416 Distributed Systems

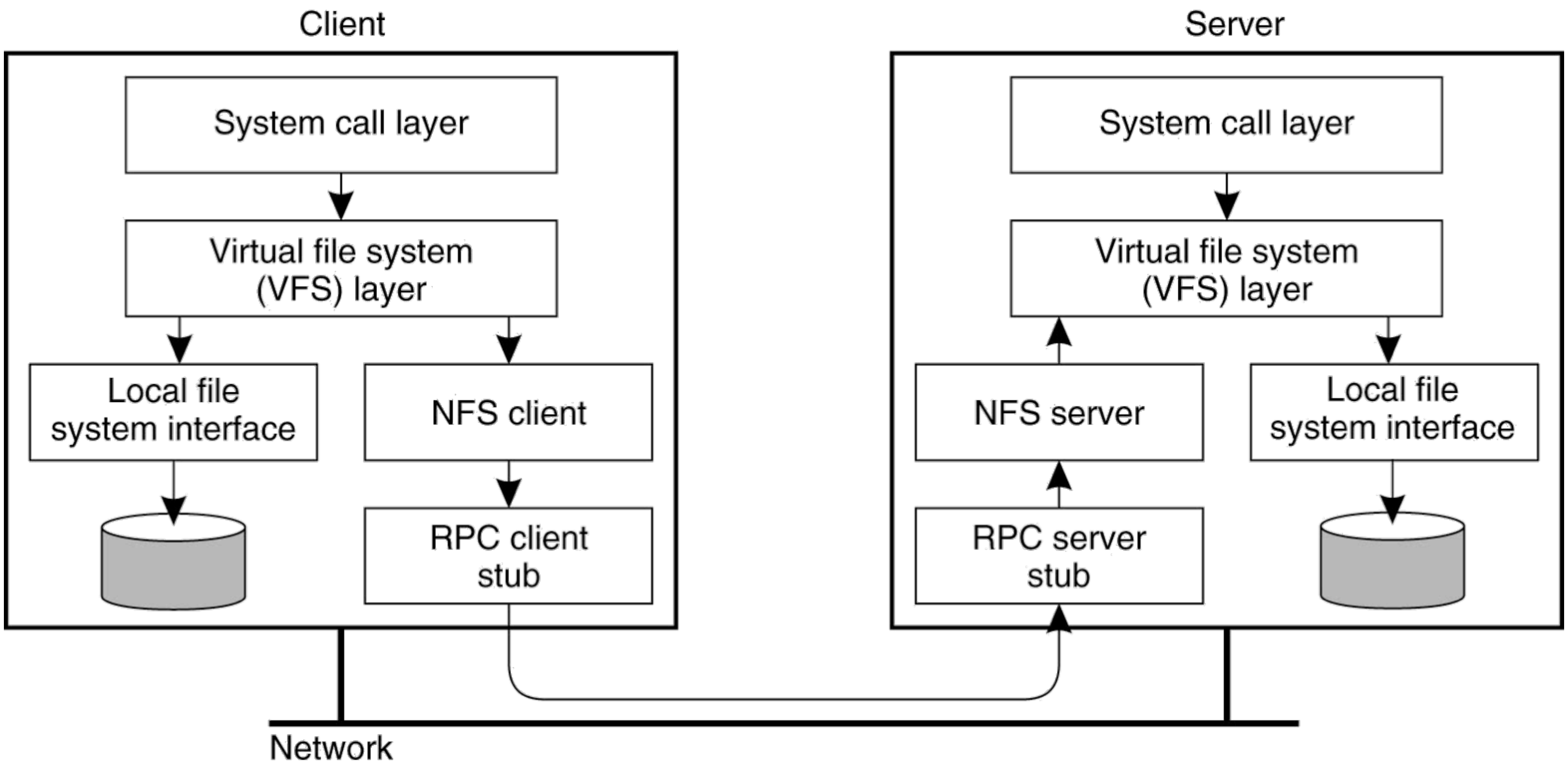
Distributed File Systems 2

Jan 20, 2016

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
    - NFS: network file system
    - AFS: andrew file system
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

VFS Interception



A Simple Approach

- Use RPC to forward every filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.
- Great: Same behavior as if both programs were running on the same local filesystem!
- Bad: Performance can stink. Latency of access to remote server often much higher than to local memory.
- For AFS context: bad bad bad: server would get hammered!

