CS 423
Operating System Design: Distributed File Systems

Acknowledgement: This slide set is based on lecture slides by Prof. John Kubiatowicz, UC Berkeley, Dr. Guohui Wang, Rice University, and Prof. Kenneth Chiu, SUNY Binghamton

Distributed File Systems

- A file system provides a service for clients. The server interface is the normal set of file operations: create, read, etc. on files.
- A Distributed File System (DFS) is simply a classical model of a file system distributed across multiple machines. The purpose is to promote sharing of dispersed files.
- The resources on a particular machine are local to itself. Resources on other machines are remote.

Distributed File Systems

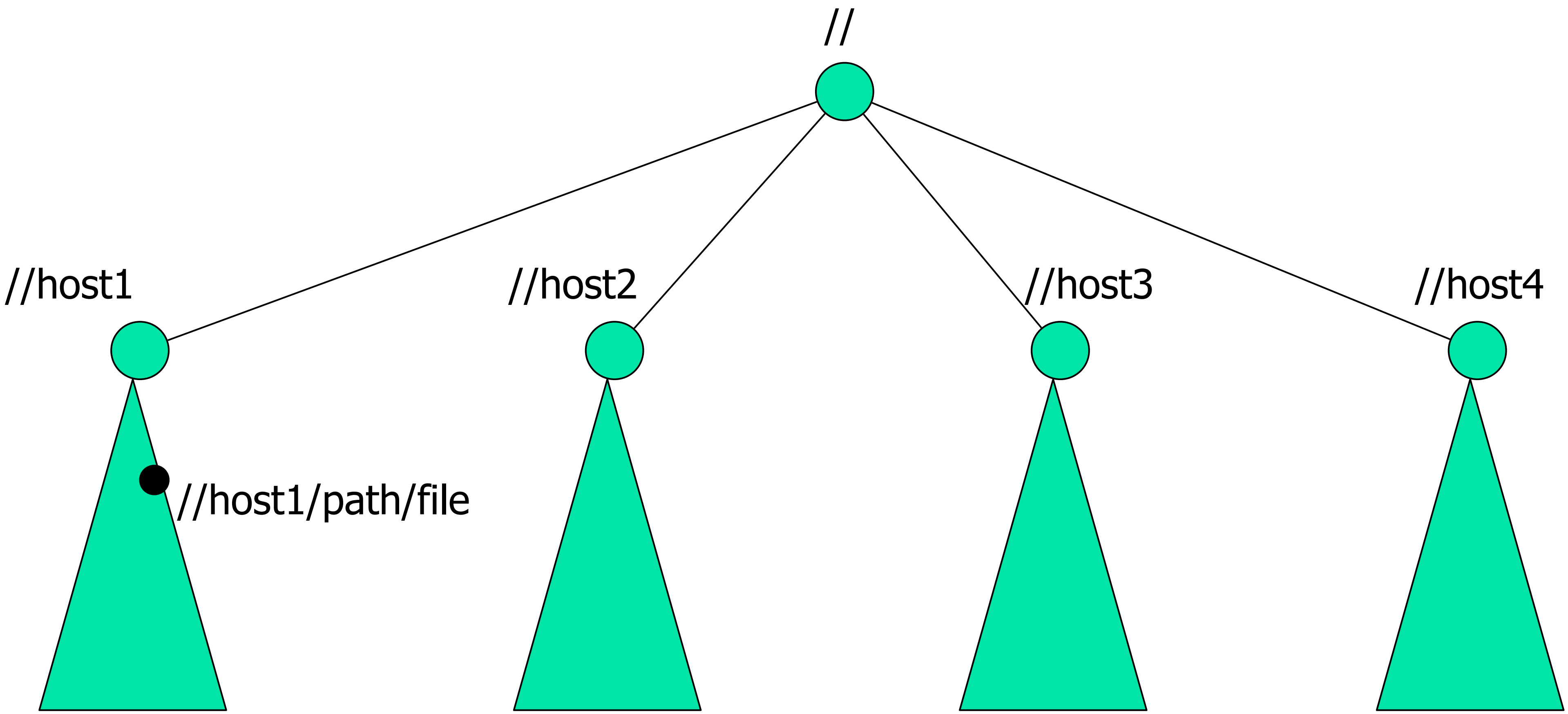
- **Naming**: mapping between logical and physical objects.
- **Location transparency**:
  - The name of a file does not reveal any hint of the file's physical storage location.
- **Location independence**:
  - The name of a file doesn't need to be changed when the file's physical storage location changes.

Naming Schemes

- Files are named with a **combination** of host and local name.
  - This guarantees a unique name. NOT location transparent NOR location independent.
  - Same naming works on local and remote files. The DFS is a loose collection of independent file systems.
- Remote directories are **mounted** to local directories.
  - So a local system seems to have a coherent directory structure.
  - The remote directories must be explicitly mounted. The files are location transparent.
  - SUN NFS is a good example of this technique.
- A **single global name structure** spans all the files in the system.
  - The DFS is built the same way as a local filesystem. Location independent.

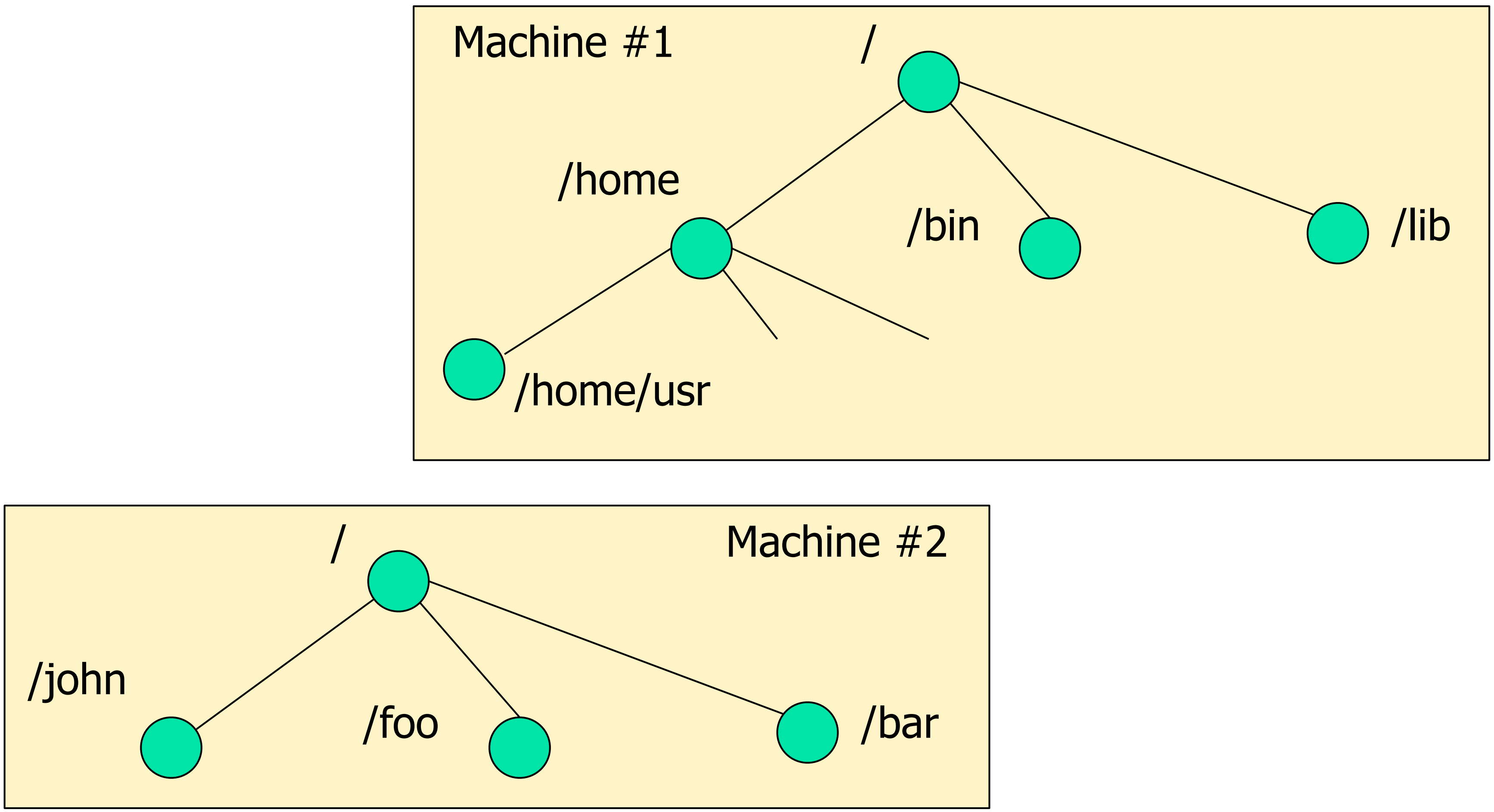
Example I

No location transparency:



