CSE 486/586: Distributed Systems

Distributed Filesystems

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Distributed Filesystems

This lecture will explore network and distributed filesystems.

We will explore:

- Sun NFS
- Andrew Filesystem (AFS)
- Coda

These are *moderately ancient* filesystems, but teach some good lessons.

Sun NFS

NFS [1] is an early network filesystem built on a client-server model.

It provides UNIX filesystem semantics for remote files.

Consistency guarantees are somewhat weaker than local filesystems.

"UNIX Filesystem Semantics"

By UNIX filesystem semantics, we mean that:

- UNIX applications need not be aware that they are accessing a remote file.
- File access uses `open()`, `read()`, `write()`, `stat()`, *etc.*
- Performance is acceptable under this model.

This entails the addition of a kernel virtual filesystem (VFS) layer capable of proxying normal file operations to a remote server.

Andrew Filesystem

The Andrew Filesystem (AFS) [2] was designed to provide better scalability than NFS.

AFS was developed at CMU.

It also provides UNIX filesystem semantics.

It uses aggressive client-side caching to reduce server load.

Its consistency guarantees are slightly weaker than NFS.

Coda

Coda is an improvement on AFS designed to provide higher availability than AFS.

Coda [3] also provides UNIX filesystem semantics.

It allows disconnected operation, where clients can both read and mutate files without access to the server.

Aggressive client-side caching with user-configurable characteristics is used to accomplish this.

Once again, consistency is weakened.

Sun NFS

NFS shares a UNIX filesystem between UNIX hosts over the network with UNIX filesystem semantics.

To accomplish this, it uses Sun RPC, a remote procedure call interface.

NFS servers are stateless.

- Local UNIX filesystem access is stateful.
- *E.g.*, `open()` creates in-kernel data structures.
- This state is stored by the NFS client.

Stateless Operation

The stateless operation of an NFS server provides scalability.

- The Number of client machines/processes does not directly effect server load.

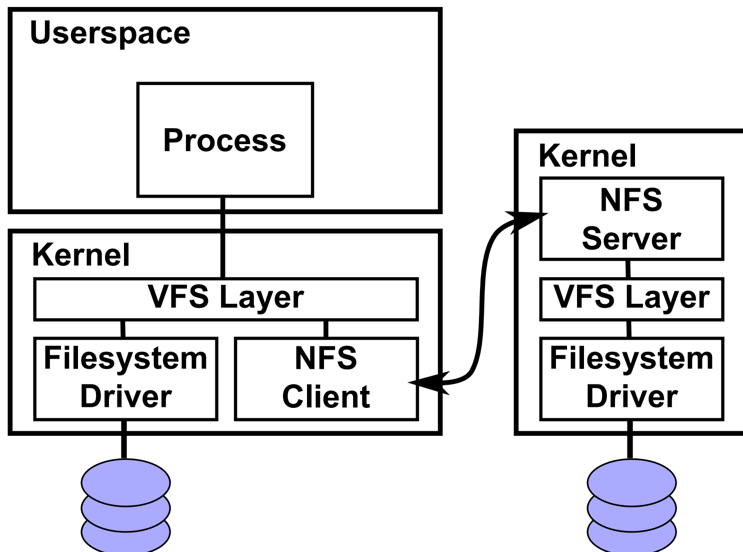
However, it also adversely effects performance.

- Certain operations (*e.g.*, permission checks) must be performed on every file access.
- Server-side caching and buffering have limited visibility into client state.

It also entails making internal structures visible to clients.

- *E.g.*, an NFS file handle contains the inode number of a referenced file.

NFS Architecture



NFS Architecture

Userspace processes are unaware of NFS.

The VFS layer redirects remote file accesses to the in-kernel NFS client

The in-kernel NFS client:

- Maintains necessary state for the userspace process
- Proxies file manipulations to the NFS server
- Performs some caching

The NFS server:

- Translates local filesystem inodes to and from NFS file handles
- Uses file handles to perform local file operations
- Returns filesystem results to the client
- Performs some caching

NFS Consistency

The weaker consistency guarantees provided by NFS are due to caching.

- Blocks are cached at the client and used to serve read requests.
- Writes are quickly pushed to the server by the client, but clients are not asynchronously notified of changes
- Clients periodically poll the server for changes
- The server caches reads and writes according to normal filesystem semantics

This caching is necessary to achieve usable performance.

However, client writes on one system may not be noticed at another system for some time.

File Distribution in NFS

Every NFS server serves a set of exports.

Each export can be mounted individually by NFS clients.

This provides semantics similar to UNIX physical disk management.

A client may mount multiple mountpoints from multiple servers.

AFS

The Andrew Filesystem improves on the scalability of NFS by reducing the number of client-server interactions.

