Operating System Principles:
Accessing Remote Data
CS 111
Operating Systems

Outline

- Data on other machines
- Remote file access architectures
- Challenges in remote data access
  - Security
  - Reliability and availability
  - Performance
  - Scalability

Remote Data: Goals and Challenges

- Sometimes the data we want isn't on our machine
  - A file
  - A database
  - A web page
- We'd like to be able to access it, anyway
- How do we provide access to remote data?

Basic Goals

- Transparency
  - Indistinguishable from local files for all uses
  - All clients see all files from anywhere
- Performance
  - Per-client: at least as fast as local disk
  - Scalability: unaffected by the number of clients
- Cost
  - Capital: less than local (per client) disk storage
  - Operational: zero, it requires no administration
- Capacity: unlimited, it is never full
- Availability: 100%, no failures or service down-time

Key Characteristics of Remote Data Access Solutions

- APIs and transparency
  - How do users and processes access remote data?
  - How closely does remote data mimic local data?
- Performance and robustness
  - Is remote data as fast and reliable as local data?
- Architecture
  - How is solution integrated into clients and servers?
- Protocol and work partitioning
  - How do client and server cooperate?

Remote Data Access and Networking

- ALL forms of remote data access rely on networking
- Which is provided by the operating system as previously discussed
- Remote data access must take networking realities into account
  - Unreliability
  - Performance
  - Security

Remote File Access Architectures

- Client/server
- Remote file transfer
- Remote disk access
- Remote file access
- Cloud model

Client/Server Models

- Peer-to-peer
  - Most systems have resources (e.g., disks, printers)
  - They cooperate/share with one-another
  - Everyone is both client and server (potentially)
- Thin client
  - Few local resources (e.g., CPU, NIC, display)
  - Most resources on work-group or domain servers
- Cloud services
  - Clients access services rather than resources
  - Clients do not see individual servers