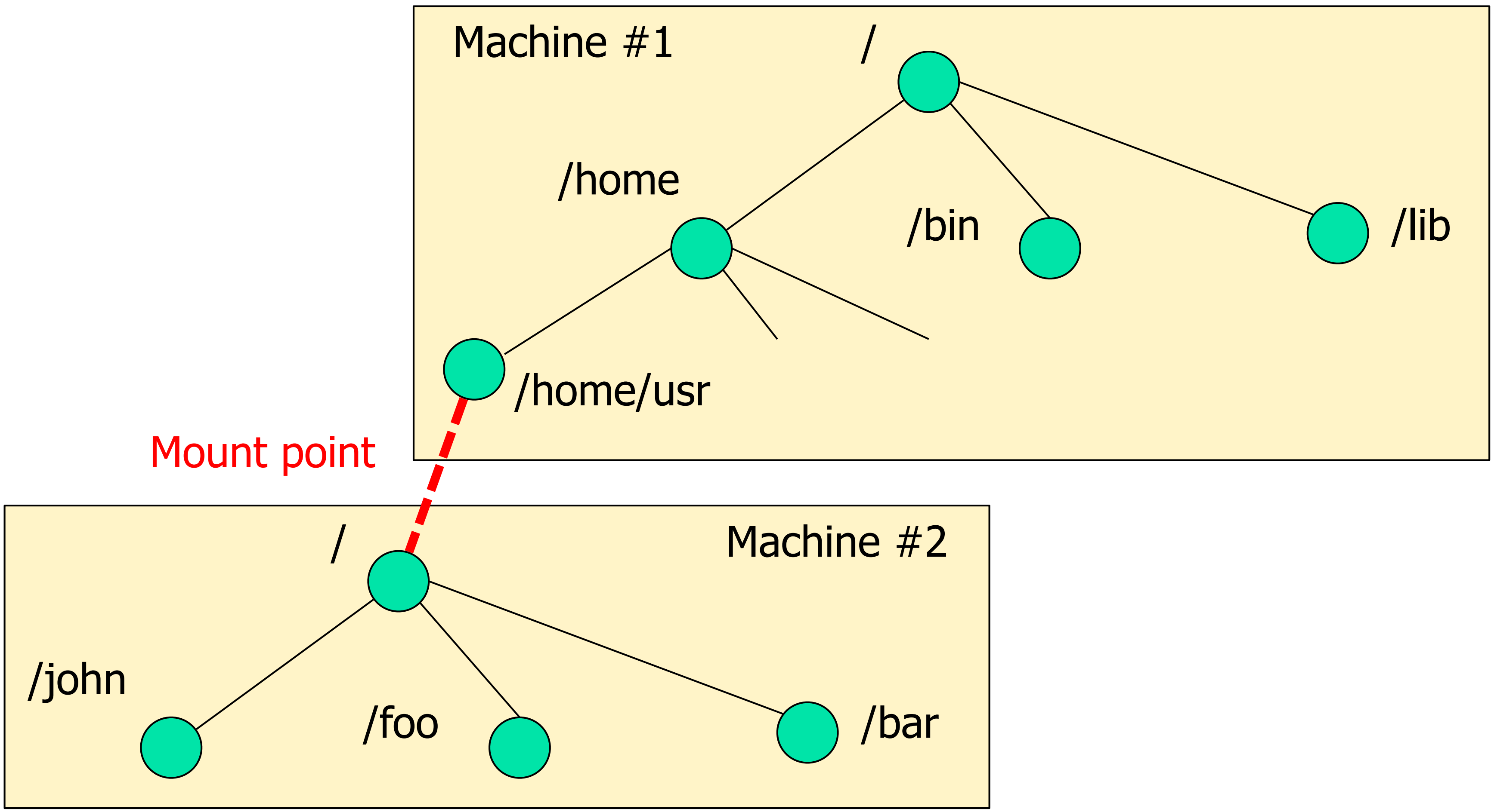
Example 2

Location transparency in NFS



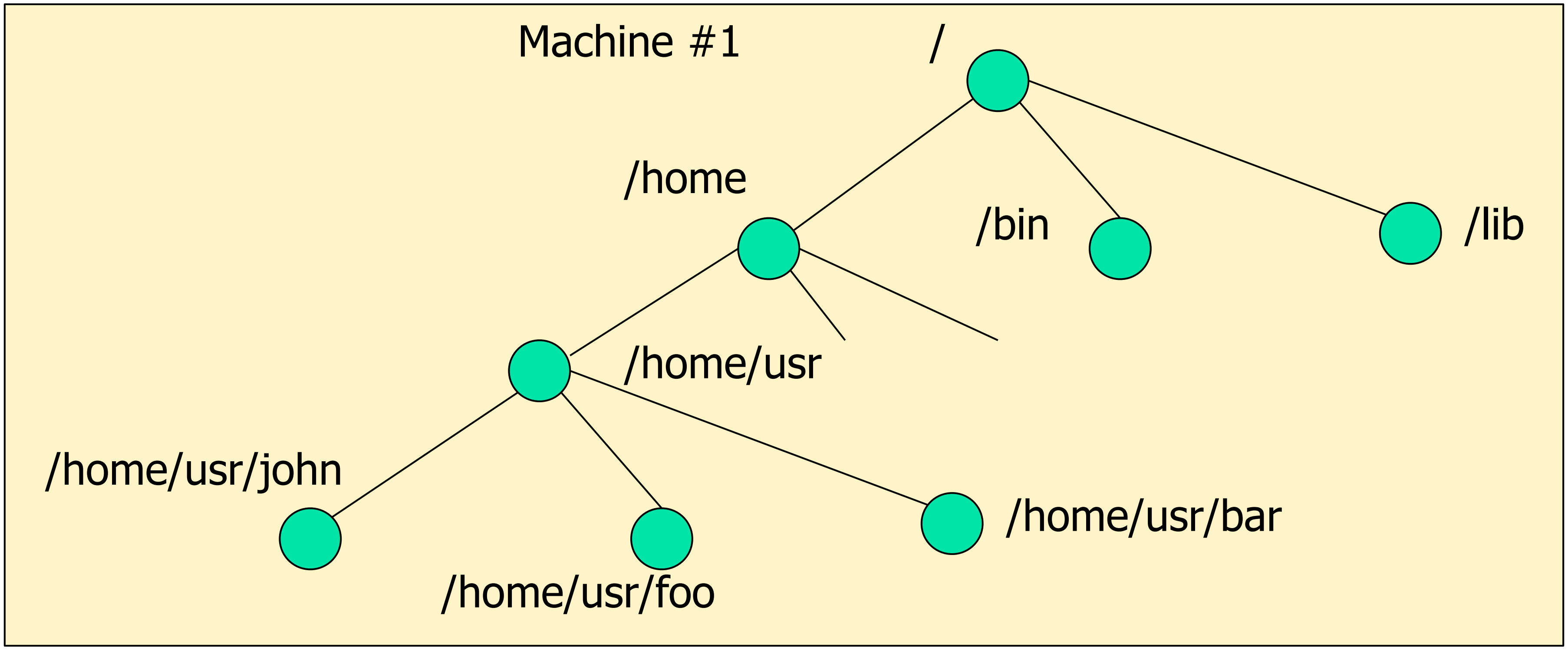
Example 2

Location transparency in NFS: **`mount`** *operation*



Example 2

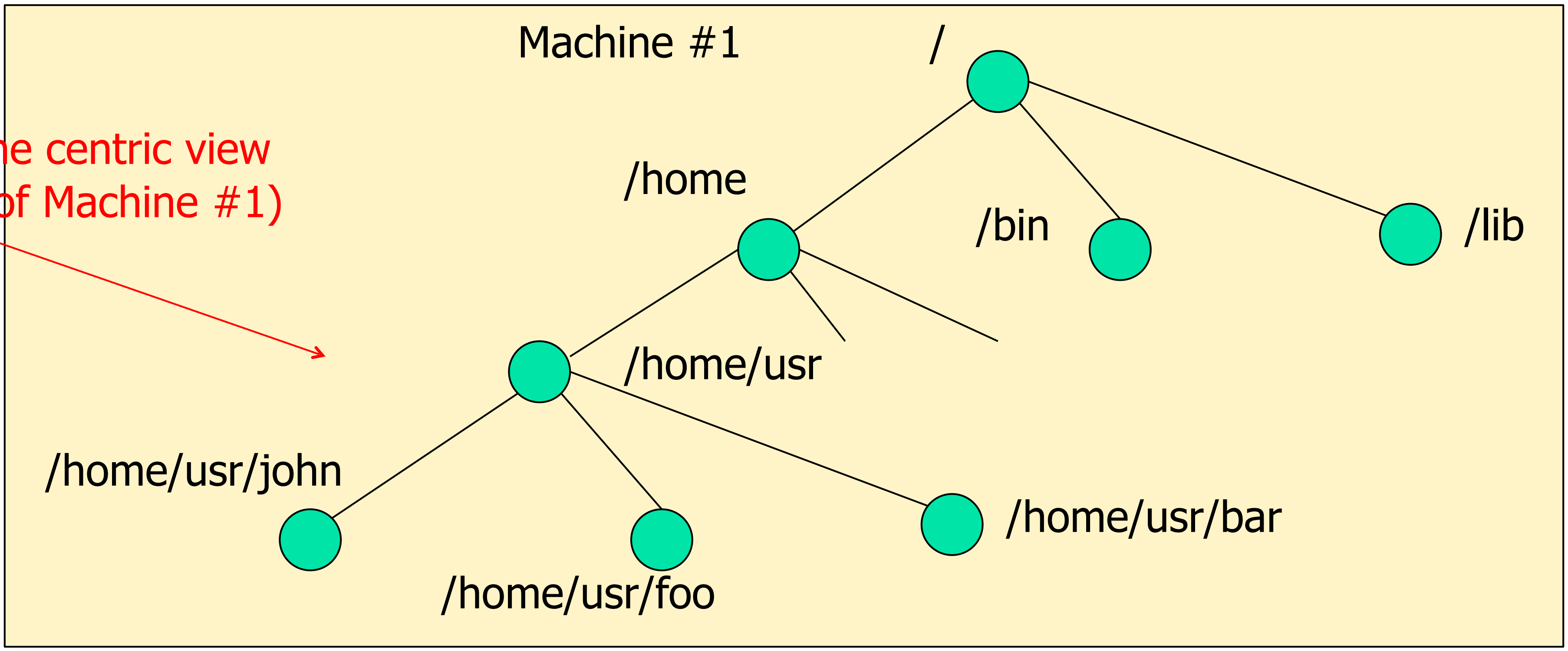
Location transparency in NFS: The Logical View