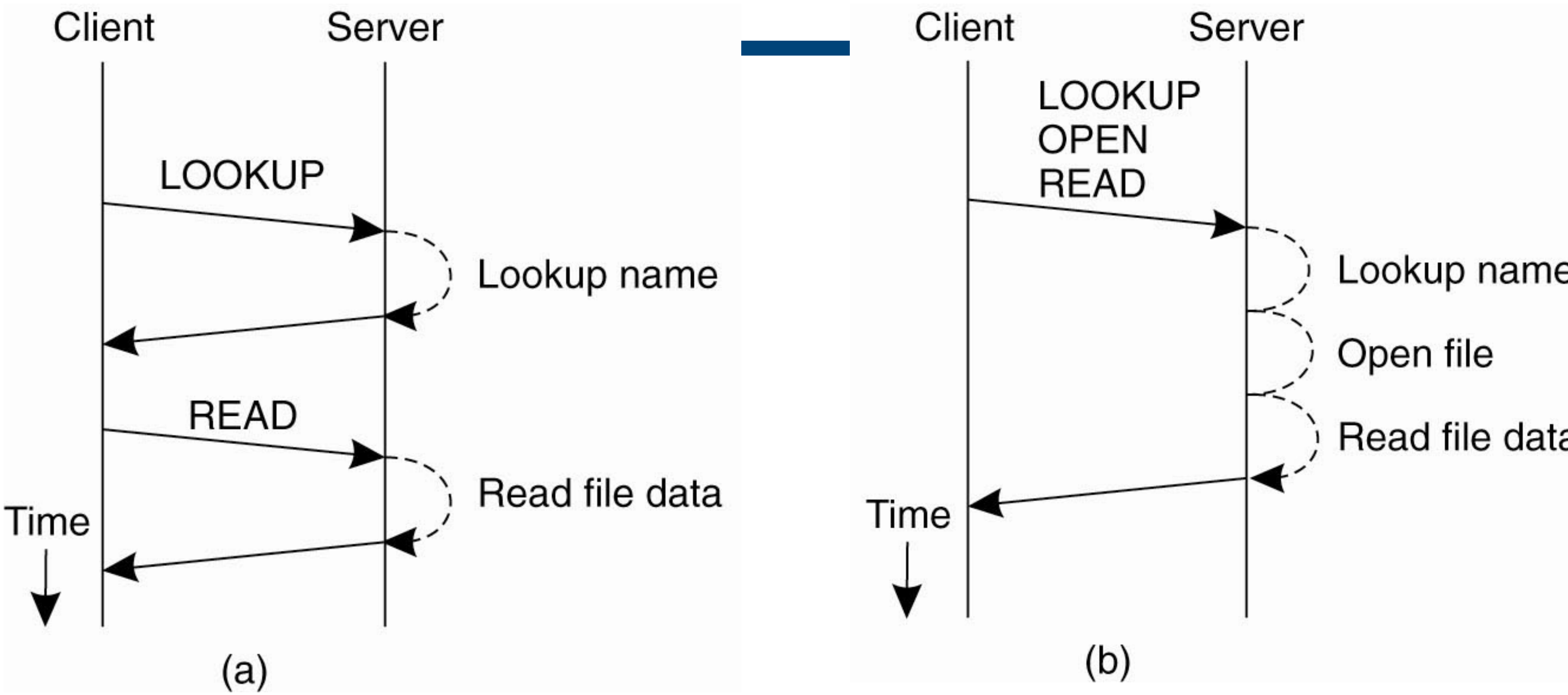
Lesson 1: Needing to hit the server for every detail impairs performance and scalability.

Question 1: How can we avoid going to the server for everything? *What* can we avoid this for? What do we lose in the process?

NFS V2 Design

- “Dumb”, “Stateless” servers w/ smart clients
- Portable across different OSes
- Low implementation cost
- Small number of clients
- Single administrative domain

Remote Procedure Calls in NFS



- (a) Reading data from a file in NFS version 3
- (b) Reading data using a compound procedure in version 4.

