Distributed File Systems

Accessing files

FTP, telnet:
- Explicit access
- User-directed connection to access remote resources

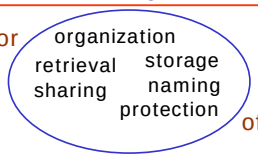
We want more transparency
- Allow user to access remote resources just as local ones

Focus on file system for now

NAS: Network Attached Storage

Operating System: File System

Responsible for



of files

File directory services
bind file name to internal handle
(inode, FAT index)

File system **controls access** to data
Low-level operations:
buffering, issuing disk I/O

Distributed file system goals

- **Access transparency**
  - Clients unaware files are remote
- **Location transparency**
  - Consistent name space (local and remote)
- **Concurrency transparency**
  - Modifications are coherent
- **Failure transparency**
  - Client and client programs should operate correctly after server failure
- **Heterogeneity**
  - File service should be provided across different hardware and software platforms

Distributed file system goals

- **Scalability**
  - Scale from a few machines to many (tens of thousands?)
- **Replication transparency**
  - Clients unaware of replication
  - Coherence maintained
- **Migration transparency**
  - Files should be able to move around without clients' knowledge
- **Fine grained distribution of data**
  - Locate objects near processes that use them

Terms

- **File service**
  - Specification of what the file system offers to clients
- **File**
  - name, data, attributes
- **Immutable file**
  - Cannot be changed once created
    - Easy to cache and replicate
- **Protection**
  - Capabilities
  - Access control lists

File service types

Upload/Download model

- *Read file:* copy file from server to client
- *Write file:* copy file from client to server

Advantage
- **Simple**

Problems
- **Wasteful**: what if client needs small piece?
- **Problematic**: what if client doesn't have enough space?
- **Consistency**: what if others need to modify the same file?

File service types

Remote access model

File service provides functional interface:
- *create, delete, read bytes, write bytes, etc…*

Advantages:
- Client gets only what's needed
- Server can manage coherent view of file system

Problem:
- Possible server and network congestion
  - Servers are accessed for duration of file access
  - Same data may be requested repeatedly

File server

File Directory Service

- Maps textual names for file to internal locations that can be used by file service

File service

- Provides file access interface to clients

Client module (driver)

- Client side interface for file and directory service
- if done right, helps provide access transparency
  - e.g. under vnode layer

Random NAS Boxes

Semantics of file sharing

Sequential semantics

Read returns result of last write

Easily achieved *if*
- Only one server
- Clients do not cache data

BUT
- Performance problems if no cache
  - Obsolete data
- We can ***write-through***
  - Must notify clients holding copies
  - Requires extra state, generates extra traffic

Session semantics

- Relax the rules
- Changes to an open file are initially visible only to the process (or machine) that modified it.
- Last process to modify the file wins.

Other solutions

Make files **immutable**

- Aids in replication
- Does not help with detecting modification

Or…

Use **atomic transactions**

- Each file access is an atomic transaction
- If multiple transactions start concurrently
  - Resulting modification is serial

File usage patterns

- We can't have the best of all worlds
- Where to compromise?
  - Semantics vs. efficiency
  - Efficiency = client performance, network traffic, server load
- Understand how files are used
- 1981 study by Satyanarayanan

File usage

Most files are <10 Kbytes

- (2005: average size of 385,341 files on my Mac =197 KB)
- (files accessed within 30 days:
  147,398 files. average size=56.95 KB)
- Feasible to transfer entire files (simpler)
- Still have to support long files

Most files have short lifetimes

- Perhaps keep them local

Few files are shared

- Overstated problem
- Session semantics will cause no problem most of the time

System design issues

Location transparency

Is the name of the server known to the client?

//server1/dir/file

- Server can move without client caring
  … if the name stays the same.
- If file moves to server2 … we have problems!

Location independence

- Files can be moved without changing the pathname
  //archive/paul

Where do you find the remote files?

Should all machines have the exact same view of the directory hierarchy?

e.g., global root directory?

```
//server/path
```

or forced "remote directories":

```
/remote/server/path
```

or….

Should each machine have its own hierarchy with remote resources located as needed?

```
/usr/local/games
```


How do you access them?

- Access remote files as local files
- Remote FS name space should be syntactically consistent with local name space

1. redefine the way all files are named and provide a syntax for specifying remote files
   - e.g. //server/dir/file
   - Can cause legacy applications to fail
2. use a file system *mounting* mechanism
   - Overlay portions of another FS name space over local name space

Name resolution: how to handle ..

Parse

(a) component at a time

versus

(b) entire path at once

(b) is more efficient but…

- Remote server may access and reveal more if its file system than it wants
- Other components cannot be mounted underneath remote tree

Perhaps use *(a)* and cache bindings

Stateful or stateless design?

Stateful

- Server maintains client-specific state

- Shorter requests
- Better performance in processing requests
- Cache coherence is possible
  - Server can know who's accessing what
- File locking is possible

Stateful or stateless design?

Stateless

- Server maintains *no* information on client accesses

- Each request must identify file and offsets
- Server can crash and recover
  - No state to lose
- Client can crash and recover
- No open/close needed
  - They only establish state
- No server space used for state
  - Don't worry about supporting many clients
- Problems if file is deleted on server
- File locking not possible

Caching

Hide latency to improve performance for repeated accesses

Four places

- Server's disk
- Server's buffer cache
- Client's buffer cache
- Client's disk

WARNING: cache consistency problems

Approaches to caching

- **Write-through**
  - What if another client reads its cached copy?
  - All accesses will require checking with server
  - Or Server maintains state and sends invalidations
- **Delayed writes**
  - Data can be buffered locally (consistency suffers)
  - Remote files updated periodically
  - One bulk wire is more efficient than lots of little writes
  - Problem: semantics become ambiguous

Approaches to caching

- **Write on close**
  - Admit that we have session semantics
- **Centralized control**
  - Keep track of who has what open on each node
  - Stateful file system with signaling traffic

The End.