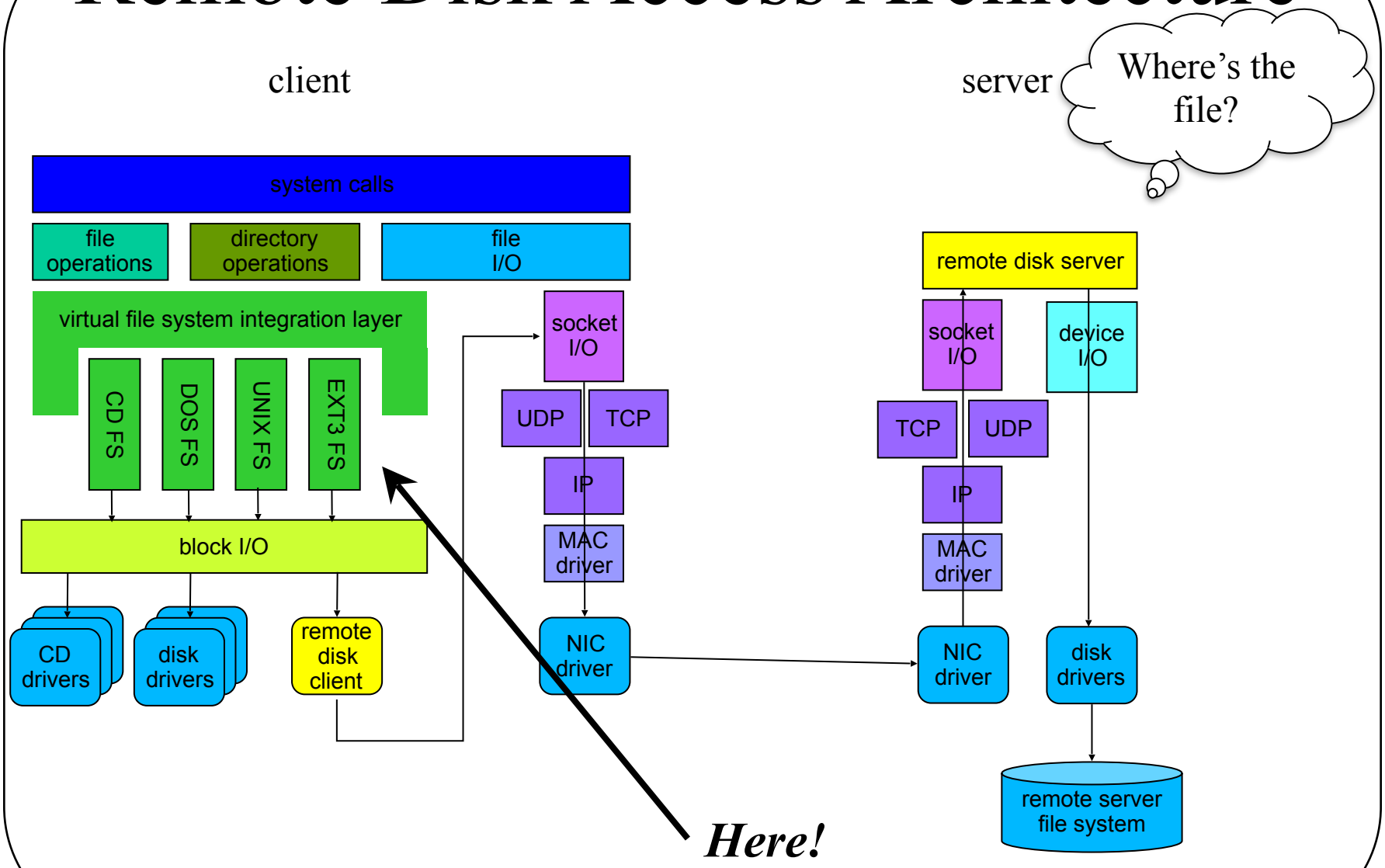
Remote File Transfer

- Explicit commands to copy remote files
  - OS specific: *scp(1)*, *rsync(1),* **S3** tools
  - IETF protocols: FTP, SFTP
- Implicit remote data transfers
  - Browsers (transfer files with HTTP)
  - Email clients (move files with IMAP/POP/SMTP)
- Advantages: efficient, requires no OS support
- Disadvantages: latency, lack of transparency

Remote Disk Access

- Goal: complete transparency
  - Normal file system calls work on remote files
  - All programs "just work" with remote files
- Typical architectures
  - Storage Area Network (SCSI over Fibre Channel)
    - Very fast, very expensive, moderately scalable
  - iSCSI (SCSI over ethernet)
    - Client driver turns reads/writes into network requests
    - Server daemon receives/serves requests
    - Moderate performance, inexpensive, highly scalable

Remote Disk Access Architecture



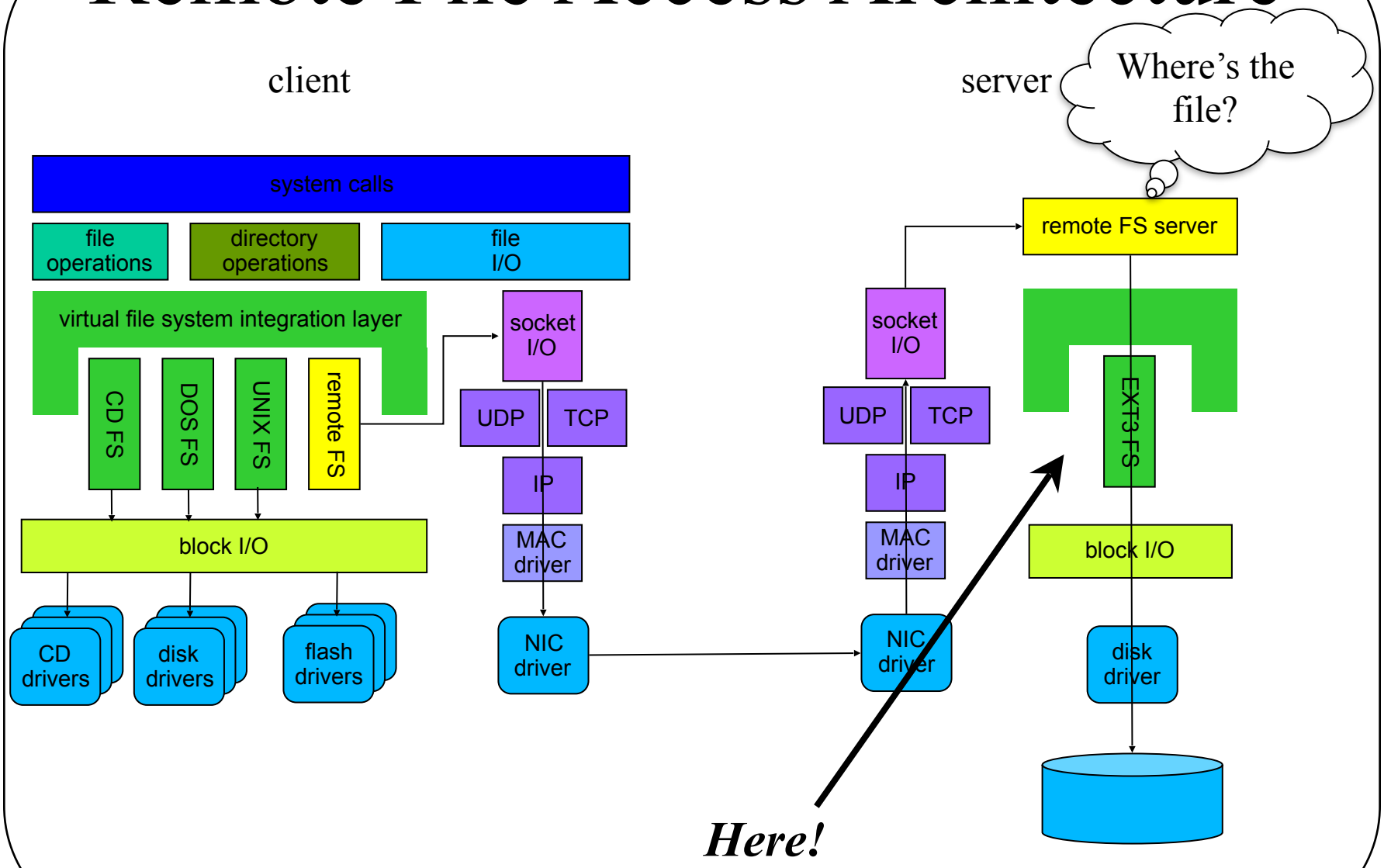
Rating Remote Disk Access

- Advantages:
  - Provides excellent transparency
  - Decouples client hardware from storage capacity
  - Performance/reliability/availability per back-end
- Disadvantages
  - Inefficient fixed partition space allocation
  - Can't support file sharing by multiple client systems
  - Message losses can cause file system errors
- This is THE model for Virtual Machines