Outline

- Why Distributed File Systems?
- Basic mechanisms for building DFSs
  - Using NFS and AFS as examples
- Design choices and their implications
  - Caching
  - Consistency
  - Naming
  - Authentication and Access Control

Topic 1: Client-Side Caching

- Many systems (not just distributed!) rely on two solutions to every problem:
  1. Cache it!
  2. “All problems in computer science can be solved by adding another level of indirection. But that will usually create another problem.” -- David Wheeler

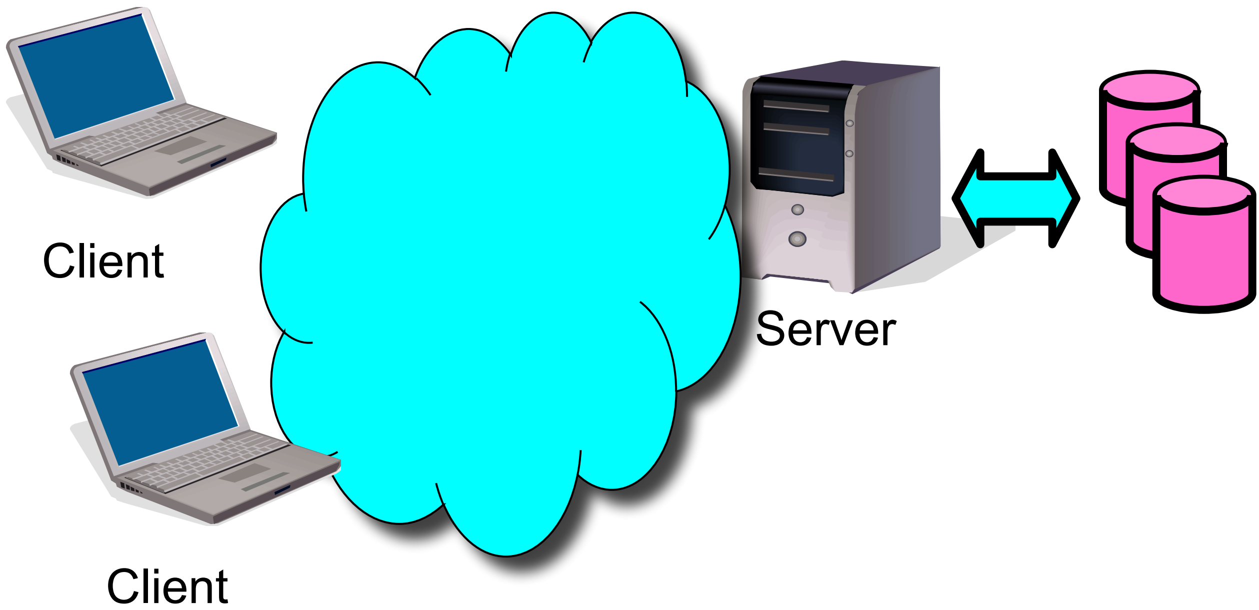
Client-Side Caching

- So, uh, what do we cache?
  - Read-only file data and directory data → easy
  - Data written by the client machine → when is data written to the server? What happens if the client machine goes down?
  - Data that is written by other machines → how to know that the data has changed? How to ensure data consistency?
  - Is there any pre-fetching?
- And if we cache... doesn’t that risk making things inconsistent?

