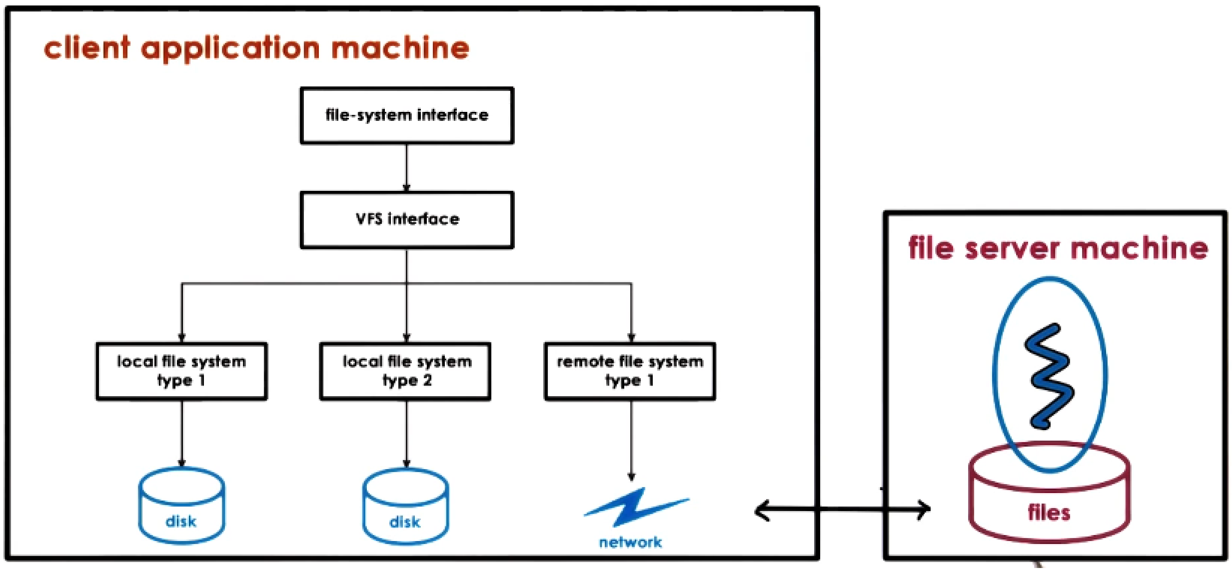
# Distributed File Systems

## Distributed File Systems Overview

1. Accessed via well-defined interface
    - Virtual File System (VFS)
2. Focus on consistent state
    - Tracking state, file updates, cache coherence, ...
3. Mixed distribution models possible
    - Replicated vs partitioned, peer-like systems, ...

---




Distributed File System Example

---

## DFS Models

1. Client/server on different machines
2. File server distributed on multiple machines
    - Replicated (each server: all files)
        - Better redundancy
    - Partitioned (each server: part of files)
        - More scalable
    - Both (files partitioned; each partition replicated)
3. Files stored on and served from all machines (peers)
    - Blurred distinction between clients and servers

## Remote File Service: Extremes

1. Upload/Download: Client downloads entire file, performs local accesses, then uploads file to server
    - Examples: FTP, SVN, ...
    - Pros:
        - Local reads/writes at client are fast
    - Cons:
        - Entire file download/upload even for small accesses
        - Server gives up control
2. True Remote File Access: Every access to remote file, nothing done locally

- Pros:
  - File accesses centralized, easy to reason about consistency
- Cons:
  - Every file operation pays network cost
  - Limits server scalability

## Compromise for Remote File Service

1. Allow clients to store parts of files locally (blocks)
   - Pro: Low latency on file operations
   - Pro: Server load reduced -> is more scalable
2. Force clients to interact with server (frequently)
   - Pro: Server has insights into what clients are doing
   - Pro: Server has control over which accesses can be permitted, making it easier to maintain consistency
   - Con: However, this makes the server more complex and requires different file sharing semantics