Remote File Access

- Goal: complete transparency
  - Normal file system calls work on remote files
  - Support file sharing by multiple clients
  - Performance, availability, reliability, scalability
- Typical architecture
  - Exploits plug-in file system architecture
  - Client-side file system is a local proxy
  - Translates file operations into network requests
  - Server-side daemon receives/process requests
  - Translates them into real file system operations

Remote File Access Architecture



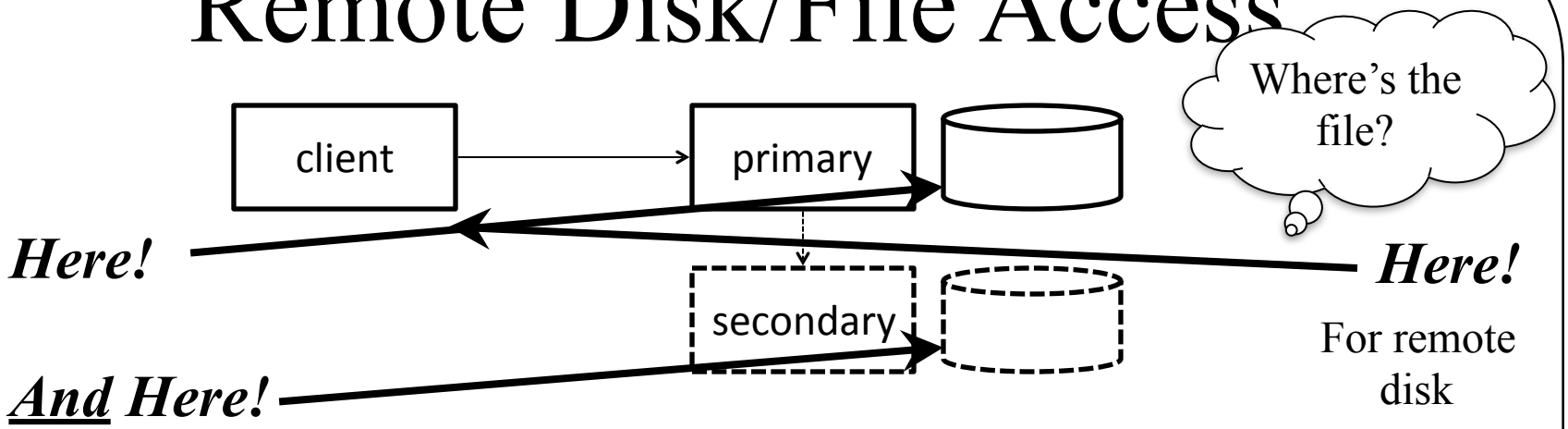
Rating Remote File Access

- Advantages
  - Very good application level transparency
  - Very good functional encapsulation
  - Able to support multi-client file sharing
  - Potential for good performance and robustness
- Disadvantages
  - At least part of implementation must be in the OS
  - Client and server sides tend to be fairly complex
- This is THE model for client/server storage

Cloud Model

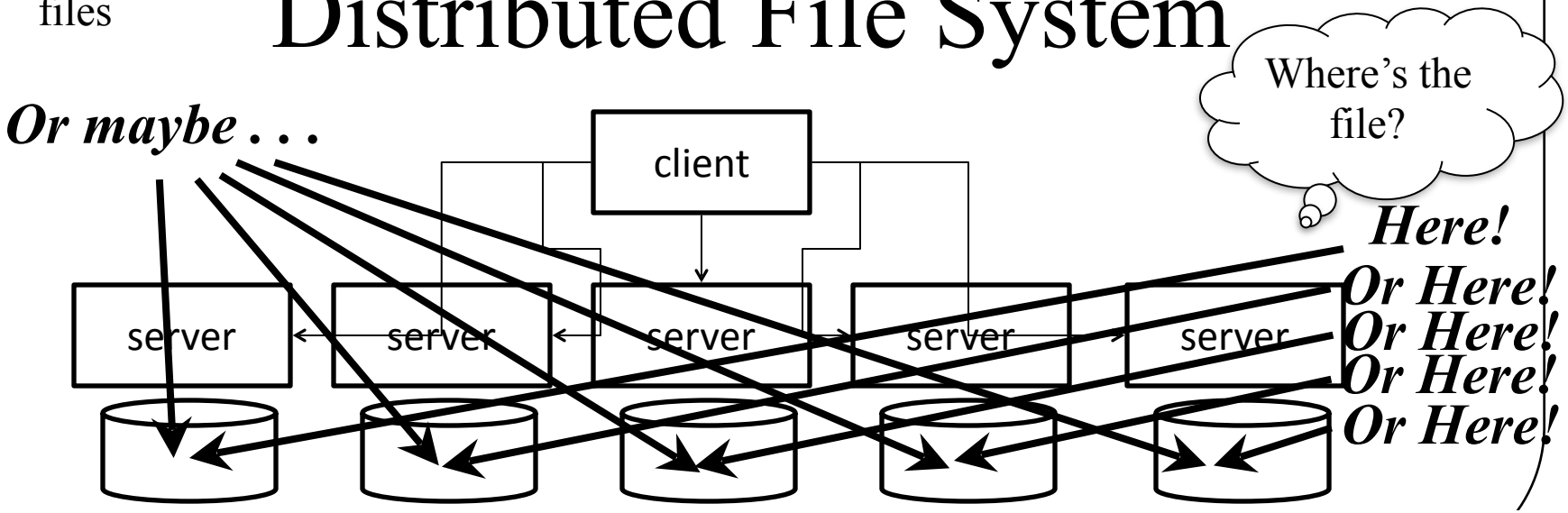
- A logical extension of client/server model
  - All services accessed via standard protocols
- Opaque encapsulation of servers/resources
  - Resources are abstract/logical, thin-provisioned
  - One highly available IP address for all services
  - Mirroring/migration happen under the covers
- Protocols likely to be WAN-scale optimized
- Advantages:
  - Simple, scalable, highly available, low cost
  - A very compelling business model