Example 2

Location transparency in NFS: The Logical View

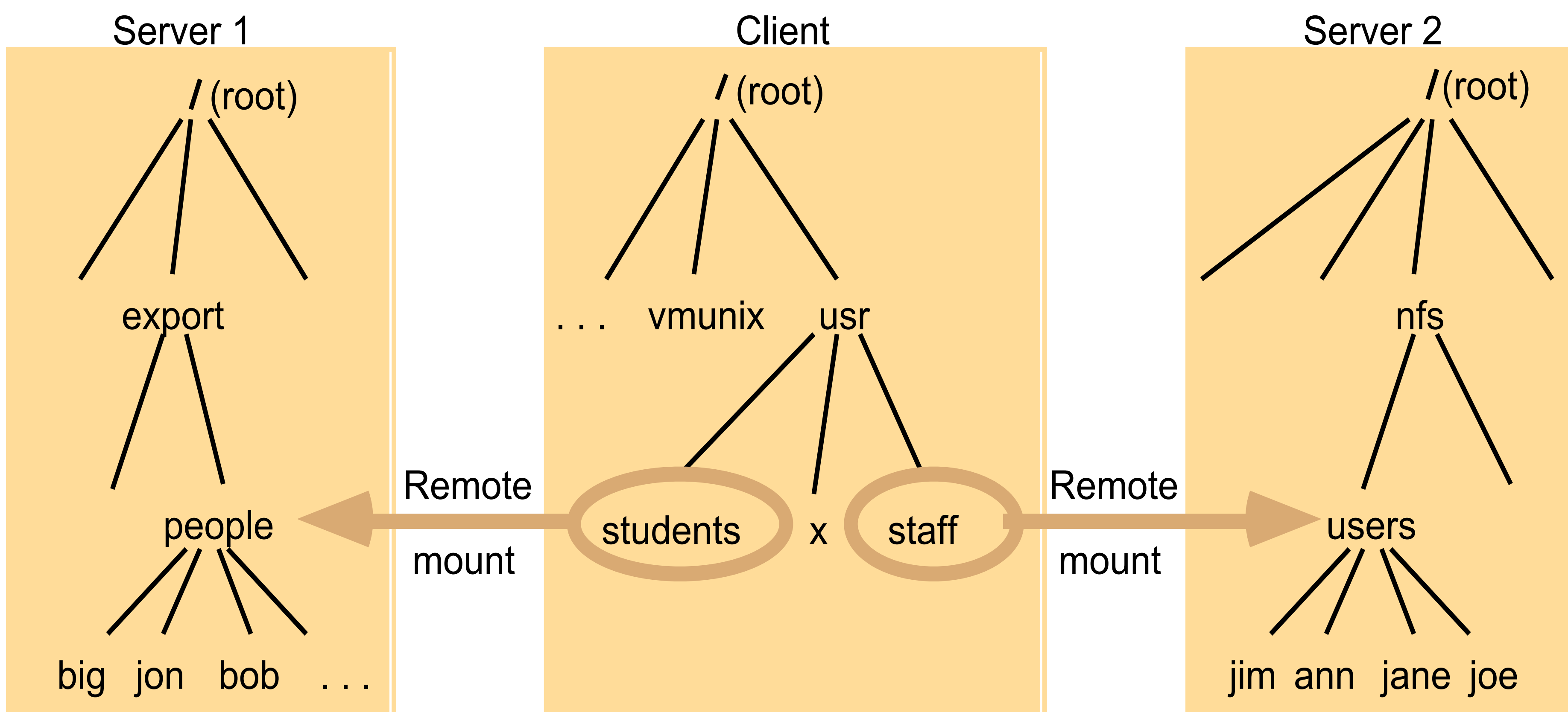
Machine centric view
(view of Machine #1)



No location independence: If I moved files from server to server, I may need to change the mount points

Example 2

Local and Remote File Systems on an NFS Client:

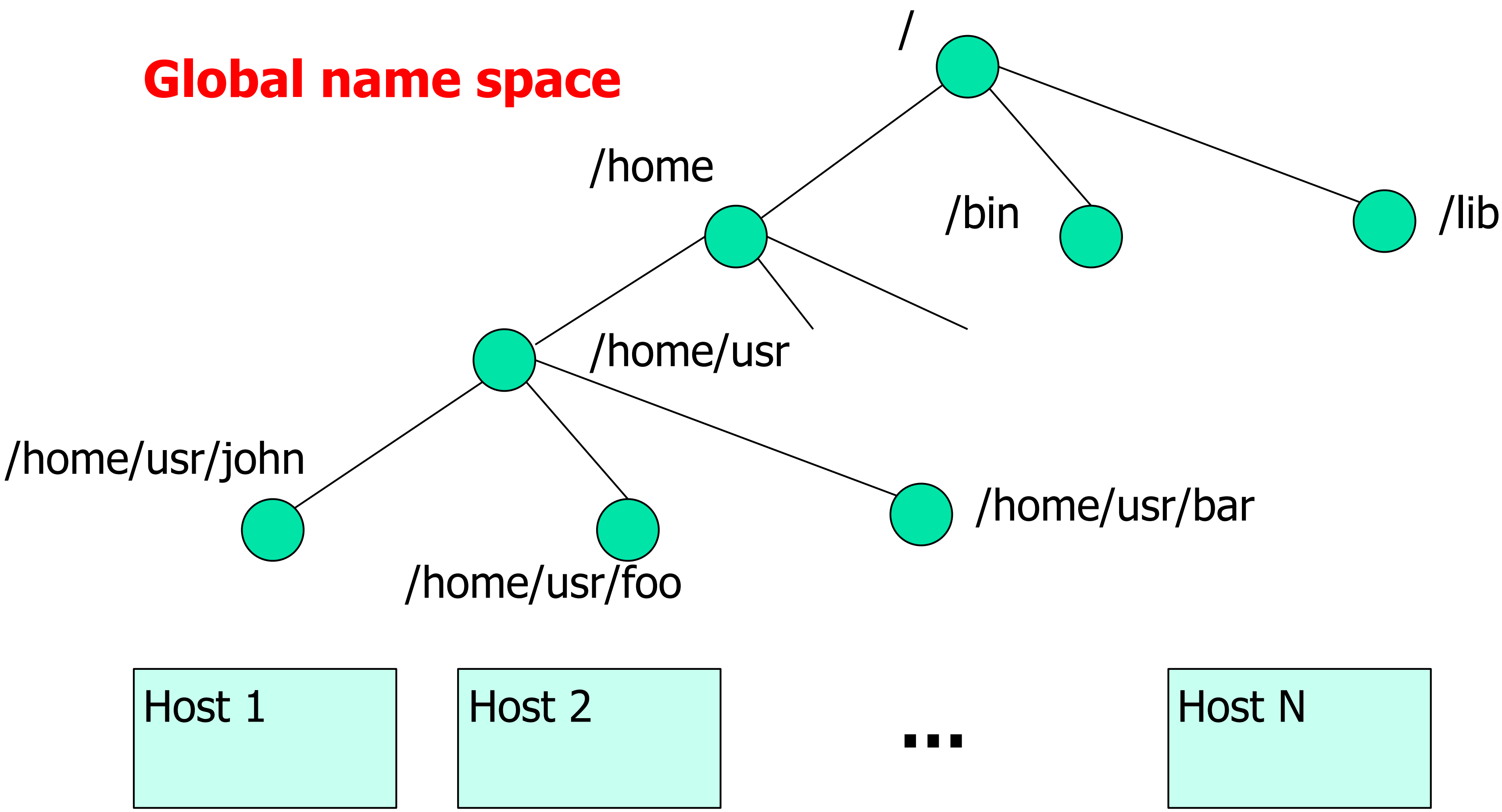


mount –t nfs Server1:/export/people /usr/students

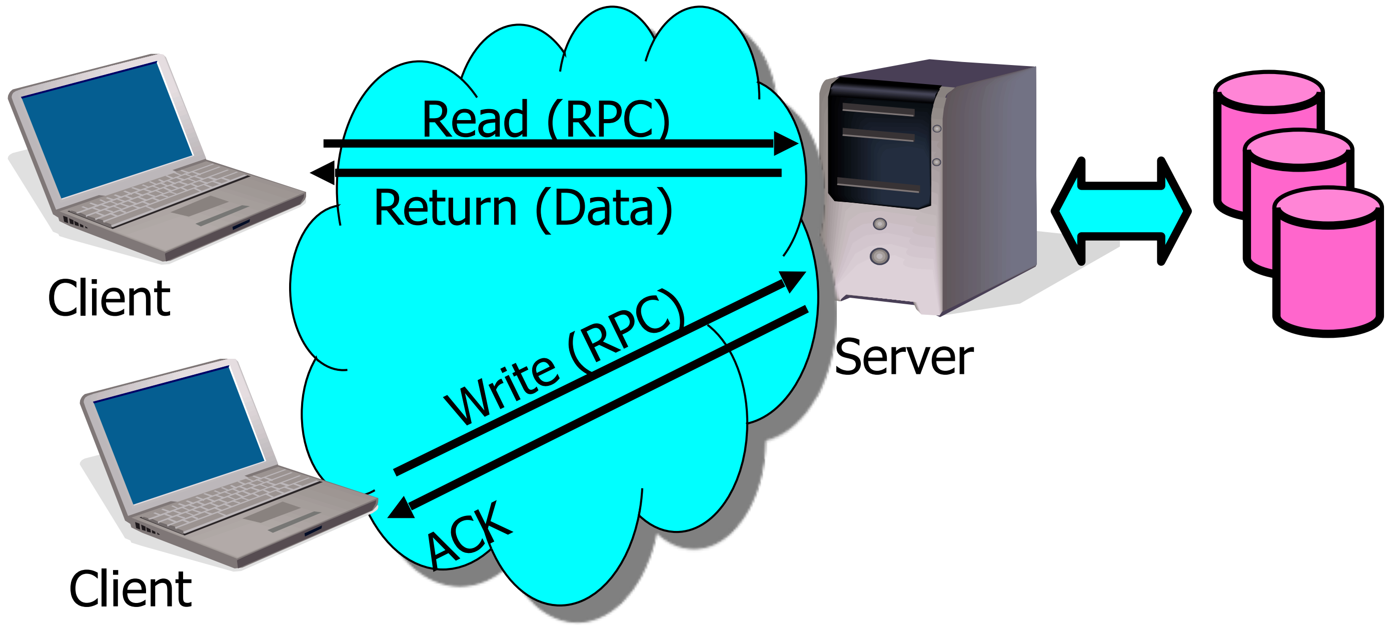
mount –t nfs Server2:/nfs/users /usr/staff

Example 3

Local independence in Andrew:

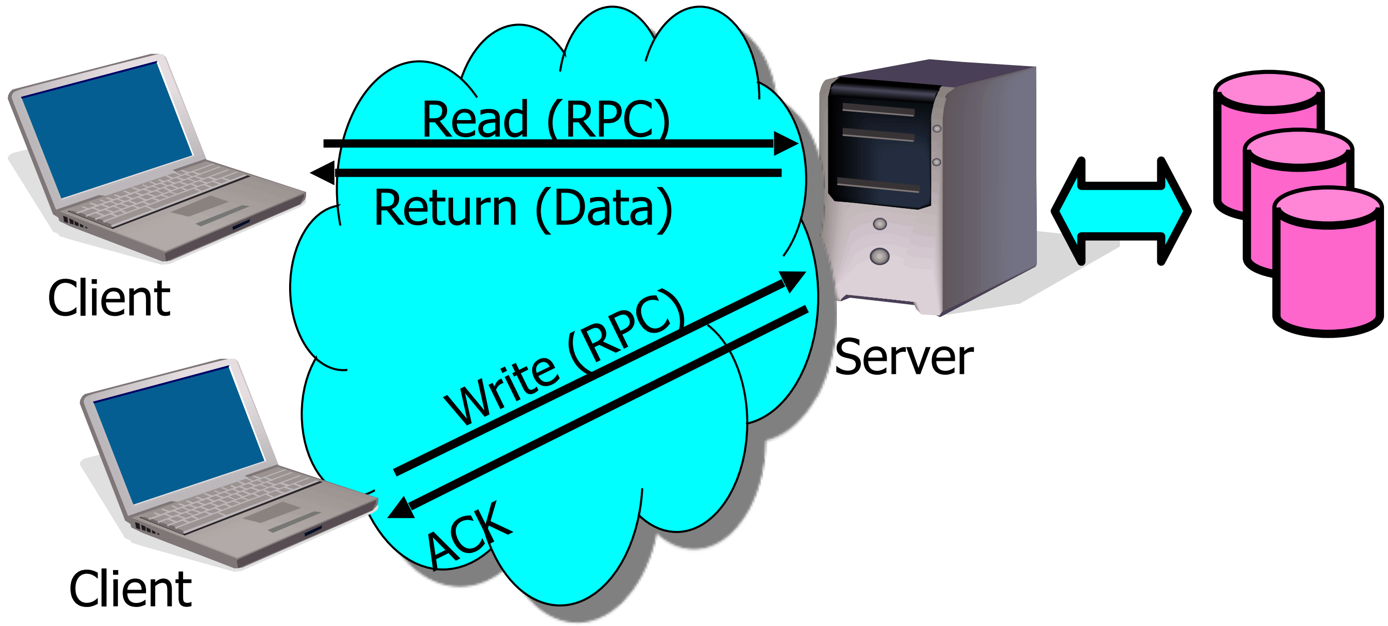


Simple Distributed FS



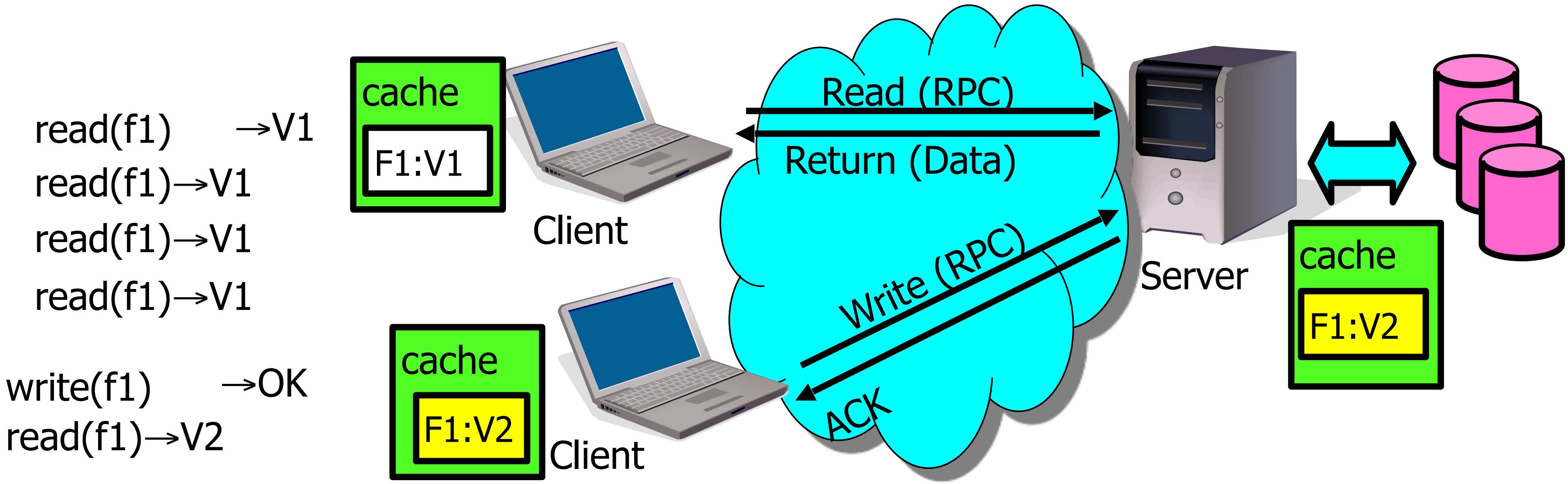
- Remote Disk: Reads and writes forwarded to server
  - Use RPC to translate file system calls
  - No local caching
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems?