## Stateless vs Stateful File Server

1. Stateless: Keeps no state
   - Works for extreme models, but cannot support "practical" model
   - Pro: No resources are used on server side (CPU/memory)
   - Pro: On failure, just restart
   - Con: Cannot support caching and consistency management
   - Con: Every request is self-contained -> more bits are transferred
2. Stateful: Keeps client state
   - Needed for "practical" model to track what is cached/accessed
   - Pro: Can support locking, caching, incremental operations
   - Con: On failure, need checkpointing and recovery mechanisms
   - Con: Overheads to maintain state and consistency -> depends on caching mechanism and protocol

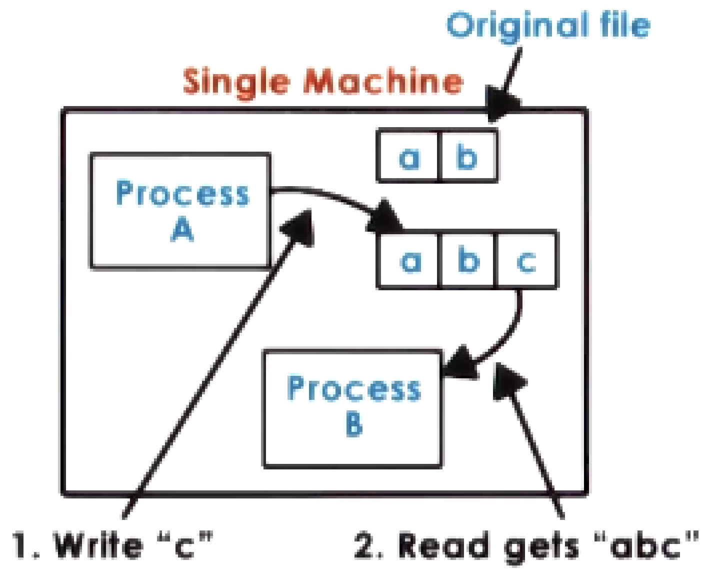
## Caching State in a DFS

1. Clients maintain portion of state locally (file blocks)
2. Clients perform operations on cached state locally (open/read/write)
   - Requires coherence mechanisms
   - How and when does a client learn that a file has been updated by a different client? Similar problem to shared multiprocessors
   - Details depend on file sharing semantics
3. How?
   - SMP: write-update/write-invalidate
   - DFS: client/server-driven
4. When?
   - SMP: On write
   - DFS: On demand, periodically, on open, . . .