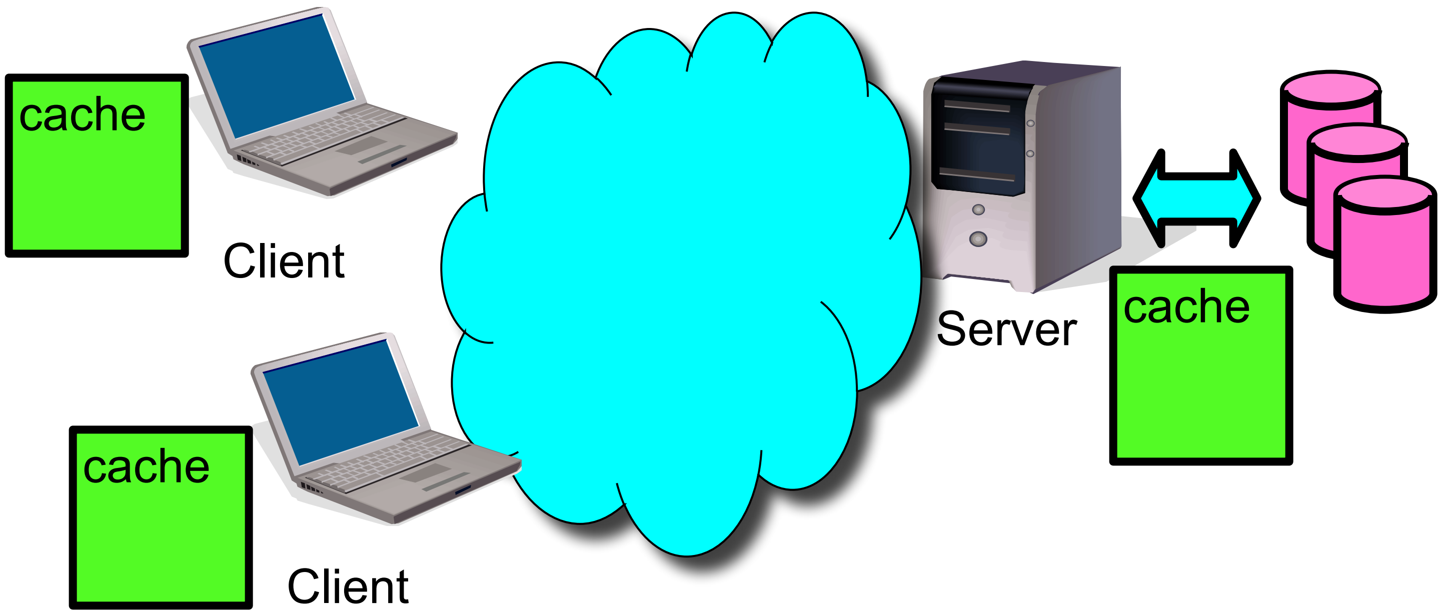
Failures

- Server crashes
  - Data in memory but not disk lost
  - So... what if client does
    - seek() ; /* SERVER CRASH */; read()
    - If server maintains file position, this will fail (Why?). Ditto for open(), read()
- Lost messages: what if we lose acknowledgement for delete("foo")
  - And in the meantime, another client created foo anew?
- Client crashes
  - Might lose data in client cache