Remote Disk/File Access



For remote files

Distributed File System



Remote vs. Distributed File System

- Remote file access (e.g., NFS, CIFS)
  - Client talks to (per file system) primary server
  - Secondary server may take over if primary fails
  - Advantages: simplicity
- Distributed file system (e.g., Ceph, Locus)
  - Data is spread across numerous servers
  - Client may talk directly to many/all of them
  - Advantages: performance, scalability
  - Disadvantages: complexity++

Security For Remote File Systems

- Major issues:
  - Privacy and integrity for data on the network
    - Solution: encrypt all data sent over network
  - Authentication of remote users
    - Solution: various approaches
  - Trustworthiness of remote sites
    - Solution: various approaches

Authentication Approaches

- Anonymous access
- Peer-to-peer approaches
- Server authentication approaches
- Domain authentication approaches

Anonymous Access

- All files are available to all users
  - No authentication required
  - May be limited to read-only access
  - Examples: anonymous FTP, HTTP
- Advantages
  - Simple implementation
- Disadvantages
  - Can't provide information privacy
  - Usually unacceptable for write access
    - Which is often managed by other means

Peer-to-Peer Security

- All participating nodes are trusted peers
- Client-side authentication/authorization
  - All users are known to all systems
  - All systems are trusted to enforce access control
  - Example: basic NFS
- Advantages:
  - Simple implementation
- Disadvantages:
  - You can't always trust all remote machines
  - Doesn't work in heterogeneous OS environment
  - Universal user registry is not scalable

Server Authenticated Approaches

- Client agent authenticates to each server
  - Authentication used for entire session
  - Authorization based on credentials produced by server
  - Example: CIFS
- Advantages
  - Simple implementation
- Disadvantages
  - May not work in heterogeneous OS environment
  - Universal user registry is not scalable
  - No automatic fail-over if server dies

Domain Authentication Approaches

- Independent authentication of client & server
  - Each authenticates with independent authentication service
  - Each knows/trusts only the authentication service
- Authentication service may issue signed "tickets"
  - Assuring each of the others' identity and rights
  - May be revocable or timed lease
- May establish secure two-way session
  - Privacy – nobody else can snoop on conversation
  - Integrity – nobody can generate fake messages
- Kerberos is one example

Distributed Authorization

1. Authentication service returns credentials
   - Which server checks against Access Control List
   - Advantage: auth service doesn't know about ACLs
2. Authentication service returns capabilities
   - Which server can verify (by signature)
   - Advantage: servers do not know about clients

- Both approaches are commonly used
  - Credentials: if subsequent authorization required
  - Capabilities: if access can be granted all-at-once

