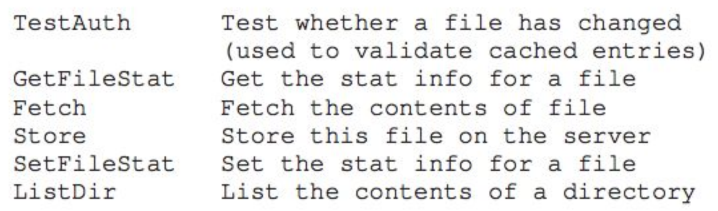
Chapter 50: The Andrew FIle System (AFS)

- Main goal: **scale**
    - how can one design a distributed file system such that a server can support as many clients as possible
    - Another goal was to have reasonable user-visible behavior
        - With NFS, cache consistency is hard to describe because it depends directly on low-level implementation details
        - With AFS, cache-consistency is simple and readily understood

50.1: AFS V1



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

-
- Basic tenets: **whole-filing caching on the local disk of the client machine accessing a file**
- Operations to open, read, and write files require no network connection because when you open() a file you receive the entire file from the server and store it on a file on your local disk
- upon a call to close() the file is flushed back to the server
- **Difference with NFS**
    - NFS uses a cache block to store, not whole files, but blocks in **client memory** instead of on the local disk
- When a client calls open(), the client side code (called Venus) sends a fetch protocol over to the server
    - The fetch protocol message will contain the entire path name of the desired file to the server (called **Vice),** which would then traverse the pathname, find the desired file, and then ship the entire file back to the client
    - Client side code would then cache the file on the local disk of the client through writing it to the local disk
    - subsequent read() and write() calls are strictly local in AFS (no communication with server necessary)
    - Once a block is accessed, it may also be cached in client memory -> AFS also uses client memory to cache copies of blocks that it has in its local disk
    - If the file has been modified (opened for writing), it flushes the new version back to the server with a Store protocol message to store the file on the server (permanent storage)

- Subsequent calls to access the file are done extremely efficiently, as if the file is unmodified, it can use the copy that it has locally on client memory -> avoids network transfer
    - only cached file contents -> directories were only kept on the server

50.2: Problems with AFS V1

- **Path-traversal costs are too high:** when performing a fetch or store protocol request, the client passes the entire pathname to the server. The server, in order to access the file, must perform a full pathname traversal, first looking at the root directory and going down the path.
- **The client issues too many TestAuth protocol messages:** much like NFS and its overabundance of the GETATTR protocol messages, AFSv1 generated a large amount of traffic to check whether a local file was valid with the TestAuth protocol message
- **Load was not balanced across servers**
- **Server used a single distinct process per client thus inducing context switching and other overheads**

50.4: AFS V2

- Introduced the notion of a callback to reduce the number of client/server interactions
    - **callback is simply a promise from the server to the client that the server will inform the client that the server will inform a client when a file that the client is caching has been modified**
- **Allowed the notion of a file identifier** similar to the file handle in NFS
    - a FID consists of a volume identifier, a file identifier, and a “uniquifier” to enable reuse of the volume and file ID’s when a file is deleted
    - Instead of sending the whole pathname, the client would walk the pathname, one piece at a time, caching the results and hopefully reducing the load on the server
    - Ex: client could fetch the contents of directory home and then store it in the local-disk cache and set up a callback on home. Then the server would set up callbacks on subsequent subdirectories and files on the local-disk cache and then eventually return a file descriptor to the calling application

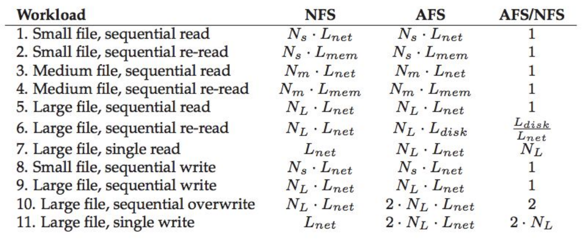
50.5: Cache Consistency

- Consider two cases: **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 -> when the updated file is closed
        - when a file is closed, the new file is flushed back to the server and thus is visible
        - the server breaks callbacks for any clients with cached copies by informing it that the callback it has on the file is no longer valid

            - subsequent calls on the new version from the server will require steps to reestablish a callback on the new version of the file
- On the same machine
    - 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)
    - Makes using a single machine behave exactly as you would expect, based upon typical UNIX semantics
- Rare case when processes on different machines are modifying a file at the same time
    - Naturally employs the last writer wins approach
    - when client calls close() last will update the entire file on the server last and thus will be "winning"
    - Difference between NFS (DISADVANTAGE)
        - on NFS, we flush out individual blocks to the server so the final file may end up being a mixture of both those files

50.6: Crash Recovery
- Crash recovery is more involved than with NFS
- Server recovery after a crash is also more complicated
    - callbacks are also kept in memory -> server reboots -> has no idea which client machine has which files
    - Upon server restart, each client of the server must realize that the server has crashed and treat all of their cache contents as suspect -> must reestablish the validity of the server
    - One must ensure that the client is aware of the server crash in a timely manner so that the client can treat the cached contents as suspect
    - We can have the server send a message saying "don't trust your cache content" to each client when it is up and running again
        - Or we can have the clients check that the server is alive periodically with a heartbeat message

