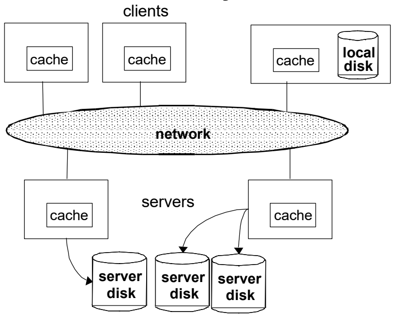
Distributed File Systems

- definition, main concepts, design goals
- semantics of file sharing
  - unix
  - session
- file access and data cashing
  - cash location
  - cash modification
  - cash validation

Distributed file systems



- *Distributed file system* is a part of distributed system that provides a user with a unified view of the files on the network.

DFS - main notions

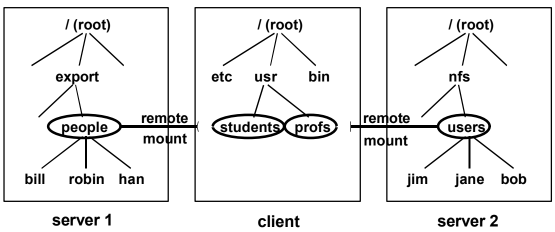
- File service — specification of the file system interface as seen by the clients
- File server — a process running on some machine which helps implement the file service by supplying files
- In principle, files in a distributed file system can be stored at any machine
  - However, a typical distributed environment has a few dedicated machines called file servers that store all the files
- A machine that holds the shared files is called a server, a machine that accesses the files is called a client.

Goals of DFS design

- Goals of a distributed file system
  - transparency
    - structure - clients should not be aware of multiple servers, replicas and cashes in use
    - access - remote and local files should be accessed the same way
    - name - the name of the file should not differ on different parts of DFS
  - user mobility/file mobility - users should be able to access the DFS in a uniform manner from different location, should be able to move files around in DFS
  - simplicity/ease of use - should be similar in use to a centralized file system
  - Availability / robustness — file service should be maintained even in the presence of partial system failures
  - performance/scalability— should overcome bottlenecks of a centralized file system, should scale well

Mounting mechanism for transparency

- transparent name space can be built using Unix *mounting* mechanism
  - file systems from servers are attached (mounted) as directories in local file system
  - the points of attachment are called mount points



File sharing semantics

- Unix semantics
  - description:
    - enforces an absolute time ordering on all operations
    - every read operation on file sees the effects of all previous write operations on that file
  - can be implemented on a single-server DFS
  - easiest to use
- session semantics
  - session - series of file accesses made between open and close operations
  - changes made to the file are visible only to client process (possibly to processes on the same client)
  - the changes are visible to the sessions that open after the session closes

File access models

- Accessing remote files:
  - remote service model - client submits requests to server, all processing is done on server, file never moves from server
    - server is bottleneck
    - excessive network communication possible
  - data-caching model
    - File service provides:
      - open — transfer entire file to client
      - close — transfer entire file to server
    - Client works on file locally (in memory or on disk)
      - Simple, efficient if working on entire file
      - Must move entire file
      - Needs local disk space

Remote file access and sharing

- Once the user specifies a remote file, the OS can do the access either:
  - Remotely on the server machine, and then return the results (RPC model), or
  - Can transfer the file (or part of the file) to the requesting host, and perform local accesses, or
  - Instead of doing the transfer for each user request, the OS can cache files, and use that cache to reduce the latency for data access (and thus increase performance)
- Issues
  - Where and when is data cached?
  - Cache consistency:
    - What happens when the user modifies the file? Does each cached copy change? Does the original file change?
    - Is the cached copy is out of date?

Cache location

- No caching — all files on server's disk
  - Simple, no local storage needed
  - Expensive transfers
- Cache files in server's memory
  - Easy, transparent to clients
  - Still involves a network access
- Cache files on client's local disk
  - Plenty of space, reliable
  - Faster than network, slower than memory
- Cache files in client's memory
  - The usual solution (either in each process's address space, or in the kernel)
  - Fast, permits diskless workstations
  - Data may be lost in a crash

Cache modification policy

- Cache modification (writing) policy decides when a modified (dirty) cache block should be flushed to the server
- Write-through — immediately flush the new value to server (& keep in cache)
  - No problems with consistency
  - Maximum reliability during crashes
  - Doesn't take advantage of caching during writes (only during reads)
- Write-back (delayed-write) — flush the new value to server after some delay
  - Fast — write need only hit the cache before the process continues
  - Can reduce disk writes since the process may repeatedly write the same location
  - Unreliable — if machine crashes, unwritten data is lost

Cache modification policy (cont.)

- Variations on write-back (when are the new values flushed to the server?)
  - Write-on-close — flush new value to the server only when the file is closed
    - Can reduce disk writes, particularly when the file is open for a long time
    - Unreliable — if machine crashes, unwritten data is lost
    - May make the process wait on the file close
  - Write-periodically — flush new value to the server at periodic intervals (maybe 30 seconds)
    - Can only lose writes in last period

Cache validation

- A client must decide whether or not a locally cached copy of data is consistent with the master copy
- Client-initiated validation:
  - Client initiates validity checks
  - Client contacts the server and asks if its copy is consistent with the server's copy
    - At every access, or
    - After a given interval, or
    - Only on file open
  - Server could enforce single-writer, multiple-reader semantics, but to do so
    - It would have to store client state (expensive)
    - Clients would have to specify access type (read / write) on open
  - High frequency of validity checks may mitigate the benefits of caching

Cache validation (cont.)

- Server-initiated validation:
  - Server records the parts of each file that each client caches
  - Server detects potential conflicts if two or more clients cache the same file
  - Handling conflicts:
    - Session semantics — writes are only visible in sessions starting later (not to processes which have file open now)
      - When a client closes a file that it has modified, the server notifies the other clients that their cached copy is invalid, and they should discard it
        - If another client has the file open, discard it when its session is over
    - UNIX semantics — writes are immediately visible to others
      - Clients specify the type of access they want when they open a file, so if two clients want to write the same file for writing, that file is not cached
  - Significant overhead at the server

Stateful vs. stateless

- Stateful server — server maintains state information for each client for each file
  - Connection-oriented (open file, read / write file, close file)
  - Enables server optimizations like read-ahead (prefetching) and file locking
  - Difficult to recover state after a crash
- Stateless server — server does not maintain state information for each client
  - Each request is self-contained (file, position, access)
    - Connectionless (open and close are implied)
  - If server crashes, client can simply keep retransmitting requests until it recovers
  - No server optimizations like above
  - File operations must be idempotent