Distributed File Systems

File Characteristics

From Andrew File System work:

- most files are small—transfer files rather than disk blocks?
- reading more common than writing
- most access is sequential
- most files have a short lifetime—lots of applications generate temporary files (such as a compiler).
- file sharing (involving writes) is unusual—argues for client caching
- processes use few files
- files can be divided into classes—handle "system" files and "user" files differently.

Newer Influences

- wide-area networks
- peer-to-peer
- mobility
- untrusted entities

Distributed File Systems

Primarily look at three distributed file systems as we look at issues.

1. File Transfer Protocol (FTP). Motivation is to provide file sharing (not a distributed file system). 1970s.

   Connect to a remote machine and interactively send or fetch an arbitrary file. FTP deals with authentication, listing a directory contents, ascii or binary files, etc.

   Typically, a user connecting to an FTP server must specify an account and password. Often, it is convenient to set up a special account in which no password is needed. Such systems provide a service called *anonymous* FTP where userid is "anonymous" and password is typically user email address.

   Has largely been superceded by use of HTTP for file transfer.

2. Sun's Network File System (NFS). Motivated by wanting to extend a Unix file system to a distributed environment. Easy file sharing and compatability with existing systems. Mid-1980's.

   Stateless in that servers do not maintain state about clients. RPC calls supported:

   - searching for a file within a directory
   - reading a set of directory entries
   - manipulating links and directories
   - accessing file attributes
   - reading/writing file data

   Latest version of NFS (version 4) introduces some amount of state.

3. Andrew File System (AFS). Research project at CMU in 1980s. Company called Transarc, acquired by IBM. Primary motivation was to build a scalable distributed file system. Look at pictures.

Other older file systems:

1. CODA: AFS spin-off at CMU. Disconnection and fault recovery.
2. Sprite: research project at UCB in 1980's. To build a distributed Unix system.
3. Echo. Digital SRC.
4. Amoeba Bullet File Server: Tanenbaum research project.
5. xFs: serverless file system—file system distributed across multiple machines. Research project at UCB.

Distributed File System Issues

Naming

How are files named? Access independent? Is the name location independent?

- FTP. location and access dependent.
- NFS. location dependent through client mount points. Largely transparent for ordinary users, but the same remote file system could be mounted differently on different machines. Access independent. See Fig 9-3. Has automount feature for file systems to be mounted on demand. All clients could be configured to have same naming structure.
- AFS. location independent. Each client has the same look within a *cell*. Have a cell at each site. See Fig 13-15.

Migration

Can files be migrated between file server machines? What must clients be aware of?

- FTP. Sure, but end-user must be aware.
- NFS. Must change mount points on the client machines.
- AFS. On a per-volume (collection of files managed as a single unit) basis.

Directories

Are directories and files handled with the same or a different mechanism?

- FTP. Directory listing handled as remote command.
- NFS. Unix-like.
- AFS. Unix-like.

Amoeba has separate mechanism for directories and files.

Sharing Semantics

What type of file sharing semantics are supported if two processes accessing the same file?

Possibilities:

- Unix semantics – every operation on a file is instantly visible to all processes.
- session semantics – no changes are visible to other processes until the file is closed.
- immutable files – files cannot be changed (new versions must be created)

- FTP. User-level copies. No support.
- NFS. Mostly Unix semantics.
- AFS. Session semantics.

Immutable files in Amoeba.

Caching

What, if any, file caching is supported?

Possibilities:

- write-through – all changes made on client are immediately written through to server
- write-back – changes made on client are cached for some amount of time before being written back to server.
- write-on-close – one type of write-back where changes are written on close (matches session semantics).

- FTP. None. User maintains own copy (whole file)
- NFS. File attributes (inodes) and file data blocks are cached separately. Cached attributes are validated with the server on file open.

  Version 3: Uses read-ahead and delayed writes from client cache. Time-based at block level. New/changed files may not visible for 30 seconds. Neither Unix nor session semantics. Non-deterministic semantics as multiple processes can have the same file open for writing.

  Version 4: Client must flush modified file contents back to the server on close of file at client. Server can also *delegate* a file to a client so that the client can handle all requests for the file without checking with the server. However, server must now maintain state about open delegations and recall (with a callback) a delegation if the file is needed on another machine.
- AFS. File-level caching with callbacks (explain). Session semantics. Concurrent sharing is not possible.

Locking

Does the system support locking of files?

- FTP. N/A.
- NFS. Has mechanism, but external to NFS in v3. Internal to file system in version 4.
- AFS. Does support.