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
- Client-side performance often came quite close to local performance, because in the common case, all file accesses were local



Workload	NFS	AFS	AFS/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**

-

- Comparing differences of performance
    - performance of both systems are roughly equivalent
        - when reading a file, the time to fetch the file from remote server dominates and is similar for both systems
    - Large file sequential re-read
        - Because AFS has a large local disk cache, it will access the file from where there when the file is accessed again
        - NFS only caches blocks in client memory -> NFS would have to refetch the entire file from the remote server -> AFS is faster because it doesn't have the additional network communication costs
    - Sequential writes of new files should perform similarly on both systems
    - AFS performs worse on a sequential file overwrite
        - **the client first fetches the old file in its entirely, only to subsequently overwrite it**
        - NFS will simply overwrite blocks and thus avoid the initial useless read
    - Workloads that only access a small subset of data within large files perform much better on NFS
        - the AFS protocol fetches the entire file when the file is opened while only a small read or write is performed
            - if the file is modified, the entire file is written back to the server
                - this doubles the performance impact
        - NFS, as a block-based protocol, performs I/O that is proportional to the size of the read/write

50.8: AFS: Other Improvements

- AFS provides a true global namespace to clients, thus ensuring that all files were named the same way on all client machines
    - NFS allowed different client machines mount NFS servers in any way that they please, and thus only by convention would files be named similarly across clients
- AFS 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
- AFS includes facilities for flexible user-managed access control
    - a user has a great deal of control over who exactly can access the files
    - NFS, like most UNIX file systems have much less support for this type of sharing
- Enables simpler management of servers for the administrators of the system by making some things more user-visible and easier such as caching

ACID Semantics

- In computer science, ACID (atomicity, consistency, isolation, and durability) is a set of properties of database transactions intended to guarantee validity even in the even of errors, power failures, etc.

- In the context of databases, a sequence of database operations that satisfies the ACID properties, and thus can be perceived as a single logical operation on the data

Characteristics

- Atomicity
    - requires that each transaction is “all or nothing”
    - If one part of the transaction fails, then the entire transaction fails, and the database state is left unchanged
    - Must guarantee atomicity in each and every situation, including power failures, errors, and crashes
- Consistency
    - ensures that any transaction will bring the database from one valid state to another
    - This does not guarantee correctness of the transaction in all ways the application programmer might have wanted, but merely that any programming errors cannot result in the violation of any defined rules
- Isolation
    - ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed sequentially
- Durability
    - ensures that once a transaction has been committed, it will remain so, even in the extent of power loss, crashes, or errors
    - ex: In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter)
    - Transactions must be recorded in non-volatile memory

Atomicity Failure

- The series of operations cannot be separated with only some of them being executed, which makes the series of operations “indivisible”
- Alternatively, we can say that a logical transaction may be made of, or composed of, one or more physical transactions
    - Until all component physical transactions are executed, the logical transactions will not have occurred to the effects of the database
- Ex: If our logical transaction consists of transferring money from account A to account B
    - physical transactions: Removing the amount from account A as a first Physical transaction and then as a second physical transaction, placing it in account B
    - we would not want to see the amount removed in account A before we are sure it has been transferred into account B
    - Unless and until both transactions have happened and the amount has been transferred to account B, the transfer has not, to the effects of the database occured

Consistency Failure

- Demands that the data meet all the validation rules -> in the bank account example, the validation requirement is that A+B=100
- All validation rules must be checked to ensure consistency
- If consistency is not met, it will affect atomicity as the transaction should not be complete if a certain validity condition is not met

Isolation Failure

- Assume two transactions execute at the same time, each attempting to modify the same data. One of the two must wait until the other completes in order to maintain isolation
- Consider two transactions T1 transfers 10 from A to B. T2 transfers 10 from B to A
  - T1 subtracts 10 from A
  - T1 adds 10 to B
  - T2 subtracts 10 from B
  - T2 adds 10 to A
- **Write-write failure:** consider what happens if T1 happens halfway through
  - the database eliminates T1's effects and T2 sees only valid data
  - two transactions attempt to write to the same data field -> resolve the problem through reverting to the last known good state

Durability Failure

- Consider a transaction that transfers 10 from A to B. First it removes 10 from A, then it adds 10 to B
- At this point the user is told the transaction was a success, however the changes are still queued in the disk buffer waiting to be committed to disk
- Power fails and the changes are lost. The user assumes that the changes persist

Implementation

- Processing a transaction often requires a sequence of operations that is subject to failure for a number of reasons
- For instance, the system may have no room left on its disk drive, or it may have used up its allocated CPU time
- Two popular techniques to deal with this
  - write-ahead logging and shadow paging
    - in both cases, locks must be acquired on all information to be updated, and depending on the level of isolation
- Locking vs Multiversioning
  - many databases rely upon locking to provide ACID capabilities. Locking means that the transaction marks the data that it accesses so that the DBMS knows not to allow other transactions to modify until the first transaction succeeds or fails
    - lock must always be acquired before processing data, including data that is read but not modified

        - nontrivial transactions require numerous locks that can increase the overhead of transactions
    - Alternative to locking: multiversion concurrency control
        - database provides each reading transaction the prior, unmodified version of data that is being modified by another active transaction
        - Allows readers to operate without acquiring locks -> writing transactions do not block reading transactions and vice versa