Use of caching to reduce network load

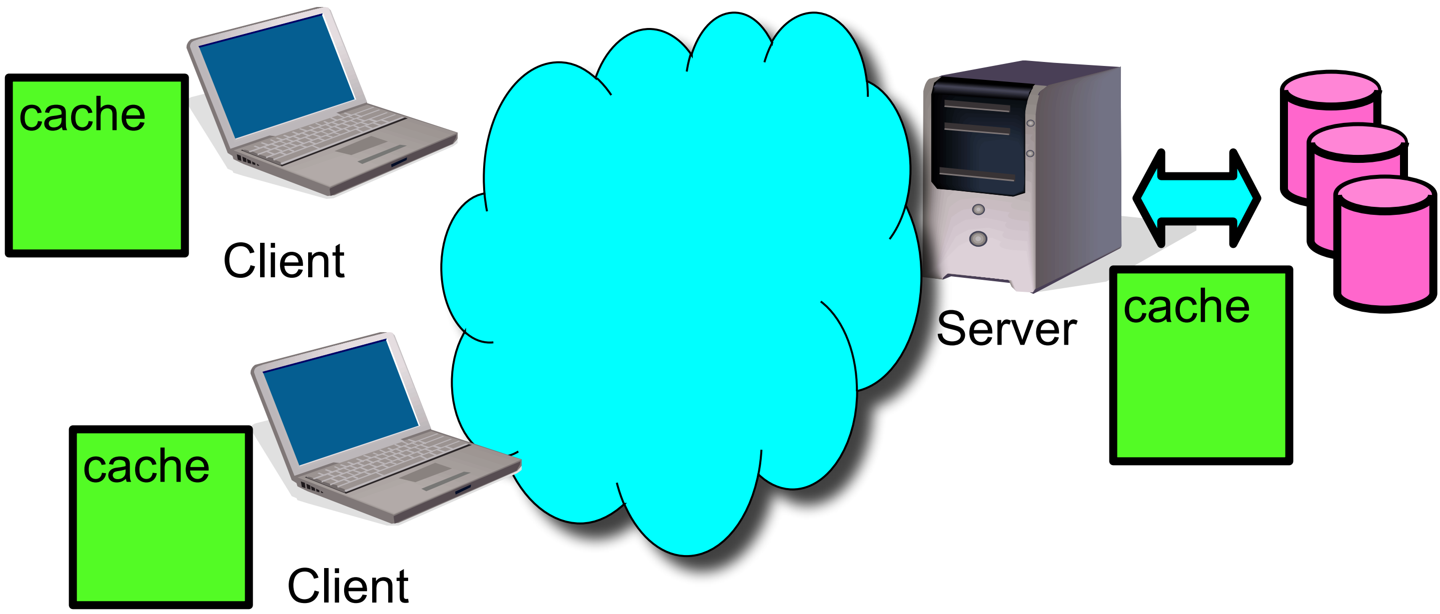


Use of caching to reduce network load



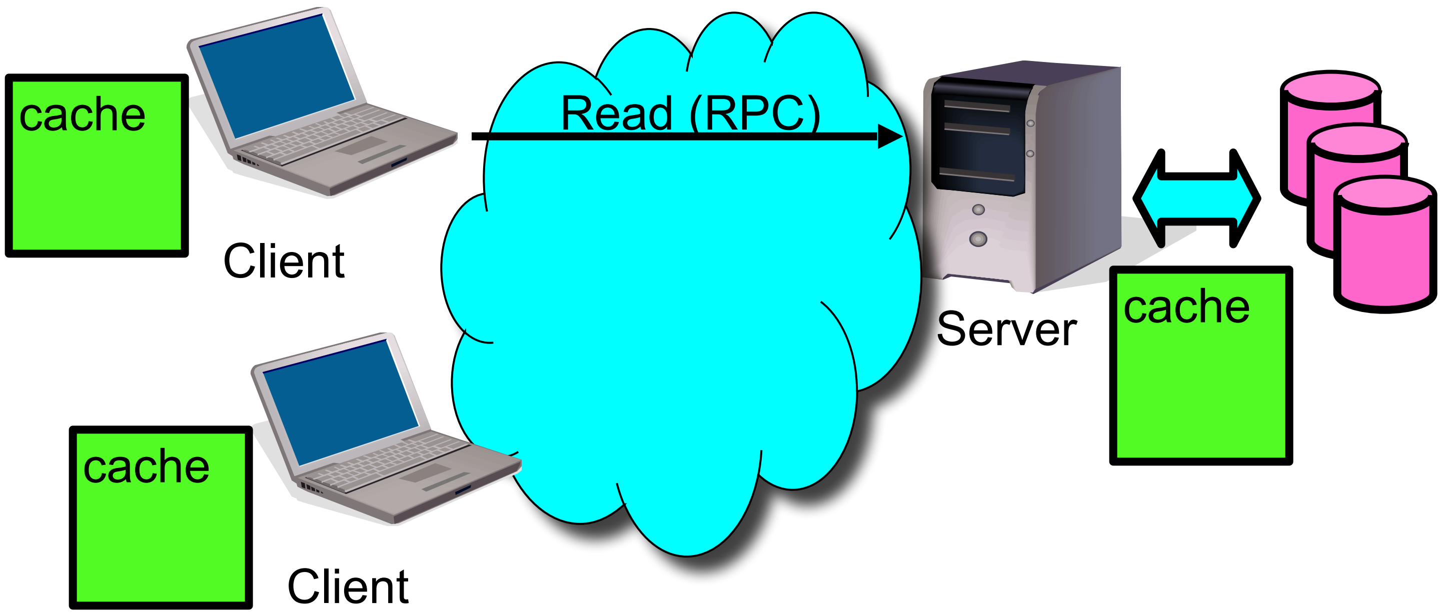
Use of caching to reduce network load

read(f1)