Simple Distributed FS



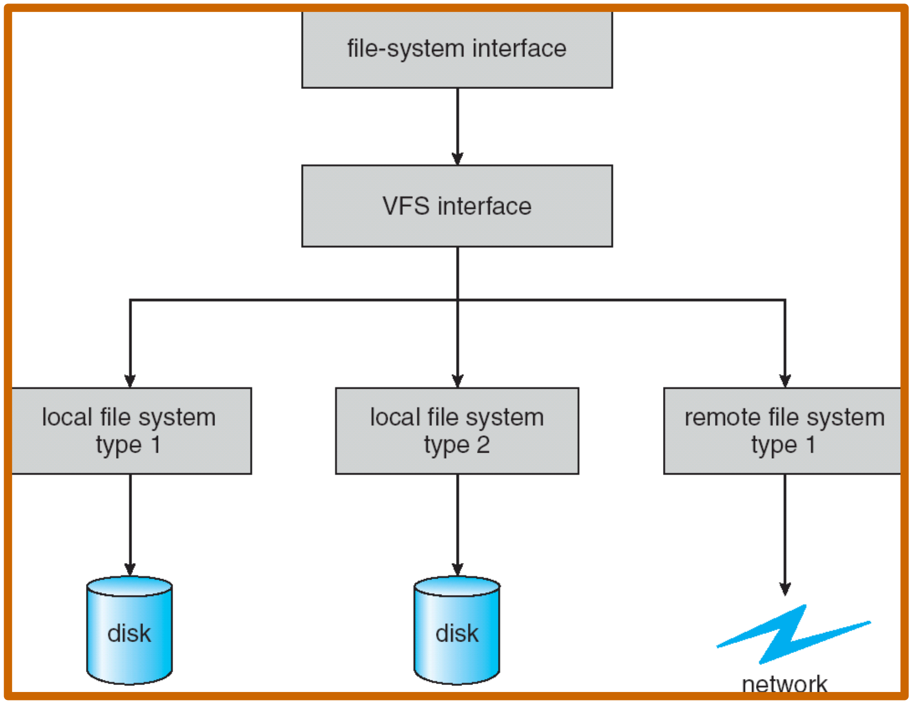
- Remote Disk: Reads and writes forwarded to server
  - Use RPC to translate file system calls
  - No local caching
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems?
  - Going over network is slower than going to local memory
  - Server can be a bottleneck

Distributed FS w/ Caching



- Idea: Use caching to reduce network load
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure: Client caches have data not committed at server
  - Cache consistency! Client caches not consistent with server/ each other

Virtual FS

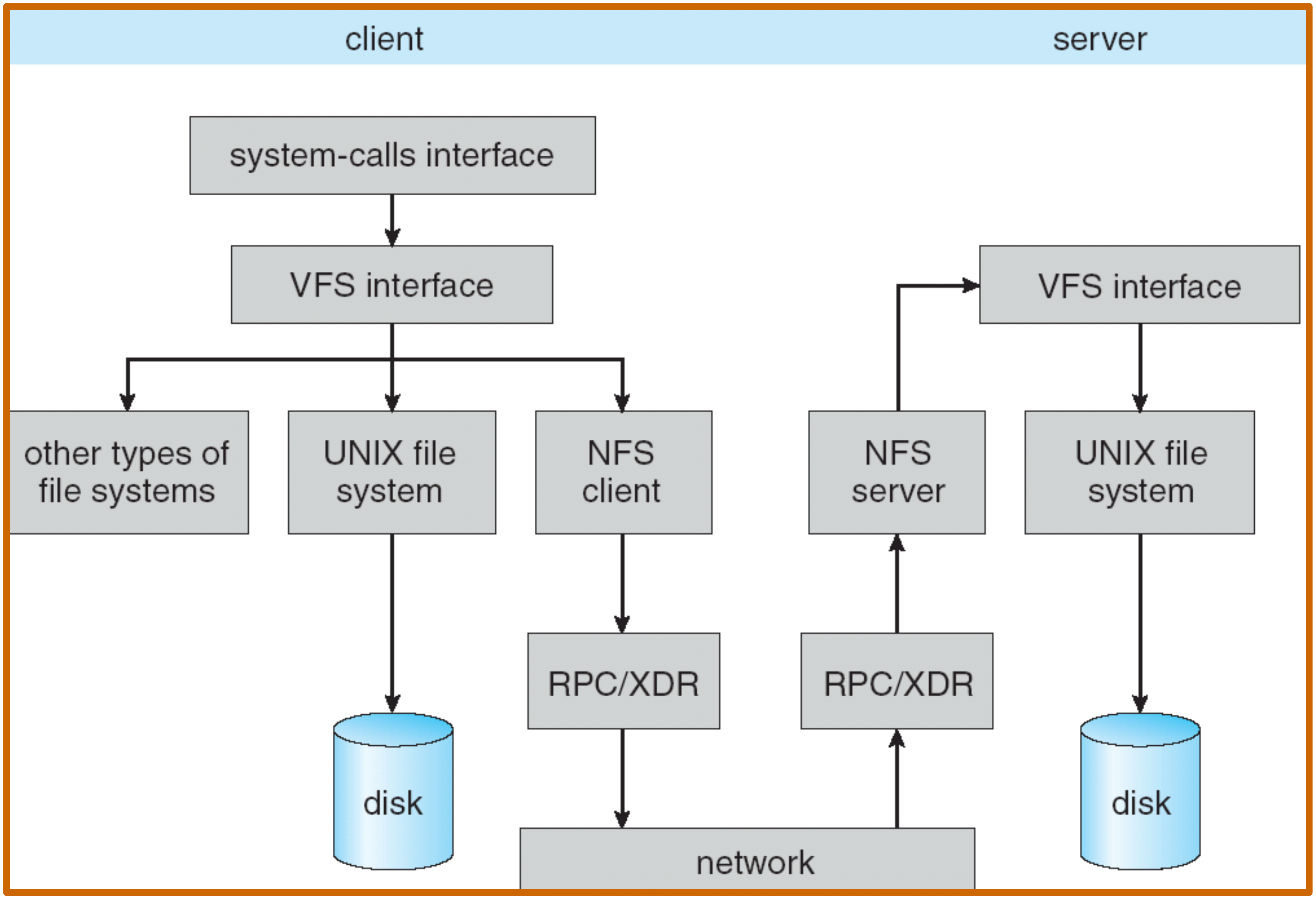


- VFS: Virtual abstraction similar to local file system
  - Instead of “inodes” has “vnodes”
  - Compatible with a variety of local and remote file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems (The API is to the VFS interface)

Network File System (NFS)

- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes!

Schematic View of NFS



Network File System (NFS)

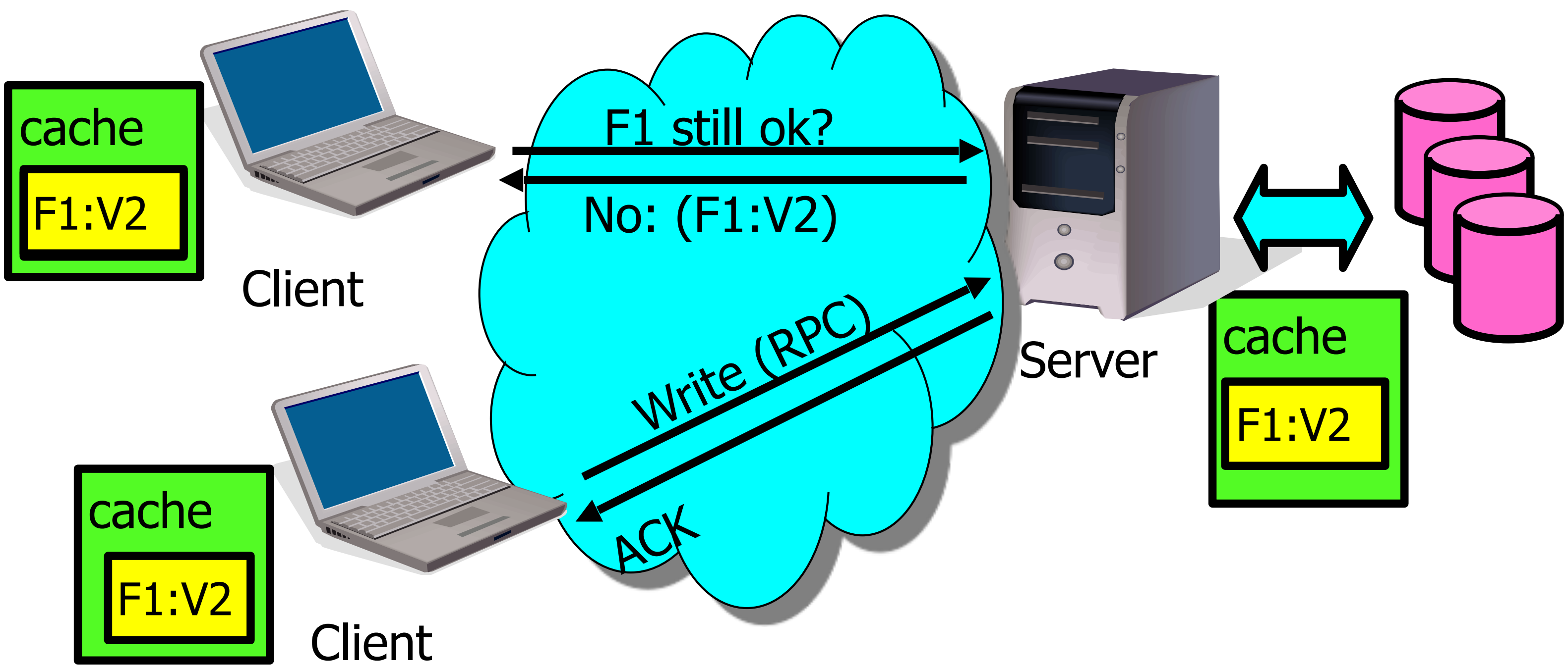
- NFS servers are stateless; each request provides all arguments required for execution
  - E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error

