Distributed File System

DISTRIBUTED FILE SYSTEMS

Clients, servers, and storage are dispersed across machines. Configuration and implementation may vary –

a) Servers may run on dedicated machines, OR
b) Servers and clients can be on the same machines.
c) The OS itself can be distributed (with the file system a part of that distribution).
d) A distribution layer can be interposed between a conventional OS and the file system.

Clients should view a DFS the same way they would a centralized FS; the distribution is hidden at a lower level.

Performance is concerned with throughput and response time.

Goals

1. **Network transparency**: users do not have to aware the location of files to access them
   - **location transparency**: the name of a file does not reveal any kind of the file's physical storage location.
     - /server1/dir1/dir2/X
     - server1 can be moved anywhere
   - **location independence**: the name of a file does not need to be changed when the file's physical storage location changes.
     - The above file X cannot moved to server2 if server1 is full and server2 is no so full.
2. **High availability**: system failures or scheduled activities such as backups, addition of nodes

Architecture

- Computation model
  - file servers -- machines dedicated to storing files and performing storage and retrieval operations (for high performance)
  - clients -- machines used for computational activities may have a local disk for caching remote files
- Two most important services
  - name server -- maps user specified names to stored objects, files and directories
  - cache manager -- to reduce network delay, disk delay problem: inconsistency
- Typical data access actions
  - open, close, read, write, etc.

Data access in a distributed system

Client side

- Client request to access data
- Check client cache
  - Data present Return to client
  - Else Check local disk
    - Data present Return to client
    - Else Send request to file server through n/w

Server side

- Check server cache
  - Data present ; load data to client cache through n/w; return to client.
  - Else Issue disk read; Local server cache; load data to client cache through n/w; return to client.

Distributed File system design

File service vs. file server

- **File service interface**: the specification of what the file system offers to its clients.
- Implemented by a user/kernel process called file server
- **File server**: a process that runs on some machine and helps implement the file service.
- A system may have one or several file servers running at the same time

Remote Files

- **What is a file?**
  - Uninterpreted sequence of bytes
  - Can be structured as a sequence of records
- **Files can have attributes**
  - Owner, size, creation date and access permissions
- **File model**
  - Files can be modified or
  - Immutable files
- **File Protection**
  - Capability
  - Access control list
- **File Service Model**
  - **upload/download model**
  - files move between server and clients, few operations (read file & write file), simple, requires storage at client, good if whole file is accessed
  - **remote access model**
  - files stay at server, reach interface for many operations, less space at client, efficient for small accesses

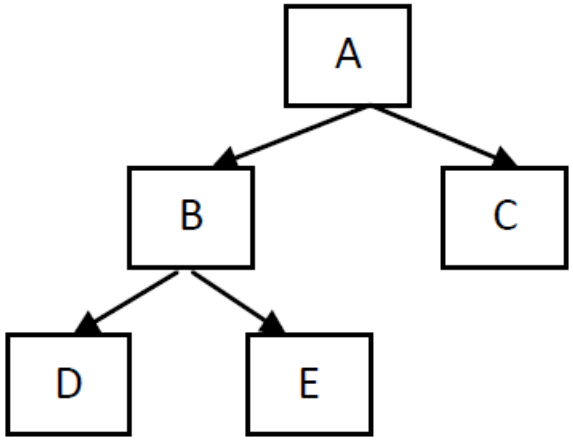
Directory Service

The directory service

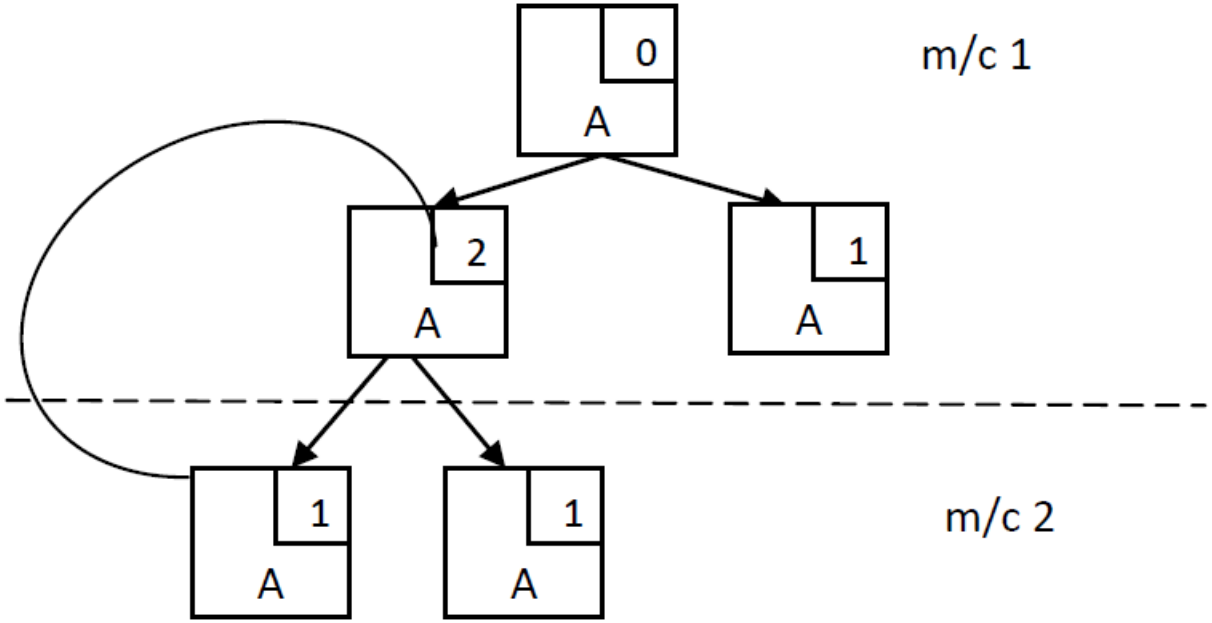
- creating and deleting directories
- naming and renaming files
- moving files