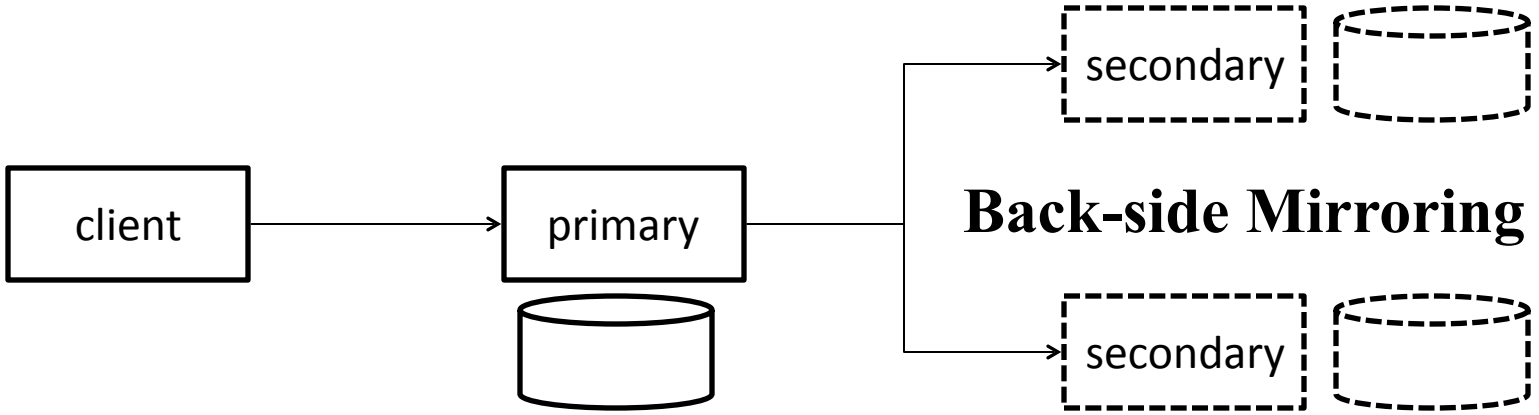
Reliability and Availability

- *Reliability* is high degree of assurance that service works properly
  - Challenging in distributed systems, because of partial failures
  - Data is not lost despite failures
- *Availability* is high degree of assurance that service is available whenever needed
  - Failures of some system elements don't prevent data access
  - Certain kinds of distributed systems can greatly improve availability
- Both, here, in the context of accessing remote files

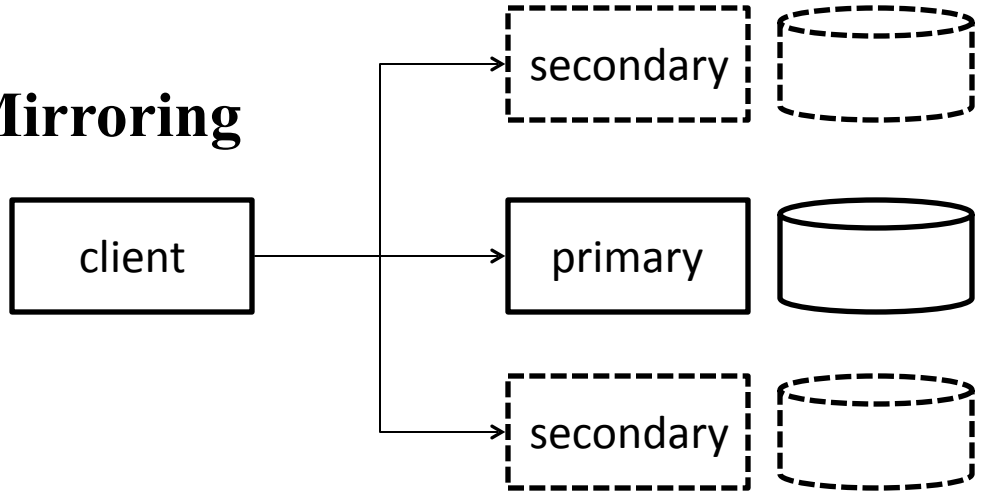
Achieving Reliability

- Must reduce probability of data loss
- Typically by some form of redundancy
  - So disk/server failures don't result in data loss
    - RAID (mirroring, parity, erasure coding)
    - Copies on multiple servers
- Also important to automatically recover after failure
  - Remote copies of data become available again
  - Any redundancy loss due to failure must be made up

Reliability: Data Mirroring



Front-side Mirroring



Mirroring, Parity, and Erasure Coding

- Similar to trade-offs in RAID
  - Extra copies of some data prevent data loss
  - In this case on another machine
  - But the extra copies mean more network I/O
- Mirroring – multiple copies
  - Fast, but requires a great deal of space
- Parity – able to recover from one/two errors
  - Lower space overhead
  - Requires full strip write buffering
- Erasure coding – recover with N/M copies
  - Very space efficient
  - Very slow/expensive reads and writes

Availability and Fail-Over

- Fail-over means transferring work/requests from failed server to some other server
- Data must be mirrored to secondary server
- Failure of primary server must be detected
- Client must be failed-over to secondary
- Session state must be reestablished
  - Client authentication/credentials
  - Session parameters (e.g. working directory, offset)
- In-progress operations must be retransmitted
  - Client must expect timeouts, retransmit requests
  - Client responsible for writes until server ACKs

Availability: Failure Detect/Rebind

- If a server fails, need to detect it and rebind to a different server
- Client driven recovery
  - Client detects server failure (connection error)
  - Client reconnects to (successor) server
  - Client reestablishes session
- Transparent failure recovery
  - System detects server failure (health monitoring)
  - Successor assumes primary's IP address (or other redirection)
  - State reestablishment
    - Successor recovers last primary state check-point
    - Stateless protocol

Availability: Stateless Protocols

- Stateful protocols (e.g., TCP)
  - Operations occur within a context
    - Server must save state
  - Each operation depends on previous operations
  - Replacement server must obtain session state to operate properly
- Stateless protocols (e.g., HTTP)
  - Client supplies necessary context with each request
  - Each operation is self-contained and unambiguous
  - Successor server needs no memory of past events
- Stateless protocols make fail-over easy