Network File System (NFS)

- Failure Model: Transparent to client system
  - Options (NFS Provides both):
    - Hang until server comes back up (next week?)
    - Return an error. (Of course, most applications don't know they are talking over network)

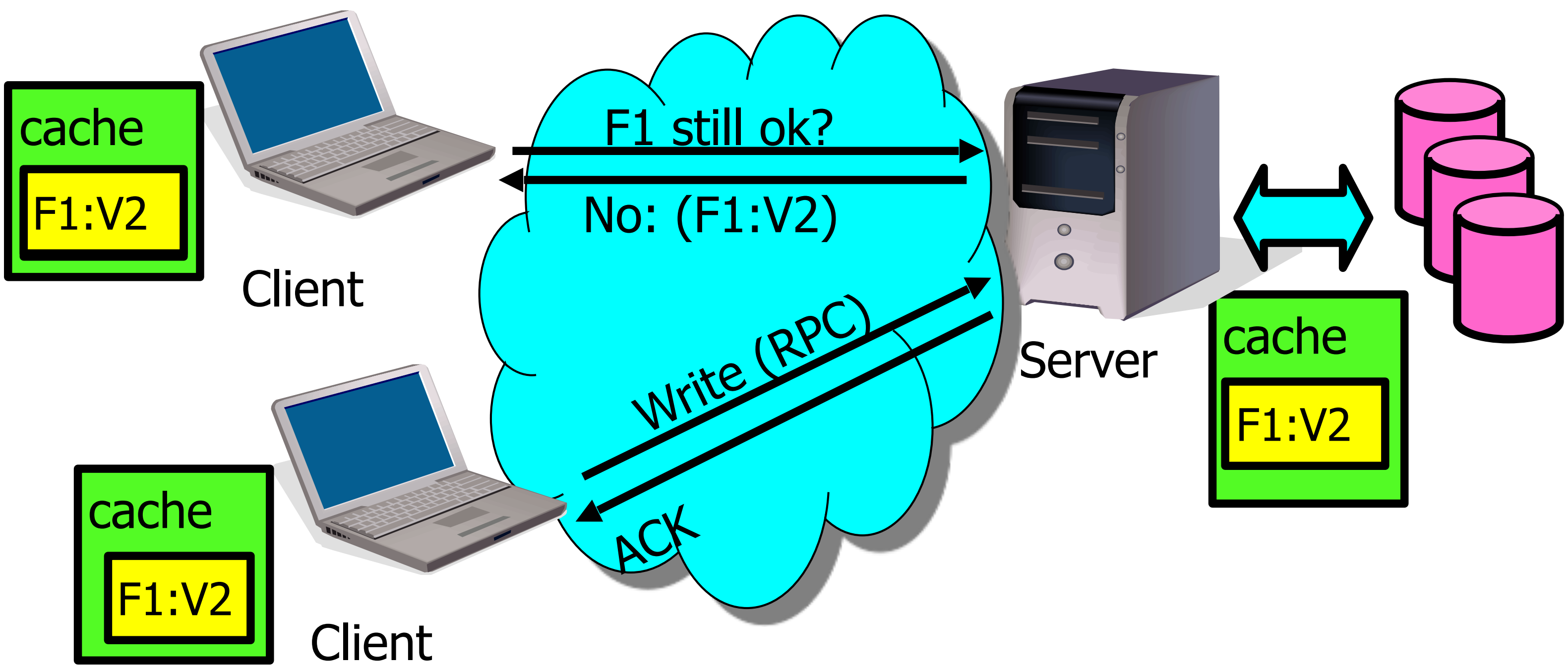
NFS Cache Consistency

- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



NFS Cache Consistency

- NFS protocol: weak consistency
  - What if multiple clients write to same file?
    - In NFS, can get either version (or parts of both)
    - Completely arbitrary!



Andrew File System

- Andrew File System (AFS, late 80's)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - As a result, do not get partial writes: all or nothing!
    - Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

Andrew File System

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - Get file from server, set up callback with server
  - On write followed by close:
    - Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost

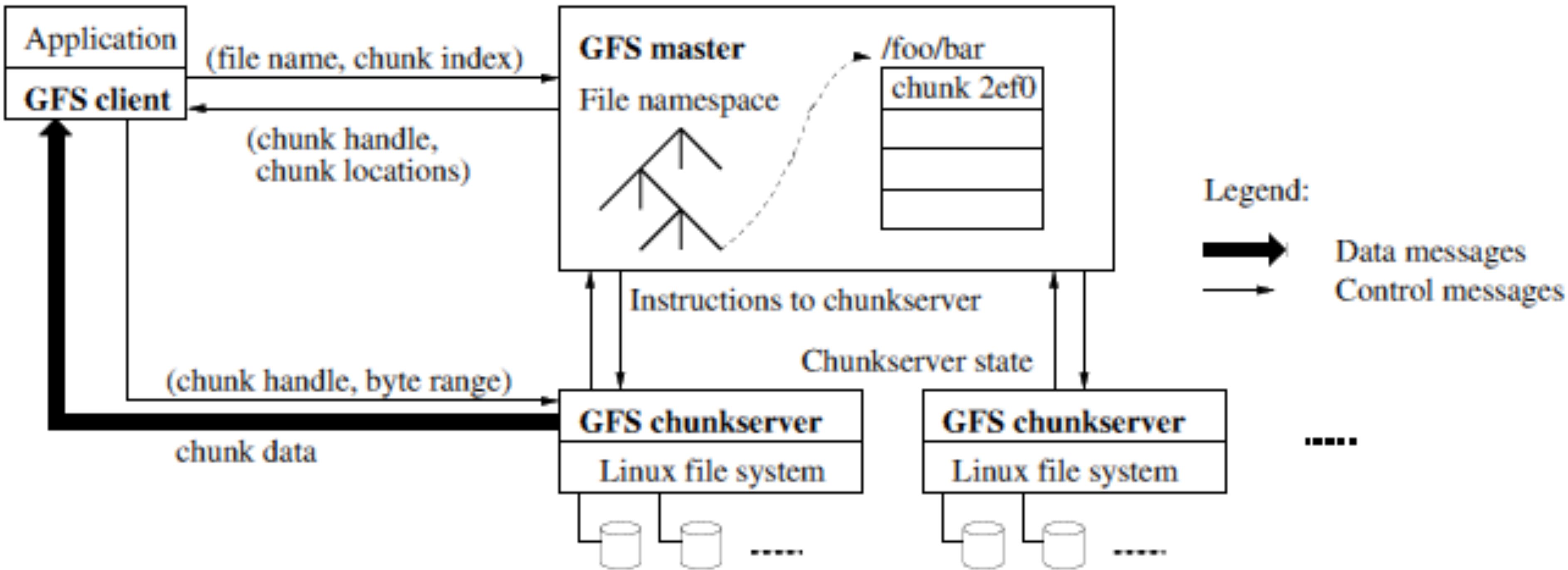
Google File System (GFS)

- System is so large that failures are the norm
- Files are very big (multi-GB is the norm)
- Files are modified by “appending” and read usually sequentially

GFS Assumptions

- System is built of commodity components that often fail
- System stores a few million files that are 100MB or longer
- Operations consist of large streaming reads and small random reads
- Most writes are large appends
- Large sustained bandwidth is more important than latency

GFS Architecture



- Master, chunk servers and clients
- Chunks (64MB) are replicated on multiple servers
- No file caching
- Clients cache metadata from master

Single Master

- General disadvantages for distributed systems:
  - Single point of failure
  - Bottleneck (scalability)
- Solution?
  - Clients use master only for metadata, not reading/ writing.
  - New master recovers state from chunk servers

Chunk Size

- Key design parameter: chose 64 MB.
- Each chunk is a plain Linux file, extended as needed.
  - Fragmentation: Internal vs. external?
- Hotspots: Some files may be accessed too much.
  - How to deal with it?