Clients can have the same view (global root directory)
or different views of the file system (remote mounting)

Hierarchical file system
Directory Tree



Directory Graph



Naming and Name Resolution

- a name space -- collection of names
- name resolution -- mapping a name to an object
  - same or different view of a directory hierarchy
- 3 traditional ways to name files in a distributed environment
  - concatenate the host name to the names of files stored on that host: system-wide uniqueness guaranteed, simple to located a file; however, not network transparent, not location independent, e.g., /machine/usr/foo
  - mount remote directories onto local directories: once mounted, files can be referenced in a location-transparent manner
  - provide a single global directory: requires a unique file name for every file, location independent, cannot encompass heterogeneous environments and wide geographical areas

Two-Level Naming

- Symbolic name (external), e.g. prog.c; binary name (internal), e.g. local i-node number as in Unix
- Directories provide the translation from symbolic to binary names
- Binary name format
  - i-node: no cross references among servers
  - (server, i-node): a directory in one server can refer to a file on a different server
  - Capability specifying address of server, number of file, access permissions, etc
  - {binary_name+}: binary names refer to the original file and all of its backups

File Sharing Semantics

UNIX semantics:

- Total ordering of R/W events
- Value read is the value stored by last write
- Writes to an open file are visible immediately to others that have this file opened at the same time.
- Easy to achieve in a non-distributed system ; In a distributed system with one server and multiple clients with no caching at client.

Session semantics:

- Writes to an open file by a user is not visible immediately by other users that have files opened already. Once a file is closed, the changes made by it are visible by sessions started later.
- Writes are guaranteed to become visible only when the file is close
- Allow caching at client with lazy updating -> better performance
- If two or more clients simultaneously write: one file (last one or non-deterministically) replaces the other

- Immutable files:

create and read file operations (no write) - i.e. a sharable file cannot be modified.
File names cannot be reused and its contents may not be altered.
Simple to implement.

Writing a file means to create a new one and enter it into the directory replacing the previous one with the same name: atomic operations

Collision in writing: last copy or nondeterministically

Transaction semantics:

- mutual exclusion on file accesses; either all file operations are completed or none is. Good for banking systems

- All changes have all-or-nothing property. W1,R1,R2,W2 not allowed where P1 = W1;W2 and P2 = R1;R2

Distributed File system Implementation

- System Structure
  - Clients and servers on different machines?
    - Combine File and directory services
    - Keep them separate
  - Lookups
    - Iterative lookup
    - Automatic lookup

Stateless vs. Stateful

Stateless Server

- requests are self-contained
- better fault tolerance
- open/close at client (fewer msgs)
- no space reserved for tables
- thus, no limit of open files
- no problem if client crashes

Stateful Servers

- shorter messages
- better performance (info in memory until close)
- open/close at server
- file locking possible
- read ahead possible

Caching

Four places to store files

- server's disk: slow performance
  - eliminates coherence problem
- server caching: in main memory
  - cache management issue, how much to cache, replacement strategy
  - still slow due to network delay
  - Used in high-performance web-search engine servers

– Client caching in main memory
  - can be used by diskless workstation
  - faster to access from main memory than disk
  - compete with the virtual memory system for physical memory space
  - avoids disk access but still network access

Three Options

- inside each process address space: no sharing at client
- in the kernel: kernel involvement on hits
- in a separate user-level cache manager: flexible and efficient if paging can be controlled from user-level