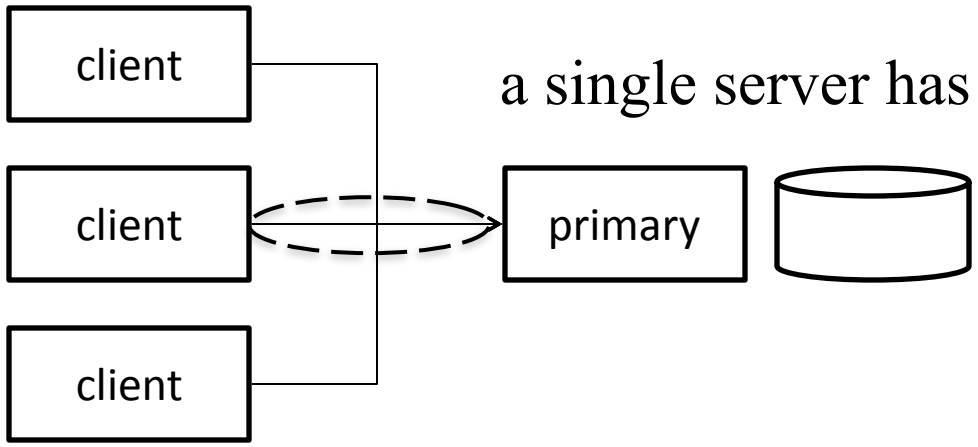
Availability: Idempotent Operations

- Idempotent operations can be repeated many times with same effect
  - Read block 100 of file X
  - Write block 100 of file X with contents Y
  - Delete file X version 3
  - Non-idempotent operations
    - Read next block of current file
    - Append contents Y to end of file X
- If client gets no response, resend request
  - If server gets multiple requests, no harm done
  - Works for server failure, lost request, lost response
    - But no ACK does not mean operation did not happen

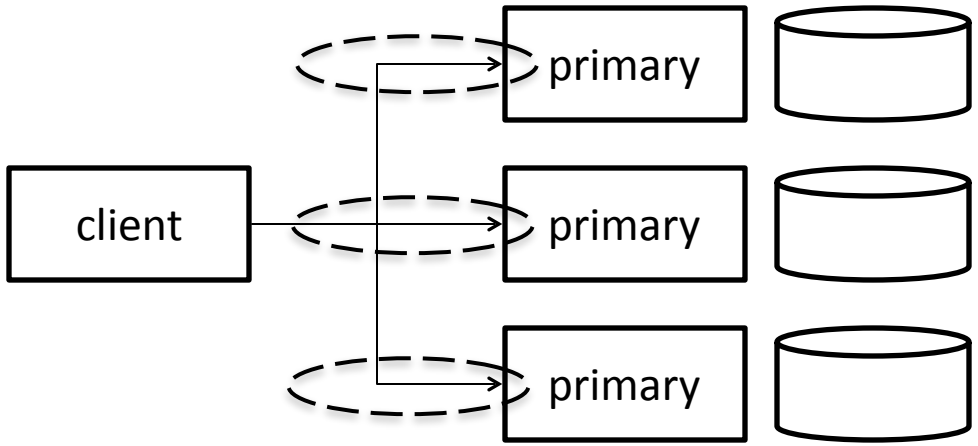
Remote File System Performance

- Drive bandwidth and performance
- Performance for reads
- Performance for writes
- Overheads particular to remote file systems
- Performance and availability

Drive Bandwidth Implications



a single server has limited throughput



striping files across multiple servers provides scalable throughput

Network Impacts on Performance

- Bandwidth limitations
  - Implications for client
  - Implications for server
- Delay implications
  - Particularly important if acknowledgements required
- Packet loss implications
  - If loss rate high, will require acknowledgements

Cost of Reads

- Most file system operations are reads, so read performance is critical
- As usual, improve read performance through caching
- Can use read-ahead, but costs of being wrong are higher than for local drive

Caching For Reads

- Client-side caching
  - Cache data permanently stored at the server at the client
  - Eliminates waits for remote read requests
  - Reduces network traffic
  - Reduces per-client load on server
- Server-side caching
  - Typically performed similarly to single machine caching
  - Reduces drive delays, but not network problems

Whole File Vs. Block Caching

- Many distributed file systems use whole file caching
  - E.g., AFS
- Higher network latency justifies whole file pulls
- Stored in local (cache-only) file system
- Satisfy early reads before entire file arrives
- Block caching is also common (NFS)
  - Typically integrated into shared block cache

Cost of Writes

- Writes at clients need to get to server(s) that store the data
  - And what about other clients caching that data?
- Not caching the writes is very expensive
  - Since they need to traverse the network
  - And probably be acknowledged
- Caching approaches improve performance at potential cost of consistency

Caching Writes For Distributed File Systems

- Write-back cache
  - Create the illusion of fast writes
  - Combine small writes into larger writes
  - Fewer, larger network and disk writes
  - Enable local read-after-write consistency
- Whole-file updates
  - No writes sent to server until *close(2)* or *fsync(2)*
  - Reduce many successive updates to final result
  - File might be deleted before it is written
  - Enable atomic updates, close-to-open consistency
  - But may lead to more potential problems of inconsistency