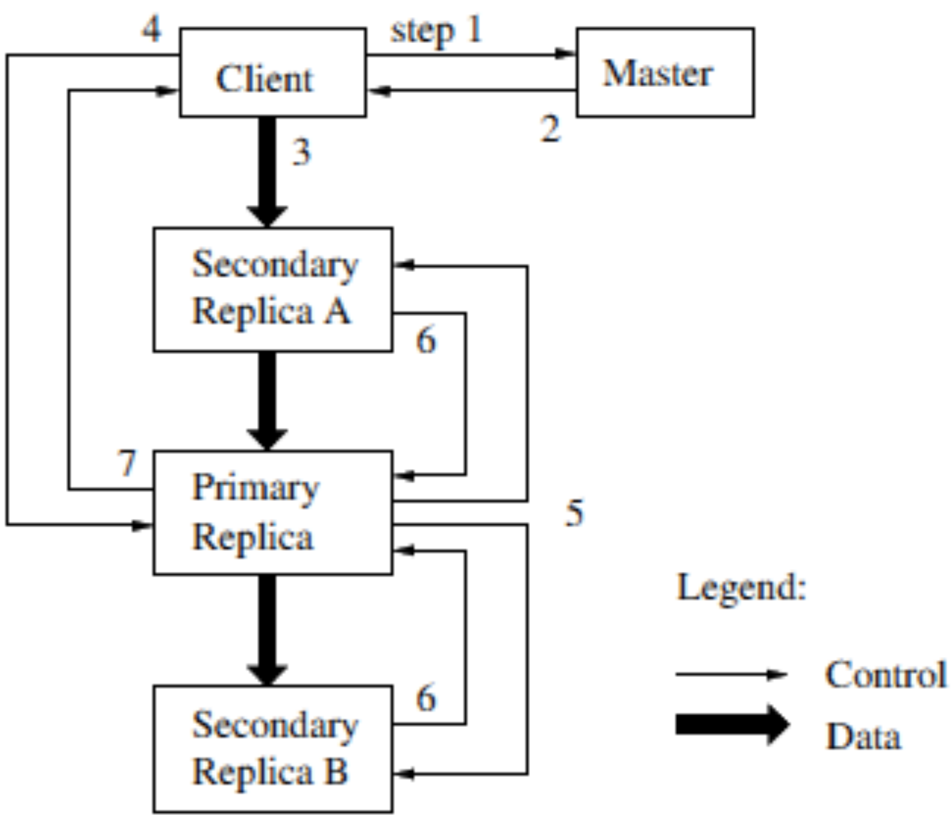
Leases

- Master grants “lease” to primary replica of each chunk
- Primary replica decides order of updates to chunk
- Lease is given for 60 seconds, renewable as needed
- Lease can be revoked by master
  - What happens if master loses communication with primary replica?

The "Append" Operation

- Concurrent "append" operations are allowed but the order they are applied is up to the GFS
- All chunk replicas are appended/modified in the same order as decided by the primary replica

Append



(1) client requests location of primary and secondary replicas from master

(2) master replies

(3) client sends data to replicas and receives acks

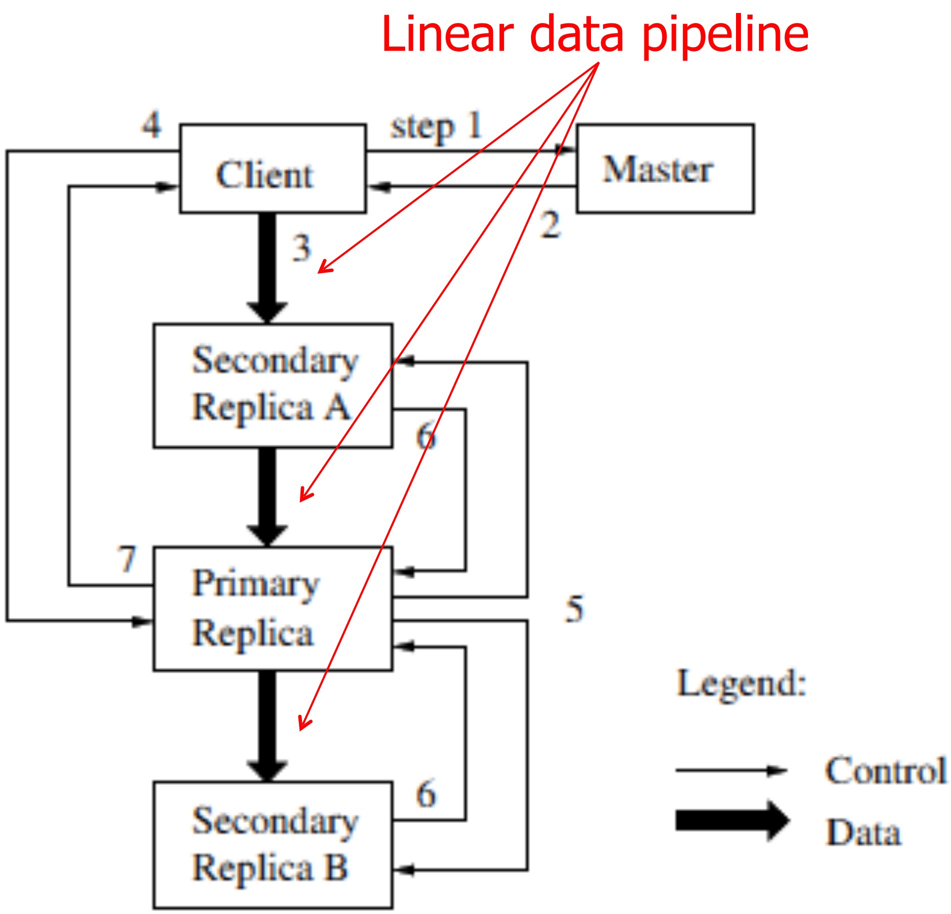
(4) client asks primary replica to do the "append" on the data

(5) primary replica decides on order of appends and asks secondary replicas to follow same order

(6) secondary replicas perform operation and ack primary

(7) primary replies to client

Append



(1) client requests location of primary and secondary replicas from master

(2) master replies

(3) client sends data to replicas and receives acks

(4) client asks primary replica to do the "append" on the data

(5) primary replica decides on order of appends and asks secondary replicas to follow same order

(6) secondary replicas perform operation and ack primary

(7) primary replies to client

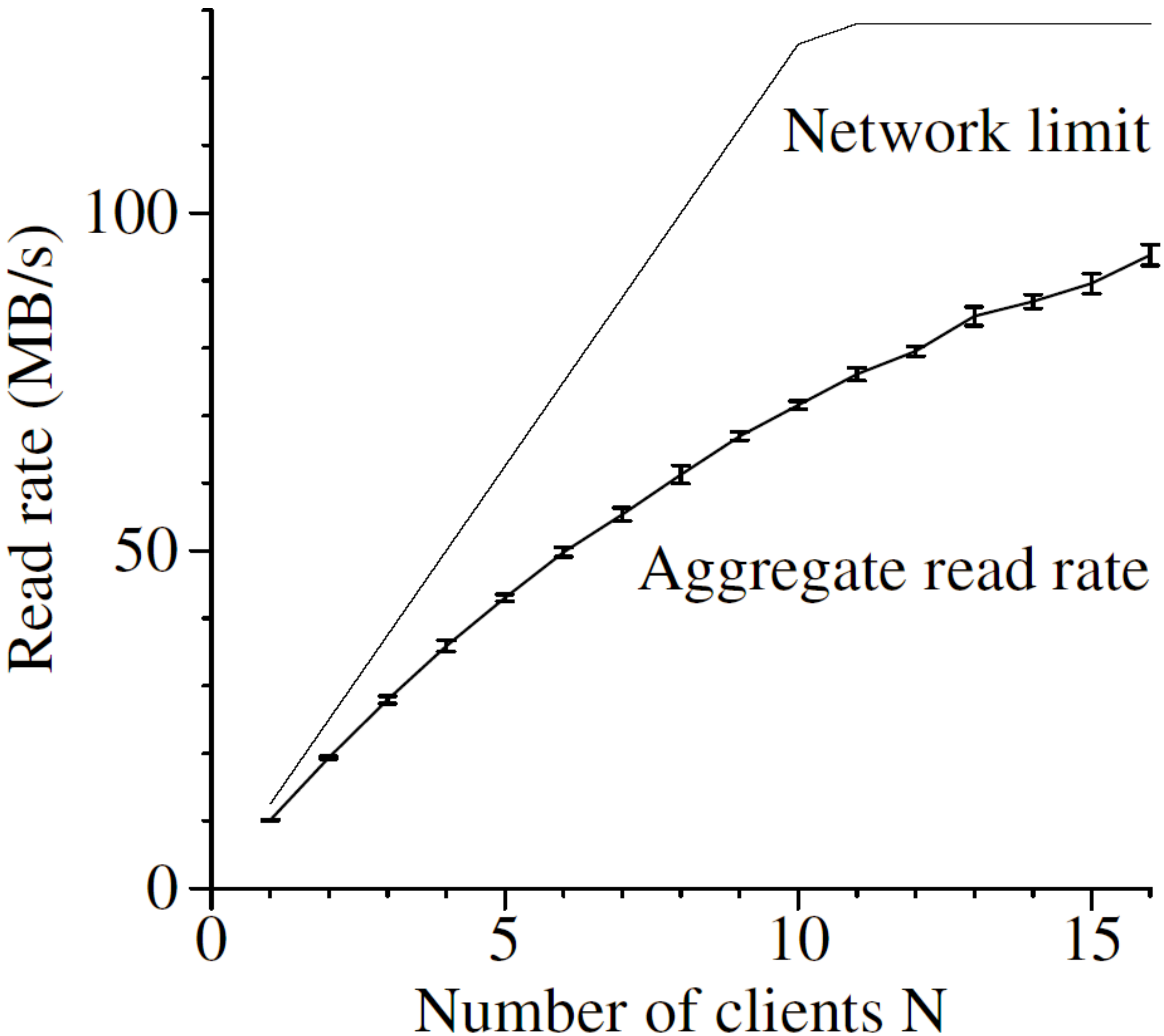
File Copy

- Master received copy (snapshot) request and revokes all leases on its chunks
  - Why revoke?
  - What if it loses communication with a chunk server?
- After all leases are revoked or expire, master duplicates metadata (pointing to the same chunks as original) and increments reference count – no actual data copy is performed
- When a client wants to write to a chunk, the request for primary comes to master who notices that the chunk has a reference count of two and asks chunk servers to replicate it
- A handle is returned to the new chunk copy.