- client-cache on a local disk
  - large files can be cached
  - the virtual memory management is simpler
  - a workstation can function even when it is disconnected from the network

Update algorithms for client caching

- write-through:
  - all writes are carried out immediately
  - writes sent to the server as soon as they are performed at the client -> high traffic, requires cache managers to check (modification time) with server before can provide cached content to any client
  - Reliable: little information is lost in the event of a client crash
  - Slow: cache not that useful
- delayed-write:
  - delays writing at the server
  - coalesces multiple writes; better performance but ambiguous semantics
  - possible to perform many writes to a block in the cache before it is written
  - if data is written and then deleted immediately, data need not be written at all (20-30 % of new data is deleted with 30 secs)

- write-on-close:
  - delay writing until the file is closed at the client
  - Implements session semantics
  - if file is open for short duration, works fine
  - if file is open for long, susceptible to losing data in the event of client crash
- Central control:
  - file server keeps a directory of open/cached files at clients -> Unix semantics, but problems with robustness and scalability; problem also with invalidation messages because clients did not solicit them

Cache Coherence

- How to maintain consistency between locally cached data with the master data when the data has been modified by another client?
  1. Client-initiated approach -- check validity on every access: too much overhead first access to a file (e.g., file open) every fixed time interval
  2. Server-initiated approach -- server records, for each client, the (parts of) files it caches. After the server detects a potential inconsistency, it reacts.
  3. Not allow caching when concurrent-write sharing occurs. Allow many readers. If a client opens for writing, inform all the clients to purge their cached data.

- Potential inconsistency:
  - In session semantics, a client closes a modified file.
  - In UNIX semantics, the server must be notified whenever a file is opened and the intended mode (read or write mode) must be indicated for every open.
  - Disable cache when a file is opened in conflicting modes.

Replication

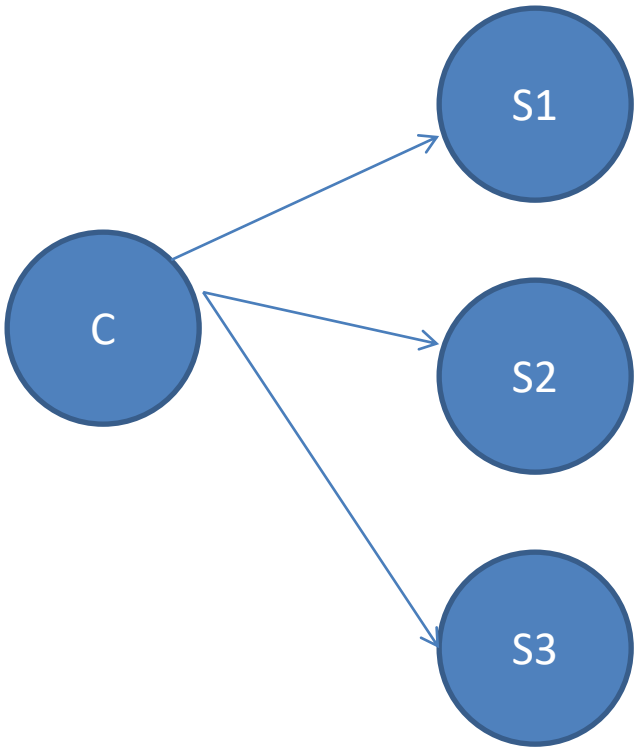
File Replication

- Multiple copies are maintained, each copy on a separate file server
- Reasons:

  To improve reliability; availability and performance.

- Replication transparency
  - explicit file replication: programmer controls replication
  - lazy file replication: copies made by the server in background
  - use group communication: all copies made at the same time in the foreground

