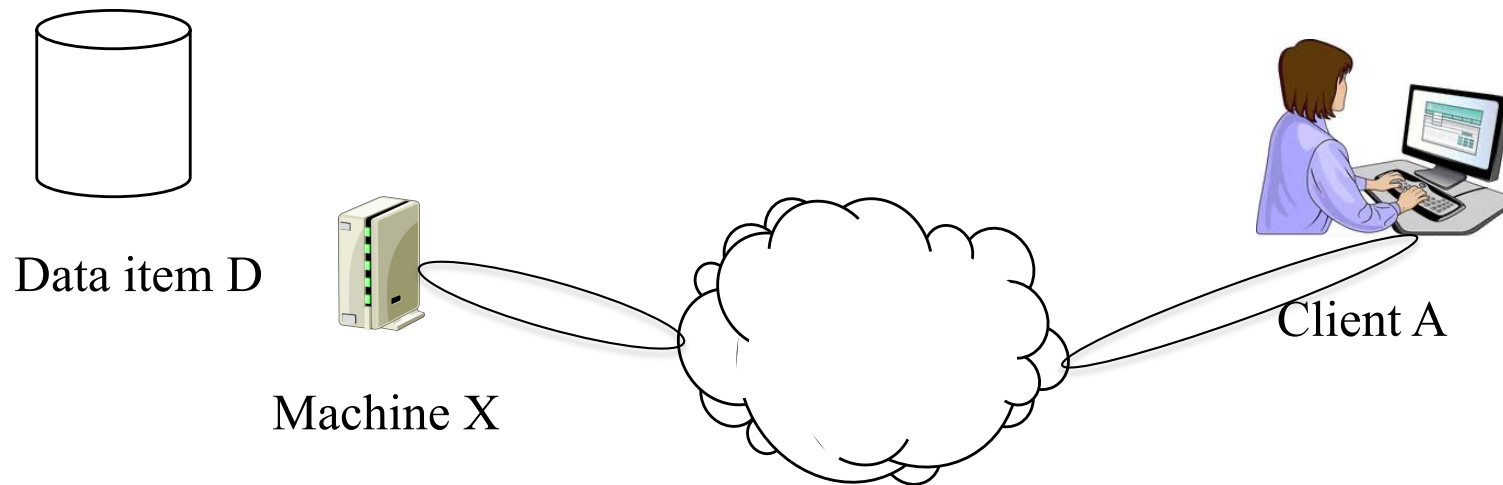


Distributed Systems:
Accessing Remote Data
CS 111
Summer 2025
Operating System Principles
Peter Reiher

Outline

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

The Basic Problem



Client A needs access to data item D

Data item D is on machine X

How do we provide D to client A?

Ideally, with the same properties as if D was
on client A's own machine

Our Goals

- Transparency
 - Indistinguishable from local data for all uses
 - All clients see all data from anywhere
- Performance
 - Per-client: at least as fast as local storage device
 - Scalability: unaffected by the number of clients
- Cost
 - Cheaper than local (per client) persistent storage, zero administrative cost
- Correctness: just like local access correctness
- Capacity: unlimited, it is never full
- Availability: 100%, no failures or service down-time

The Challenges

- Both those of local data access
 - Performance of storage devices
 - Ensuring consistency
 - Providing security
- And those of general distributed systems
 - Uncertainty and extra costs of networking
 - Synchronization and consensus issues
 - Issues of partial system failure
 - Additional trust and security issues due to distributed nature of the system

A Core Decision

- A user/programmer interface decision
- Is accessing data on another machine just like accessing it locally?
 - Which implies using the file system
- Or do you do something very different to access remote data
 - Which implies requiring programmers and users to do something different for remote vs. local
- The former sounds better, but . . .

Key Characteristics of Remote Data Access Solutions

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

A Common, Big Decision

- In most cases of remote data access, the data is treated as a file
 - Rather than a database record, a block on a storage device, a stream of bytes, or whatever
- Usually, a remote file is stored on its host as a local file
 - Usually in identical data format, bit for bit
- But that doesn't necessarily mean it's accessed via the same interface as local files

Remote File Access Architectures

- Remote file transfer
- Remote file access
- Distributed file systems
- Cloud model

Remote File Transfer

- Access to remote data is by obtaining a copy of the file locally
 - Typically the entire file
- Identical bit for bit
- But the storing site does not "connect" the remote file to the local copy
 - If the remote file changes, the local copy doesn't
- Often, remote access is read-only
 - The local site can't change the file