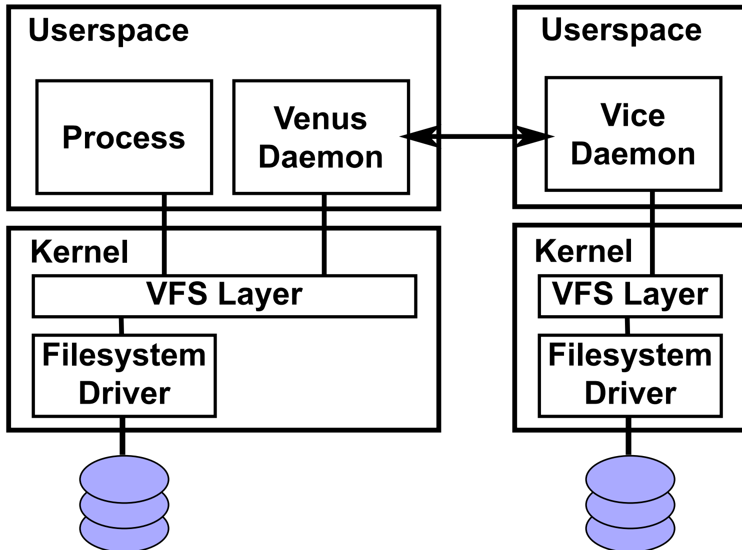
Ironically, AFS servers keep some state regarding clients.

NFS clients contact the server for most file operations.

AFS clients contact the server only to open or close a file.

- Clients cache entire files on the local filesystem.
- Reads may be served entirely from the cache.
- Writes are kept locally until the file is closed.
- Changes are flushed to the server on close.
- The server notifies clients if another host modifies an open file.

AFS Architecture



AFS Architecture

The VFS layer redirects remote file accesses to the userspace Venus daemon.

The Venus daemon:

- Maintains a local on-disk cache of open and recently-used files with whole-file contents.
- Fetches remote or changed files on open.
- Uploads locally modified files on close.

The Vice daemon:

- Maintains callback promises for open client files, so clients can be notified of changes from other hosts.
- Sends and receives whole-file data from Venus daemons.

AFS State and Caching

Unlike NFS, AFS servers retain client state.

This is to provide better consistency with aggressive caching.

- Venus daemons maintain a registry of open files with their Vice counterparts.
- Vice servers keep callback promises for open files.
- When a changed file is uploaded by a Venus client, the Vice server that receives it notifies other Venus daemons with callbacks for that file.

Thus, unlike NFS, clients get immediate notification of changed files and can provide reasonably UNIX-like semantics even with whole-file caching.

File Distribution in AFS

AFS servers serve a portion of a global namespace.

All AFS servers are part of this notional global namespace, but may not make their shares public.

AFS clients mount an AFS server cluster (or cell) on the unix path `/afs/<cell>`. *E.g.*:

- Carnegie Mellon University CS: `/afs/cs.cmu.edu`
- The University of Notre Dame: `/afs/nd.edu/`

The files within a cell may be provided by different servers.

Certain shares within a cell may be replicated read-only on multiple servers.

AFS Scalability

Because:

- Whole files are cached at clients
- Individual reads and writes do not involve servers

…AFS servers can handle many more clients and open files than NFS servers.

This is despite the fact that AFS servers track open files, while NFS servers are stateless.

Fewer, somewhat more complicated transactions are easier to handle than many small transactions in this case.

This is because most file operations are reads, and most files do not have concurrent accesses that include writes.

Coda

Coda was developed at CMU to address some AFS limitations:

- Server crashes disrupted clients significantly
- Disconnected operation was impossible
- Read-write volumes could not be replicated

Coda's architecture looks, at a glance, like AFS:

- Vice daemons on clients cache whole files
- Venus daemons on servers serve files and notifies clients of changes via callbacks

Coda Consistency

Files replicated across servers are versioned with a vector timestamp.

Sharing proceeds as in AFS when all replicas are available, and all replicas make updates together.

If the servers are partitioned, clients can optimistically write to the servers they can contact, and modified files are resolved via timestamps when everything reconnects.

Concurrent timestamps lead to conflicts, and Coda does not resolve conflicts. User intervention is required.

Disconnected Operation

A disconnected client can operate entirely from its cache.

This requires that all files the client intends to use are in the cache.

Users can configure Venus with files to keep perpetually in the local cache, and Venus will update them when connected.

When reconnecting, reconciliation takes place, with files being updated with a conflict policy based on timestamps.

Summary

NFS, AFS, and Coda all try to provide UNIX filesystem semantics:

- One file, on one disk

AFS and Coda in particular depart from this model in well-understood ways to achieve better scalability and availability (respectively).

Caching and write conflict detection are used by all three.

- NFS clients poll the server
- AFS and Coda notify clients with open files of changes to those files
- Coda uses vector timestamps to version files

References I

Required Readings

[1] Textbook. Section 12.3.

[2] Textbook. Section 12.4.

[3] Textbook. Section 18.4.3.