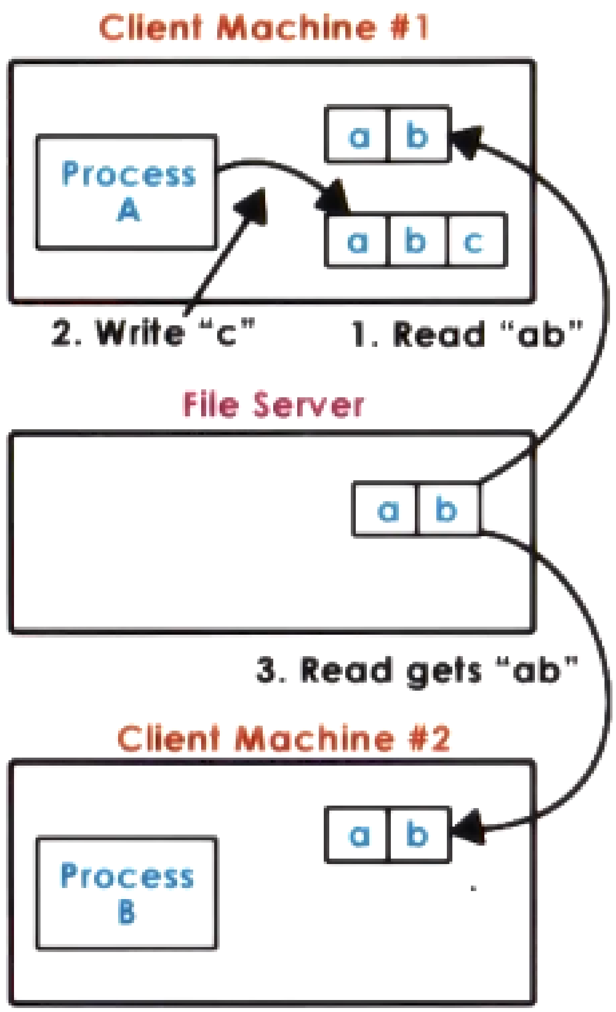
## File Sharing Semantics on a DFS

1. For a single machine, it is guaranteed that if process A updates a file, process B will see the changes immediately
2. There is no such guarantee on a distributed system (some latency in sending the "update" message)
3. Unix semantics: Every write visible immediately
4. Session semantics: Write-back on close(), update on open()
   - Session is period between open() and close()
   - Easy to reason, but may be insufficient (concurrent updating)

5. Periodic updates
    - Client writes-back periodically -> clients have a “lease” on cached data (not exclusive necessarily)
    - Server invalidates periodically -> provides bounds on “inconsistency”
    - Augment with flush()/sync() API
6. Immutable files: Never modify, new files created (Instagram)
7. Transactions: All changes atomic




File Accesses on a Single Machine




File Accesses on Multiple Machines

## File vs Directory Service

1. Access patterns (optimize for common case)
   - Sharing frequency
   - Write frequency
   - Importance of consistent view
2. Two types of files: Regular files vs directories
   - Different access patterns, so commonly choose different policies
     - Session-semantics for files, UNIX for directories
     - Less frequent write-back for files than directories

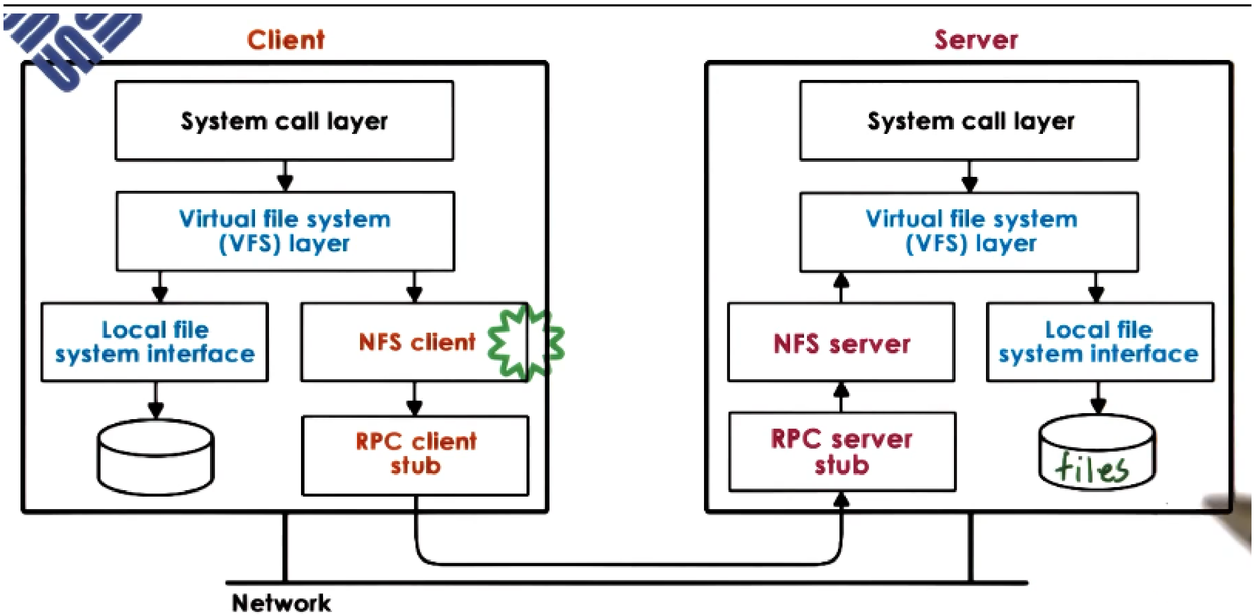
## Replication vs Partitioning

1. Replication: Each machine holds all files
   - Pro: Load balancing, availability, fault tolerance
   - Con: Writes become more complex
     - Synchronously write to all