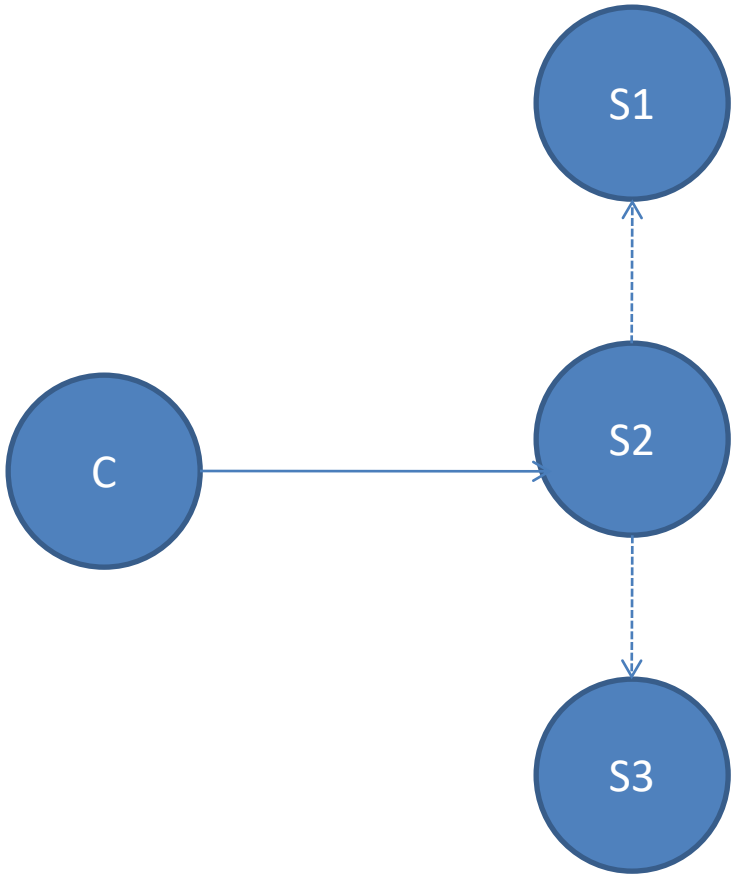
Explicit File Replication



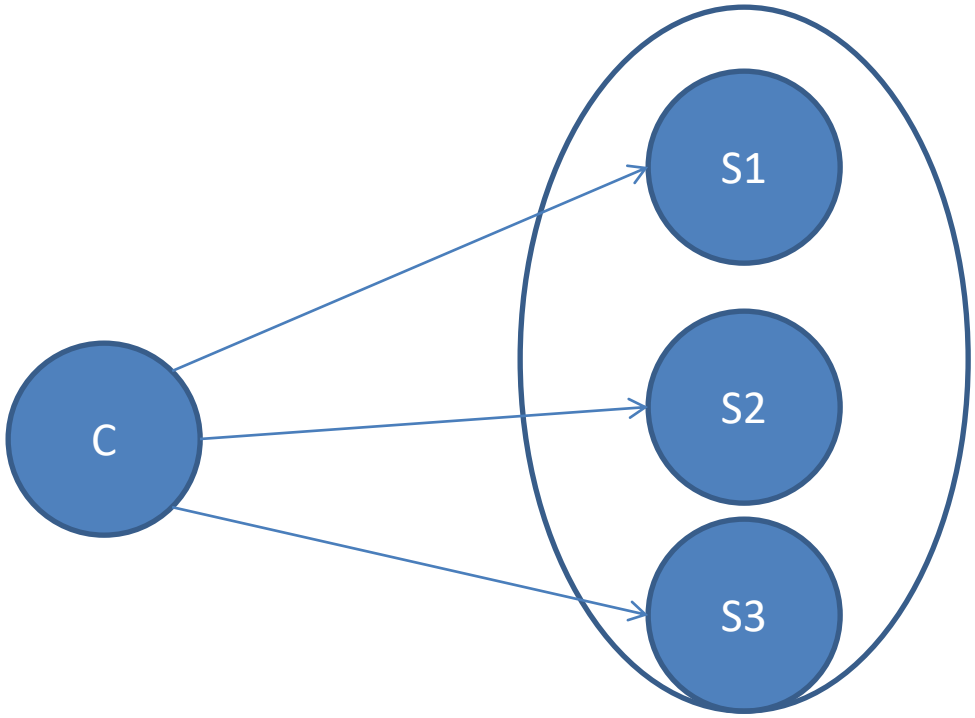
File : 1.14 : 2.16 : 3.19

Pl : 1.2.1 : 2.43 : 3.41

Lazy replication



File replication using Group



Update protocols

- Primary Copy Replication
- Voting
- Voting with ghosts

Modifying Replicas: Voting Protocol

•Updating all replicas using a coordinator works but is not robust (if coordinator is down, no updates can be performed) => Voting: updates (and reads) can be performed if some specified # of servers agree.

•Voting Protocol:

- A version # (incremented at write) is associated with each file
- To perform a read, a client has to assemble a read quorum of Nr servers; similarly, a write quorum of Nw servers for a write
- If Nr + Nw > N, then any read quorum will contain at least one most recently updated file version
- For reading, client contacts Nr active servers and chooses the file with largest version #
- For writing, client contacts Nw active servers asking them to write. Succeeds if they all say yes.

Voting Protocol with Ghosts

- Nr is usually small (reads are frequent), but Nw is usually close to N (want to make sure all replicas are updated). Problem with achieving a write quorum in the presence of server failures
- Voting with ghosts: allows to establish a write quorum when several servers are down by temporarily creating dummy (ghost) servers (at least one must be real)
- Ghost servers are not permitted in a read quorum (they don't have any files)
- When server comes back it must restore its copy first by obtaining a read quorum