Remote File Transfer Examples

- FTP servers
- File download via the World Wide Web
- Downloading software update files from servers
- Sites that support software version control
 - E.g., Github
 - These do allow update of remote files

Benefits of Remote File Transfer Approaches

- Simplicity
 - You're always moving full files
 - There is no consistency issue (unless the remote file changes during a download)
- Generally good performance
 - The initial download might be slow
 - But access to the local downloaded file is fast
- Good scaling model
 - Use a horizontal scaling system for the server

Downsides of Remote File Transfer

- Doesn't support standard file system operations
 - So there's a big, visible difference between local and remote data
 - Which can complicate program development
- Harder to manage files that receive remote updates
- Usually no support for update propagation
 - You're not told that the remote file changed
 - Unless a separate system notifies you

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



Echoing
RPC

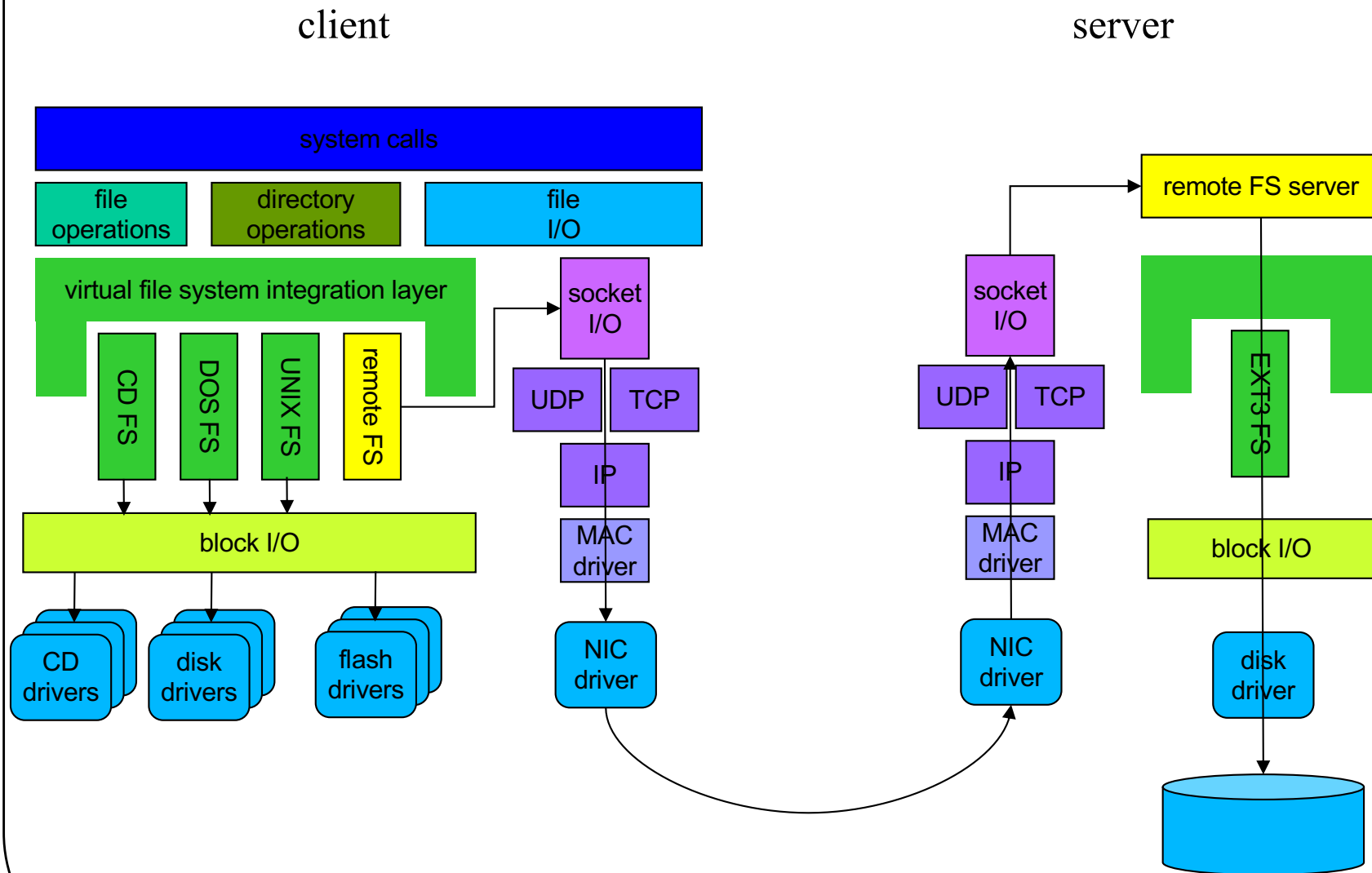
For Example,

- A remote file system client is paired with a remote file system server on another machine
- Access to remote files goes through the file system interface and VFS to the local client
- That client formats messages to be sent to the remote server
 - Which are transmitted through the network I/O protocol stack
- The messages transit the network to the remote machine hosting the server