Replication/Reliability

Is file replication/reliability supported and how?

- FTP. No.
- NFS. minimal support in version 4.
- AFS. For read-only volumes within a cell. For example binaries and system libraries.

Scalability

Is the system scalable?

- FTP. Yes. Millions of users.
- NFS. Not so much. 10-100s
- AFS. Better than NFS, keep traffic away from file servers. 1000s.

Homogeneity

Is hardware/software homogeneity required?

- FTP. No.
- NFS. No.
- AFS. No.

File System Interface

Is the application interface compatible to Unix or is another interface used?

- FTP. Separate.
- NFS. The same.
- AFS. The same.

Security

What security and protection features are available to control access?

- FTP. Account/password authorization.
- NFS. RPC Unix authentication. Version 4 uses RPCSEC_GSS, a general security framework that can use proven security mechanisms such as Kerberos.
- AFS. Unix permissions for files, access control lists for directories. CODA has secure RPC implementation.

State/Stateless

Do file system servers maintain state about clients?

- FTP. No.
- NFS. No. In Version 4 servers maintains state about delegations and file locking.
- AFS. Yes.

AFS Design Principles

What was learned.

Think about for file systems and other large distributed systems.

- Workstations have cycles to burn. Make clients do work whenever possible.
- Cache whenever possible.
- Exploit file usage properties. Understand them. One-third of Unix files are temporary.
- Minimize system-wide knowledge and change. Do not hardwire locations.
- Trust the fewest possible entities. Do not trust workstations.
- Batch if possible to group operations.

Elephant: The File System that Never Forgets

by Santry, et al (U. British Columbia and HP) in USENIX OSDI99

Motivation that disks and storage are cheap and information is valuable.

Straightforward idea to store all (significant) versions of a file without need for user intervention.

"All user operations are reversible."

Simple, but powerful goal for the system.

A new version of a file is created each time it is written—similarities to a log-structured file system.

File versions are referenced by time and extend to directories.

Per-file and per-file-group policies for reclaiming file storage.

What Files to Keep?

Basic idea is to keep *landmark* or distinguished file versions and discard the others.

- Keep One—current situation. Good for unimportant or easily recreatable files.
- Keep All—complete history maintained.
- Keep Landmarks—how to determine
  - user-defined landmarks (similar to check-in idea in RCS) are allowed
  - heuristic to tag other versions as landmarks.

Not all files should be treated the same. For example object files and source files have different characteristics.

Google File System

From paper by Ghemawat, et al (Google), ACM SOSP03

Design of a file systems for a different environment where assumptions of a general purpose file system do not hold—*interesting to see how new assumptions lead to a different type of system.*

Differences:

1. component failures are the norm.
2. huge files (not just the occasional file)
3. append rather than overwrite is typical
4. co-design of application and file system API—*specialization*. For example can have relaxed consistency.

Architecture

Single *master* and multiple *chunkservers* as shown in Fig 1. Each is a commodity Linux server.

Files stored in fixed-size 64MB *chunks* as Linux files. Each has a 64-bit chunk handle.

By default have three replicas for each chunk.

GFS maintains metafile information.

Clients do not cache data—typically not reused. Do cache metadata.

Large chunk sizes help to minimize client interaction with master (potential bottleneck). Client can maintain persistent TCP connection with chunkserver. Reduces amount of metadata at master.

Shark: Scaling File Servers via Cooperative Caching

by Annapureddy, et al (NYU), USENIX NSDI05

Motivated by distributed computing environments where computations are made in replicated execution environments.

Lots of replication drawbacks: bandwidth needed, requires hard state at each replication, and replicated run-time environment not the same as devlp env.

Shark designed to support widely distributed applications. Can export a file system

Scalability through a location-aware cooperative cache—a p2p file system for read sharing.

At heart is centralized file system like NFS.

Design

Key ideas:

Once a client retrieves a file, it becomes a *replica proxy* for serving to other clients.

Files are stored and retrieved as chunks. Client can retrieve chunks from multiple locations.

A token is assigned for the whole file and for each chunk.

Use Rabin fingerprint algorithm to preserve data commonality in chunks—idea is that different versions of a file have many chunks in common.

File Consistency

Uses leases and whole-file caching—ala AFS.

Default lease of 5min with callbacks.

Must refetch entire file if modified—but may not have to retrieve all chunks and can do so from client proxies.

Summary

Key ideas:

- centralized administration
- shared reads through cooperative caching
- "smart" chunking
- parallel chunk download
- distributed index using tokens