    - Or, write to one, then propagate to others
  - Con: Replicas must be reconciled (e.g., voting)
2. Partitioning: Each machine has a subset of files
   - Pro: Availability vs single server DFS
   - Pro: Scalability with file system size
   - Pro: Single file writes are simpler
   - Con: On failure, lose portion of data
   - Con: Load balancing more difficult; if not balanced, then hot-spots are a possibility
3. Can combine both techniques - replicate each partition

## Network File System (NFS) Design

1. Clients access files as normal; NFS determines if request can be handled locally or if it must be passed to the remote filesystem
2. Client/server interaction uses RPC




NFS Design

## NFS Versions

1. Created in the 80s, currently on NFSv3 and NFSv4
2. NFSv3 == stateless, NFSv4 == stateful
3. NFSv4 is able to suppport client caching, file locking, . . .
4. Caching semantics:
   - Session-based for files not accessed concurrently
   - Periodic updates: Default: 3 seconds for files, 30 seconds for directories
   - NFSv4: Delegation to client for a period of time (avoids "update checks")
5. Locking:
   - Lease-based for some time period
     - Must release lock or explicitly extend
     - Helps deal with client failure when a lock is possessed
   - NFSv4: Also "share reservation" - reader/writer lock
6. NFS is not purely session or periodic, but a hybrid of each

## Sprite DFS

1. Research DFS, not production like NFS (people did use it)
2. Great value in the explanation of the design process
3. Used trace data on usage/file access patterns to analyze DFS design requirements and justify decisions

## Sprite DFS Access Pattern Analysis

1. 33% of all file accesses are writes
2. 75% of files are open less than 0.5 seconds
3. 90% of files are open less than 10 seconds
4. 20-30% of new data is deleted within 30 seconds
5. 50% of new data deleted within 5 minutes
6. File sharing is rare (multiple clients concurrently accessing file)
7. Takeaways:
    - Caching OK, but write-through not sufficient
    - Session semantics will have too high overhead
    - Write-back on close not really necessary
    - No need to optimize for concurrent access, but must support it

## Sprite DFS from Analysis to Design

1. Sprite supports caching using a write-back policy
    - Every 30 seconds, client will write-back blocks that have NOT been modified for the last 30 seconds
        - Blocks more recently modified will (likely) continue being modified
    - When another client opens file, server will get all dirty blocks
    - Open goes to the server; directories not cached on client
    - Disable caching on "concurrent write"
2. Sprite sharing semantics
    - Sequential write sharing: Caching and sequential semantics
    - Concurrent write sharing == no caching

## File Access Operations in Sprite

1. n readers, 1 writer
    - All open() operations go through the server
    - All clients cache blocks
    - Writer keeps timestamps for each modified block
    - Client metadata (per file)
        - Cache [Y/N]
        - Cached blocks
        - Timer for each dirty block
        - Version
    - Server metadata (per file)
        - Readers
        - Writers
        - Version
        - Cacheable [Y/N]
2. n readers, 2 sequential writers (sequential sharing)
    - Server contacts last writer for dirty blocks
    - If w1 has closed update version, w2 can now cache the file
3. n readers, w3 is a concurrent writer (concurrent sharing)
    - Server contacts last writer for dirty blocks
    - Since w2 hasn't closed the file, disable caching
        - All file accesses must go to the server

4. Dynamically enabling/disabling caching is a unique feature of Sprite