Completing the Example

- The remote file system server translates the messages into local file system ops
 - E.g., fetching a block of data to satisfy a read request
- The data is packaged into network messages
- Which are sent back to the client's machine
- That machine's client code provides the results to the requesting process

Remote File Access Architecture



Remote File System Administration

- The local system needs to know where to look for the remote files
- Usually done by system administrators setting up the necessary information
 - E.g., mount tables pointing to locations on remote machines
- Implying largely static locations of files
 - Which implies, in turn, potential problems in some failure cases

Remote File Access: Pros and Cons

- 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

Distributed File Systems

- Like remote file systems
- But more so
- The entire collection of all files in a set of machines is regarded as one file system
- Any file accessible from any machine
- Using a single access method, regardless of location
- Often supporting file replication and high degree of file mobility

Distributed File System: Pros and Cons

- Advantages
 - Extremely high application level transparency
 - Potential for excellent reliability and availability
 - Potential for high degree of correct behavior with multiple users of a single file
- Disadvantages
 - Extremely complex with many tough problems
 - Usually high overheads
- Not widely used in production systems

Cloud Model File Systems

- At the client level, provide whatever other model they want
- At the implementation level, scatter their storage (and thus files) over many machines
 - Which are typically virtual
 - And may be spread over many physical cloud machines
- Hide any mismatches between the levels as much as possible

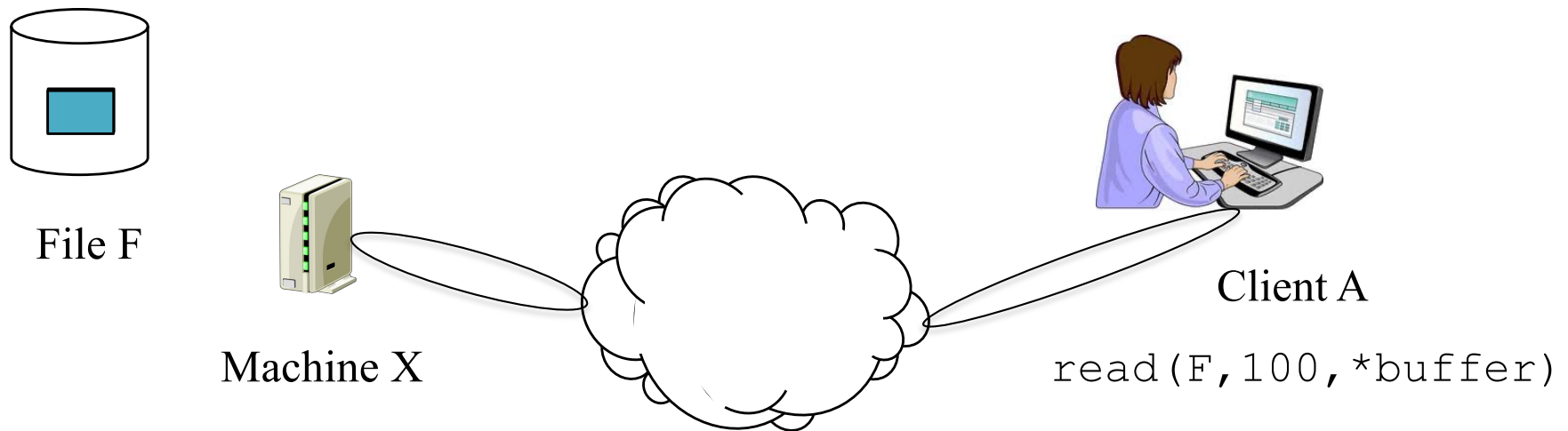
For Example,

- The client wants 20 machines
 - With 5 of them being NFS servers
 - Giving remote file access to the other 15 machines
 - All machines also have local storage
- Maybe the cloud provider uses VMs on 4 physical machines for this service
- Maybe the cloud provider uses VMs on 20 physical machines, due to high load
- Regardless, the 15 NFS clients must have good access to the 5 NFS servers

Remote Data Access Performance

- Perhaps the gating issue for whether a remote data access solution works
- We'll discuss in remote file system contexts
- Storage device bandwidth and performance
- Performance for reads
- Performance for writes
- Overheads particular to remote file systems

The Performance Issues



There will be a delay for the read message to go across the network

There will be a delay Machine X's OS to fetch the block from its storage device

There will be a delay for the message containing the data to go back to Client A

Plus the OS delays on Client A for context switches, buffer management, etc.