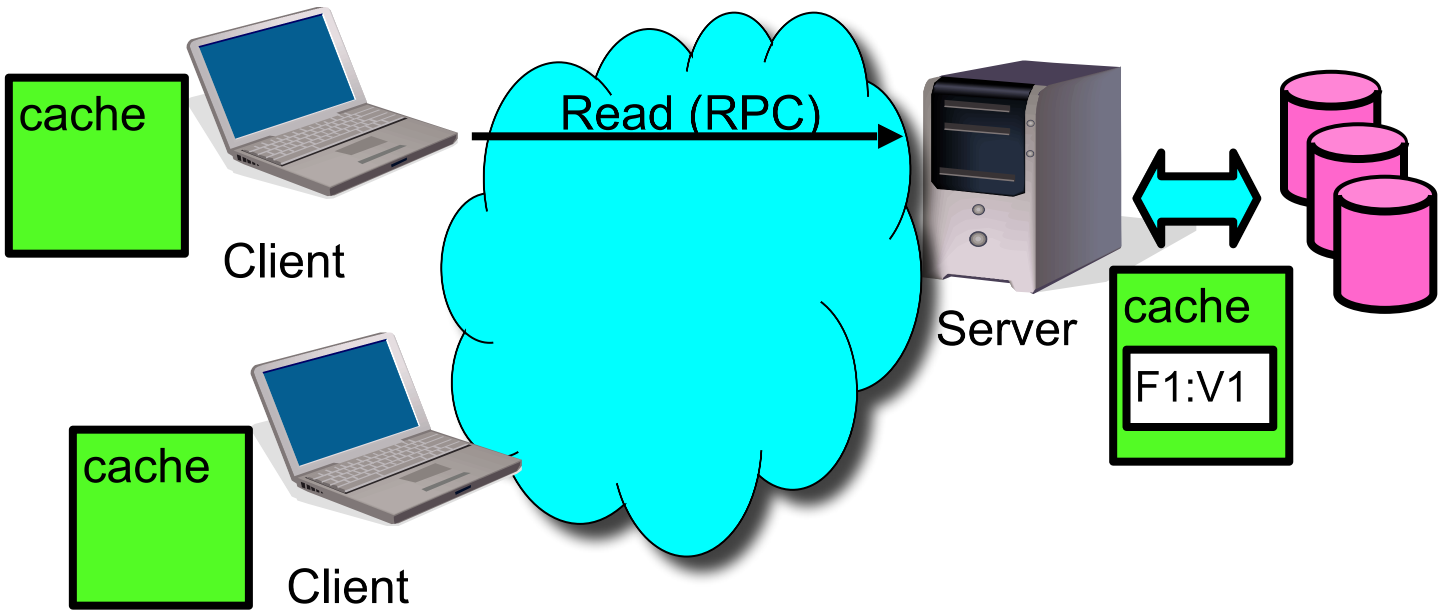
Use of caching to reduce network load

read(f1)



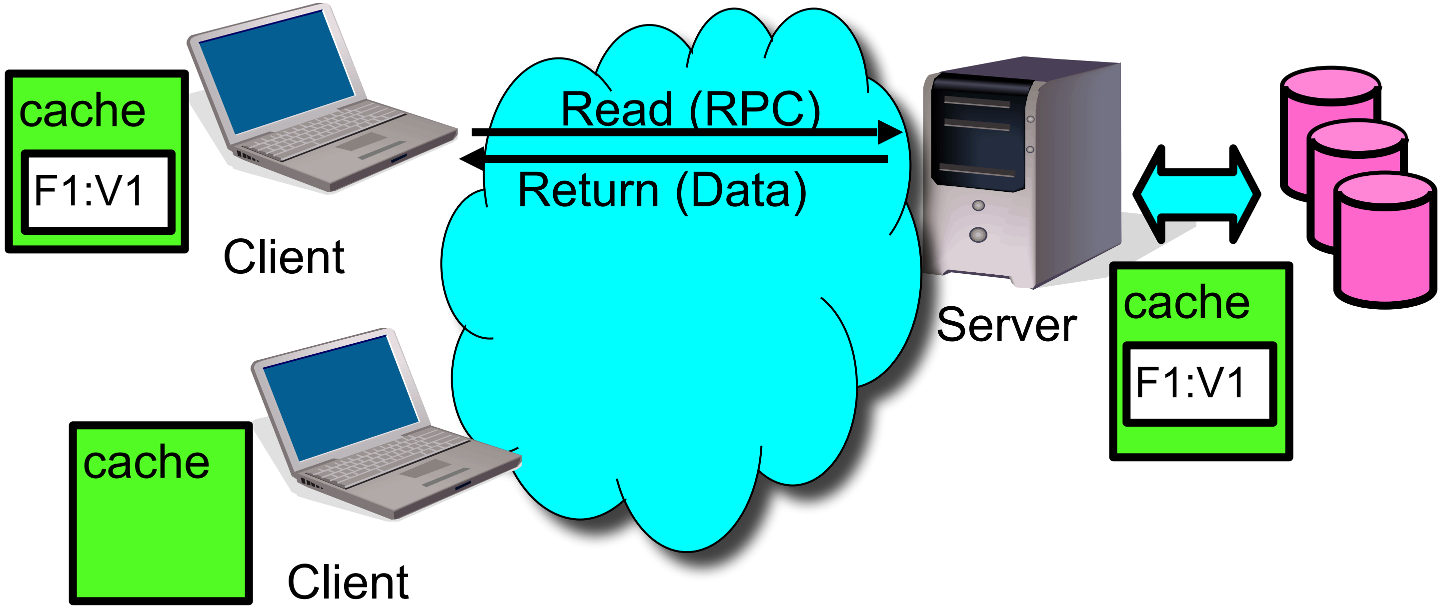
Use of caching to reduce network load

read(f1)