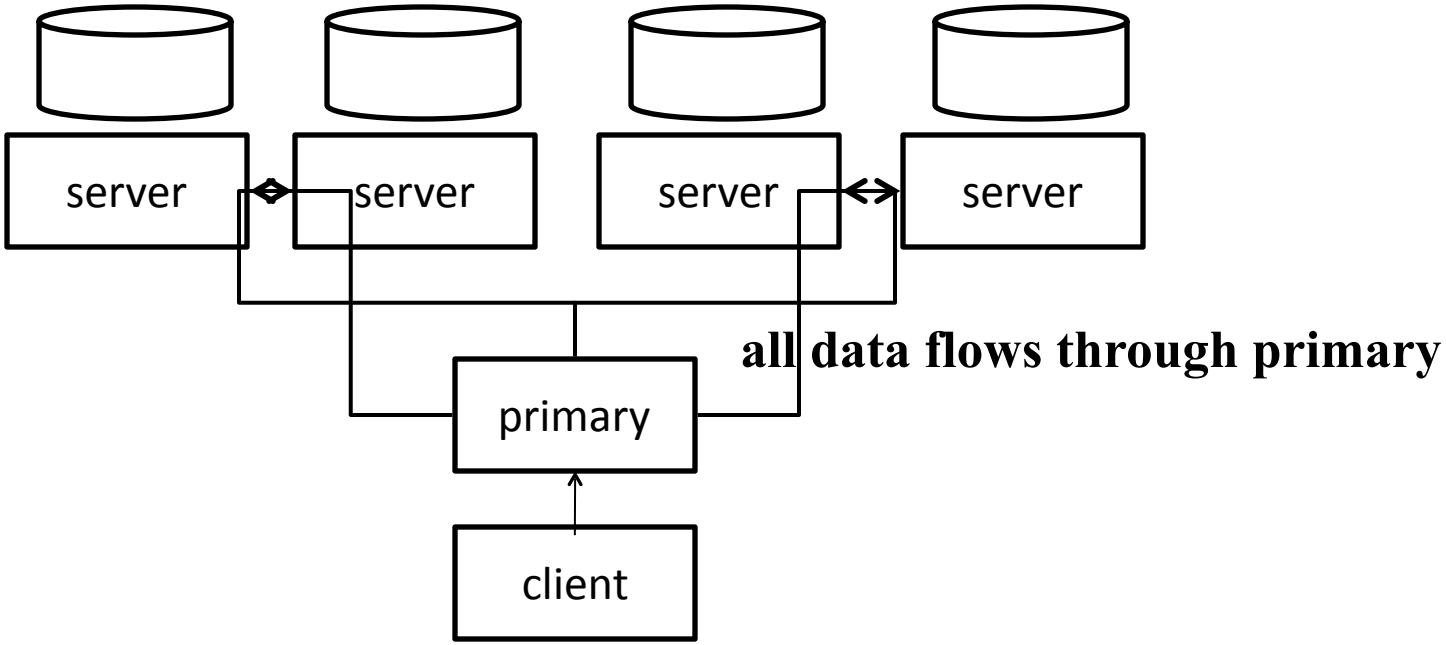
Cost of Consistency

- Caching is essential in distributed systems
  - For both performance and scalability
- Caching is easy in a single-writer system
  - Force all writes to go through the cache
- Multi-writer distributed caching is hard
  - Time To Live is a cute idea that doesn't work
  - Constant validity checks defeat the purpose
  - One-writer-at-a-time is too restrictive for most FS
  - Change notifications are a reasonable alternative
    - But do add network overhead

Cost of Mirroring

- Generally done for reliability and scalability
- Multi-host vs. drive mirroring
  - Protects against host and drive failures, respectively
  - Multi-host creates much additional network traffic
- Mirroring by primary
  - Primary becomes throughput bottleneck
  - Move replication traffic to back-side network
- Mirroring by client
  - Data flows directly from client to storage servers
  - Replication traffic goes through client NIC
  - Parity/erasure code computation on client CPU