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

Performance of Reads

- Most file system operations are reads, so read performance is critical
- Network read consists of several steps:
 - Client application requests read
 - Read request is sent via network
 - Server receives read request
 - Server fetches requested data
 - Server sends data across network
 - Client receives data and gives it to application



Multiple opportunities for optimizations here

Read Caching in Remote File Systems

- Common way to improve read performance is through caching
- Can use read-ahead, but costs of being wrong are higher than for local disk
 - Though benefits are also higher
 - Client can speculatively request reads
 - Or server can speculatively perform them
 - And either send them or cache them locally

Caching For Reads

You can
do both!

- Client-side caching
 - Data permanently stored at the server is cached 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 disk delays, but not network costs

But potential consistency issues

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

Performance 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 multiple 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

But potential
poor remote
consistency

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
 - What do you do when one local cached copy is updated?
 - Must you know about other cached copies?
 - Will they see the update? When?
 - What if two local cached copies are updated?

Distributed Cache Consistency Approaches

- Time To Live
 - Delete items from cache after some time
 - Hoping you don't miss any writes meanwhile
- Check validity on use
 - Which requires remote access, defeating the purpose
- Only allow one writer at a time, per file
 - Too restrictive for most FS
- Change notifications
 - Notify cachiers when main copy gets an update
 - When exactly does the main copy hear about updates?

Usually the chosen solution

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

Peer-to-Peer Authentication

- 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: Login-based FTP, SCP, 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

Trustworthiness of Remote Sites

- Sometimes based on administrative issues
 - E.g., all sites controlled by one company/administrator
- Some authentication solutions limit need for trust
 - E.g., Kerberos
- Cryptographic approaches sometimes possible
 - E.g., if you don't trust your cloud storage site, only store encrypted data there

Reliability and Availability

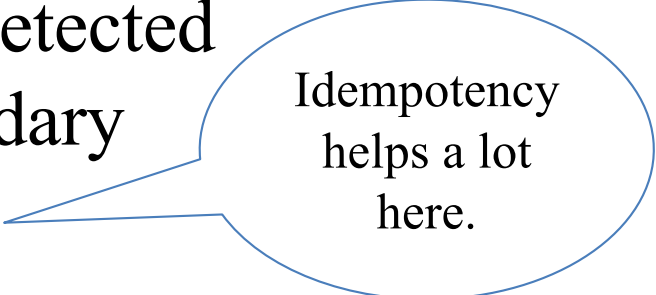
- *Reliability* is high degree of assurance that service works properly
 - Challenging in distributed systems, because of partial failures
 - Data should not be lost despite failures
- *Availability* is high degree of assurance that service is available whenever needed
 - Failures of some system elements 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
 - Disk/server failures must not result in data loss
 - Spread data across multiple disks, with redundancy
 - Copies on multiple servers
 - Backup, at the worst
- Also important to automatically recover after failure
 - Remote copies of data become available again
 - Redundancy loss due to failure must be made up

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



Idempotency
helps a lot
here.

Availability: Failure Detect/Rebind

- If a server fails, clients 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

Scalability

- All hardware has its limits
 - Networks have limited bandwidth
 - Caches are of limited capacity
 - CPUs perform limited instructions per second
 - Storage devices are of limited size and can service a limited number of requests per second
- For small distributed systems with light load, maybe not a problem
- For large ones, remote data access scaling is a huge deal

Scalability and Performance: Network Traffic

- Network messages are expensive
 - They use NIC and network capacity to carry them
 - And 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

But remember

...

**If you don't send a message,
no one else knows it happened**

Scaling of Servers

- With modern hardware, CPU speed is probably not the main issue for file servers
- But storage device speed can be a problem
 - When a server is handling many clients per second
- Partially handled by server caching
- But also an issue for device scheduling
 - See some of the discussion in the device driver lecture

Hardware Device Scaling Solutions

- Servers can host multiple storage devices
- Total capacity to handle requests can thus be increased
- Either by storing many files on many devices
 - But can you keep all the devices busy?
- Or storing multiple copies of fewer files on multiple devices
 - But you're paying more hardware cost per byte stored

Conclusion

- Accessing data on remote machines is key to most distributed processing
- There are major challenges to doing so:
 - Performance
 - Consistency
 - Reliability
 - Scalability
- Solutions are available, but have associated costs and drawbacks
 - None of them are perfect for all purposes