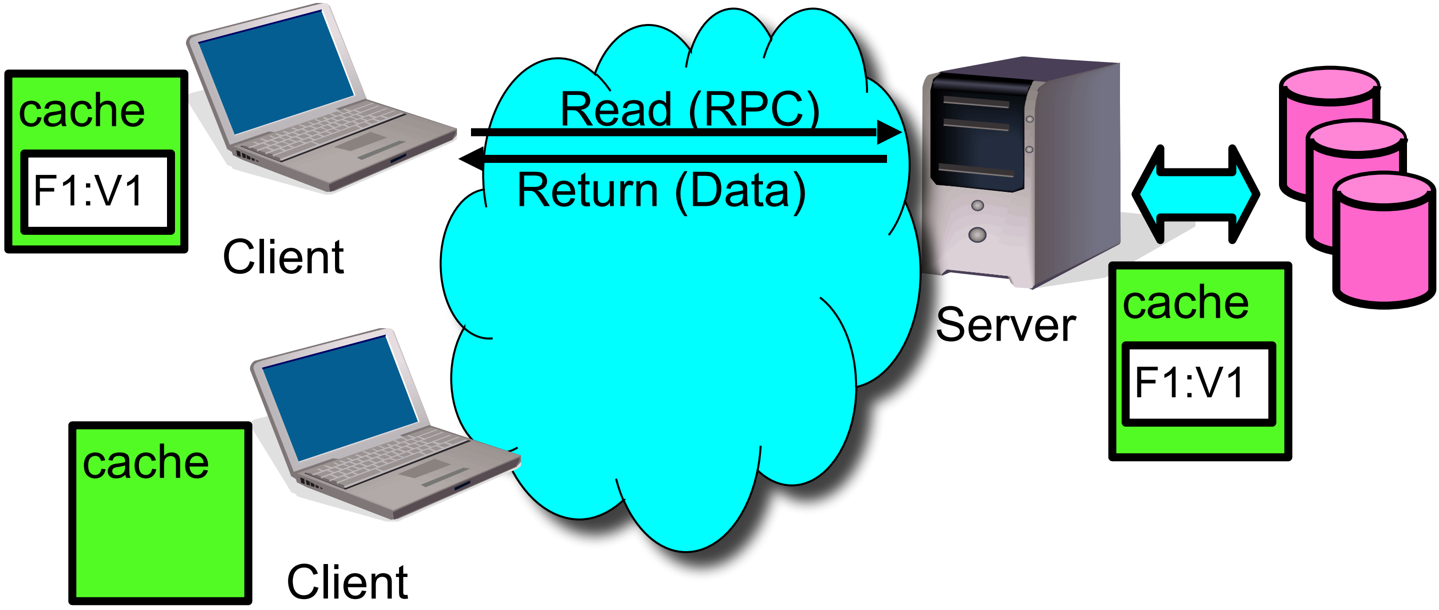
Use of caching to reduce network load

read(f1)