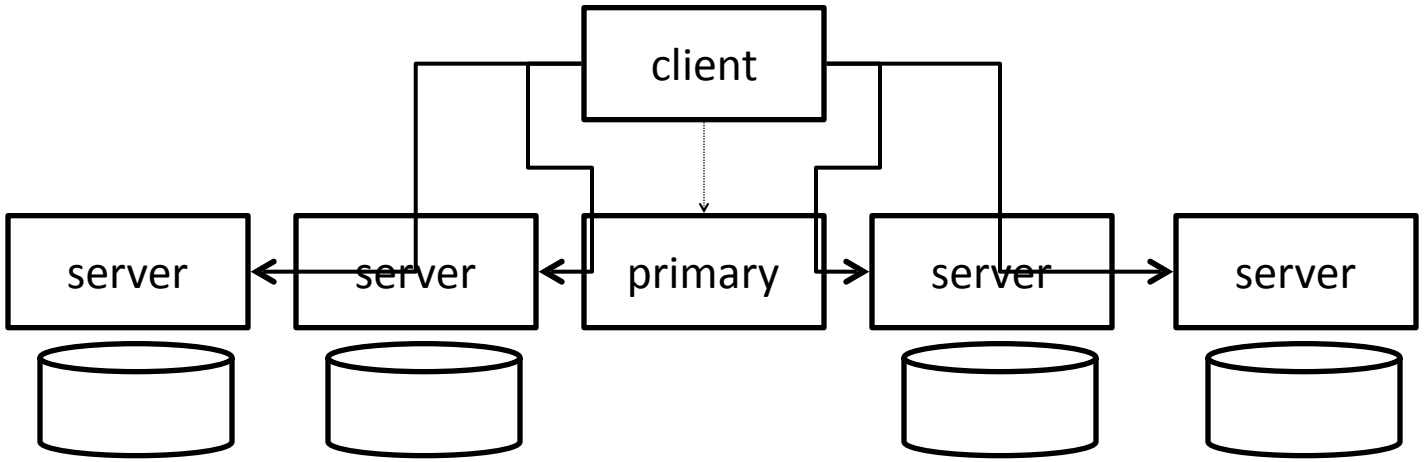
Mirroring Through Primary



Mirroring Through Direct Data Flow

Primary directs client to storage nodes

Data flows direct to storage nodes



Benefits of Direct Data Path

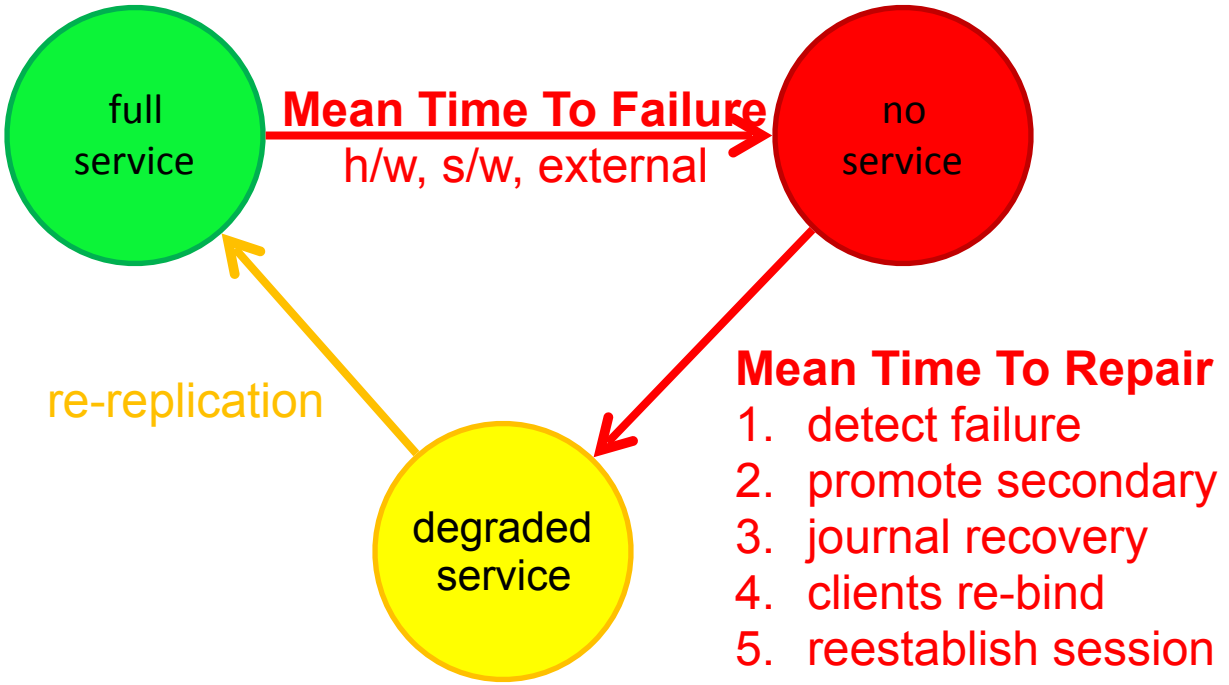
- Architecture
  - Primary tells clients where which data resides
  - Client communicates directly with storage servers
- Throughput
  - Data can be striped across multiple storage servers
- Latency
  - No intermediate relay through primary server
- Scalability
  - Fewer messages on network
  - Much less data flowing through primary servers

Reliability and Availability Performance

- Distributed systems must expect some failures
- Distributed file systems are expected to offer good service despite those failures
- How do we characterize that performance characteristic?
- How do we improve it?

Recovery Time

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$



Improving Availability

- Reduce MTTF
  - Use more reliable components
  - Get rid of bugs
- Or reduce MTTR
  - Use architectures that provide service quickly once recovery starts
  - There are several places where you can improve MTTR

Improving MTTR

- Detect failures more quickly
- Promote secondary to primary role quickly
- Recover recent/in-progress operations quickly
- Inform and rebind clients quickly
- Re-establish session state (if any) quickly
- Degraded service may persist longer
  - Restoring lost redundancy may take a while
  - Heavily loading servers, drives, and network

Scalability and Performance: Network Traffic

- Network messages are expensive
  - NIC and network capacity to carry them
  - Server CPU cycles to process them
  - Client delays awaiting responses
- Minimize messages/client/second
  - Cache results to eliminate requests entirely
  - Enable complex operations with single request
  - Buffer up large writes in write-back cache
  - Pre-fetch large reads into local cache

Scalability Performance: Bottlenecks

- Avoid single control points
  - Partition responsibility over many nodes
- Separated data- and control-planes
  - Control nodes choreograph the flow of data
    - Where data should be stored or obtained from
    - Ensuring coherency and correct serialization
  - Data flows directly from producer to consumer
    - Data paths are optimized for throughput/efficiency
- Dynamic re-partitioning of responsibilities
  - In response to failures and/or load changes

Scalability Performance: Cluster Protocols

- Consensus protocols do not scale well
  - They only work fast for small numbers of nodes
- Minimize number of consensus operations
  - Elect a single master who makes decisions
  - Partitioned and delegated responsibility
- Avoid large-consensus/transaction groups
  - Partition work among numerous small groups
- Avoid high communications fan-in/fan-out
  - Hierarchical information gathering/distribution