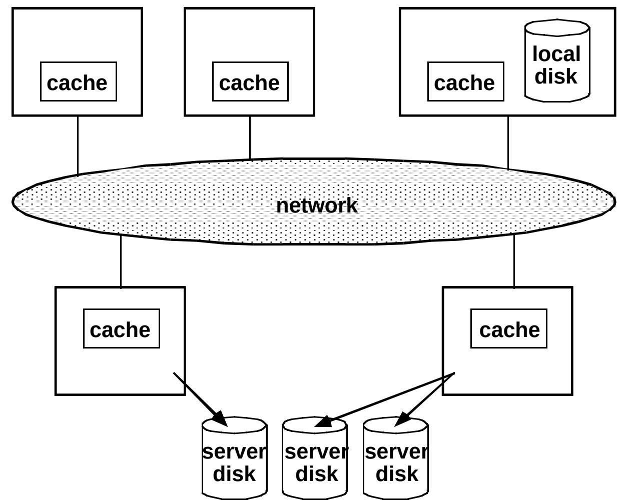
Distributed File Systems

- *Distributed file system* — a distributed implementation of a file system
  - *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
- Goals of a distributed file system
  - *Network transparency*
    - Provide same operations for accessing remote and local files
    - Ideally, clients should not have to know the location of files to access them
  - *Availability / robustness* — file service should be maintained even in the presence of partial system failures
  - *Performance* — should overcome bottlenecks of a centralized file system

Distributed File Systems (cont.)



- 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

Distributed File System Services — File Service Interface

- Need operations for creating and deleting, opening and closing, and reading and writing, files
- Upload / download model
  - File service provides:
    - Read — transfer entire file to client
    - Write — 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 access model
  - File service provides usual file operations
  - File stays on server

Distributed Naming Structures

- Need operations for name translation, support for multilevel directories and links
  - *Location transparency* — the name of the file does not reveal the physical storage location
    - True for many naming schemes
  - *Location independence* — the name of the file need not change if the file's storage location changes
    - False for most naming schemes
- Absolute names
  - Names of form: *machine* : *pathname*
  - Used by:
    - Old UNIX distributed file systems
    - Current web browsers (e.g., Netscape)
  - ✔ User can use same tools and file operations for local and remote access
  - ✘ Not location transparent or independent

Distributed Naming Structures (cont.)

- Mount remote directories onto local directories (possibly on demand)
  - Client-maintained mount information:
    - Used by UNIX and NFS — Sun's Network File System
    - Client maintains:
      - A set of local names for remote locations
      - A *mount table* (/etc/fstab) that specifies a:
        - » < remote machine name : pathname >
        - » and < local pathname >
    - At boot time, the local name is bound to the remote name
      - Afterwards, users refer to local pathname as if it were local, and the distributed OS takes care of the mapping
      - Location transparent and independent after the mount operation, but not before
  - Server-maintained mount information:
    - If files are moved to a different server, mount information need only be updated at servers

Distributed Naming Structures (cont.)

- Single name space for remote and local directories
  - Names of form: */.../machine/fs/pathname*
  - Used by:
    - CMU's Andrew, now in OSF's Distributed Computing Environment (DCE)
    - Berkeley's Sprite
  - File names are always the same, whether file is remote or local
  - As clients access a file, the server sends a copy to the client's workstation, and the workstation caches the file
    - In Andrew, local disks are used
    - In Sprite, large memories are used, and workstations are diskless
    - More details on these two next time…
  - Location independent, not location transparent

Remote File Access and Caching

- 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 (<u>when</u> 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 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
  - Concurrency control for 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