Microbenchmarks

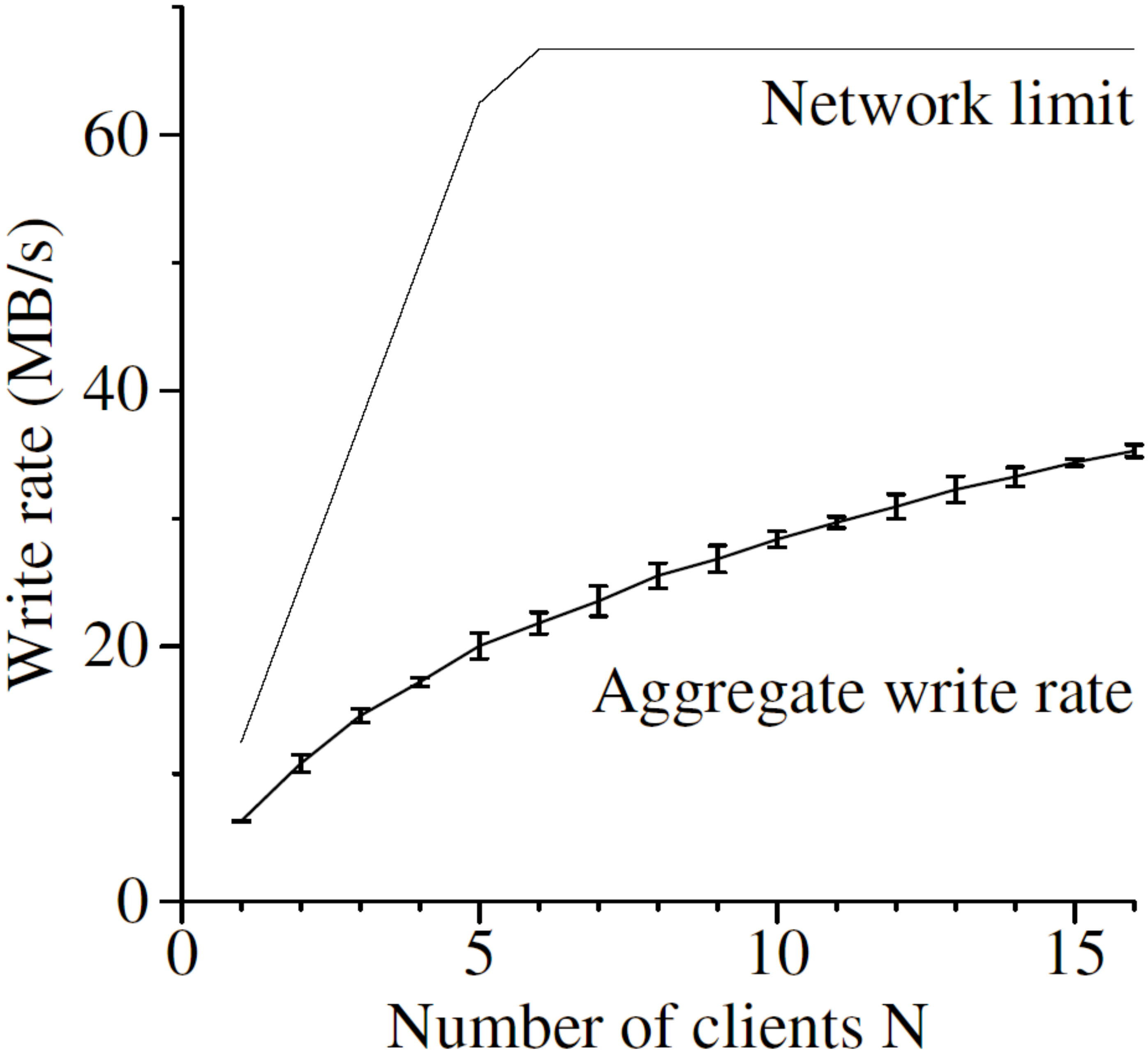
- GFS cluster consisting of:
  - One master
    - Two master replicas
  - 16 chunkservers
  - 16 clients
- Machines were:
  - Dual 1.4 GHz PIII
  - 2 GB of RAM
  - 2 80 GB 5400 RPM disks
  - 100 Mbps full-duplex Ethernet to switch
  - Servers to one switch, clients to another. Switches connected via gigabit Ethernet.

Reads



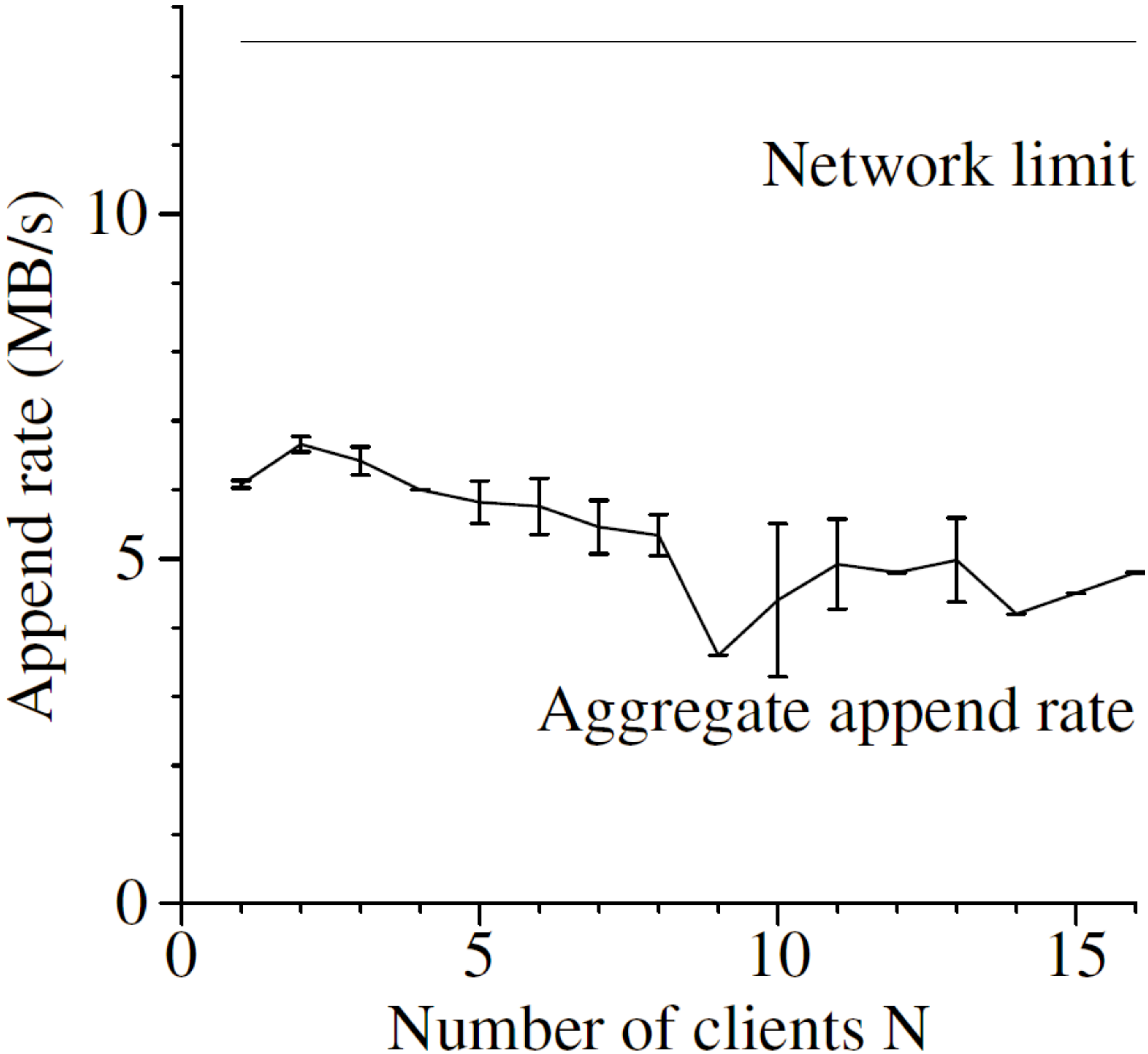
- N clients reading 4 MB region from 320 GB file set.
- Read rate slightly lower as clients go up due to probability reading from same chunkserver.

Writes



- N clients write simultaneously to N files. each client writes 1 GB to a new file in a series.
- Low performance is due to network stack.

Appends



- N clients appending to a single file.