, Washington, December 1985. *The older paper about a distributed file system. Much of the basic design of AFS is in place in this older system, but not the improvements for scale. The name change to "Andrew" is an homage to two people both named Andrew, Andrew Carnegie and Andrew Mellon. These two rich dudes started the Carnegie Institute of Technology and the Mellon Institute of Industrial Research, respectively, which eventually merged to become what is now known as Carnegie Mellon University.*

[V99] "File system usage in Windows NT 4.0" by Werner Vogels. SOSP '99, Kiawah Island Resort, South Carolina, December 1999. *A cool study of Windows workloads, which are inherently different than many of the* UNIX*-based studies that had previously been done.*

Homework (Simulation)

This section introduces `afs.py`, a simple AFS simulator you can use to shore up your knowledge of how the Andrew File System works. Read the README file for more details.

Questions

1. Run a few simple cases to make sure you can predict what values will be read by clients. Vary the random seed flag (`-s`) and see if you can trace through and predict both intermediate values as well as the final values stored in the files. Also vary the number of files (`-f`), the number of clients (`-C`), and the read ratio (`-r`, from between 0 to 1) to make it a bit more challenging. You might also want to generate slightly longer traces to make for more interesting interactions, e.g., (`-n 2` or higher).
2. Now do the same thing and see if you can predict each callback that the AFS server initiates. Try different random seeds, and make sure to use a high level of detailed feedback (e.g., `-d 3`) to see when callbacks occur when you have the program compute the answers for you (with `-c`). Can you guess exactly when each callback occurs? What is the precise condition for one to take place?
3. Similar to above, run with some different random seeds and see if you can predict the exact cache state at each step. Cache state can be observed by running with `-c` and `-d 7`.
4. Now let's construct some specific workloads. Run the simulation with `-A oa1:w1:c1,oa1:r1:c1` flag. What are different possible values observed by client 1 when it reads the file `a`, when running with the random scheduler? (try different random seeds to see different outcomes)? Of all the possible schedule interleavings of the two clients' operations, how many of them lead to client 1 reading the value 1, and how many reading the value 0?
5. Now let's construct some specific schedules. When running with the `-A oa1:w1:c1,oa1:r1:c1` flag, also run with the following schedules: `-S 01`, `-S 100011`, `-S 011100`, and others of which you can think. What value will client 1 read?
6. Now run with this workload: `-A oa1:w1:c1,oa1:w1:c1`, and vary the schedules as above. What happens when you run with `-S 011100`? What about when you run with `-S 010011`? What is important in determining the final value of the file?