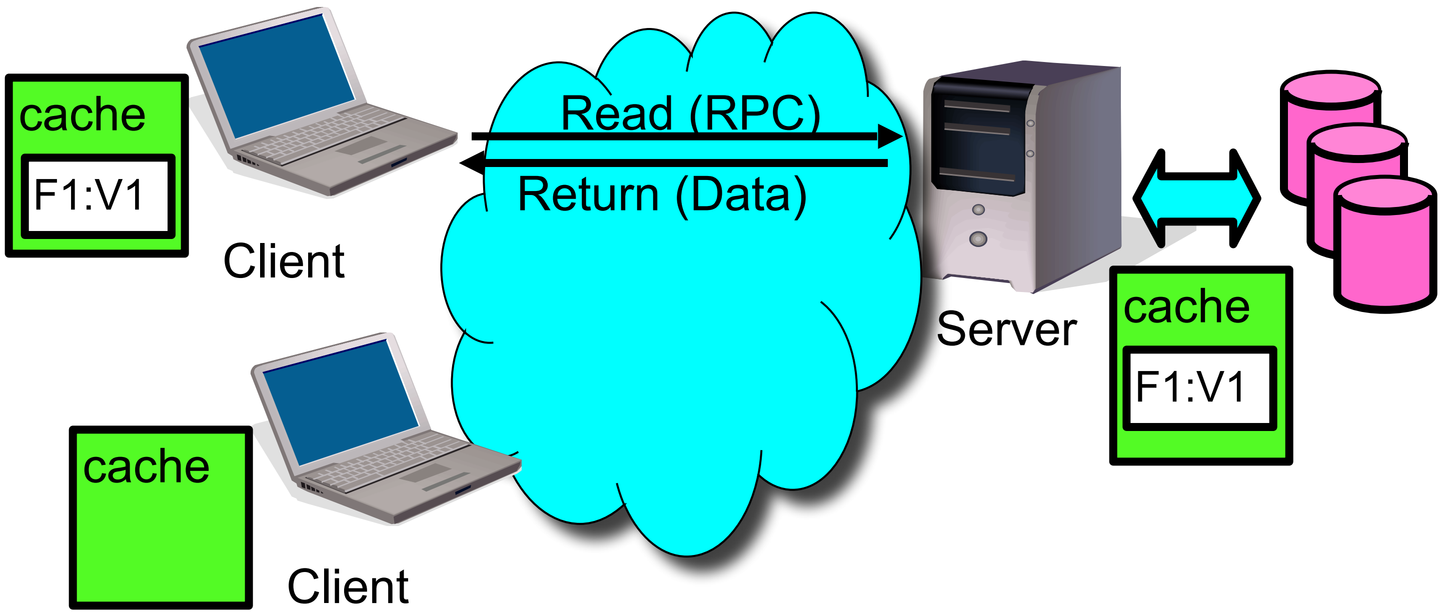
Use of caching to reduce network load

read(f1)→V1



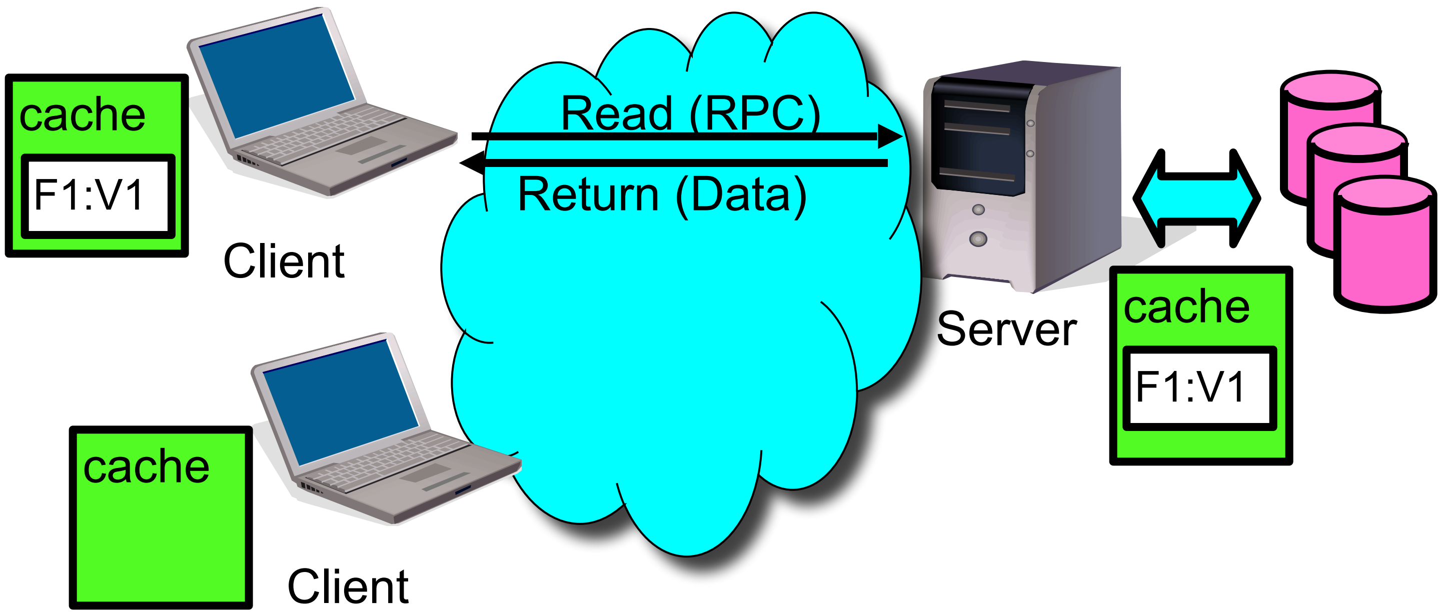
Use of caching to reduce network load

read(f1)→V1
read(f1)→V1



Use of caching to reduce network load

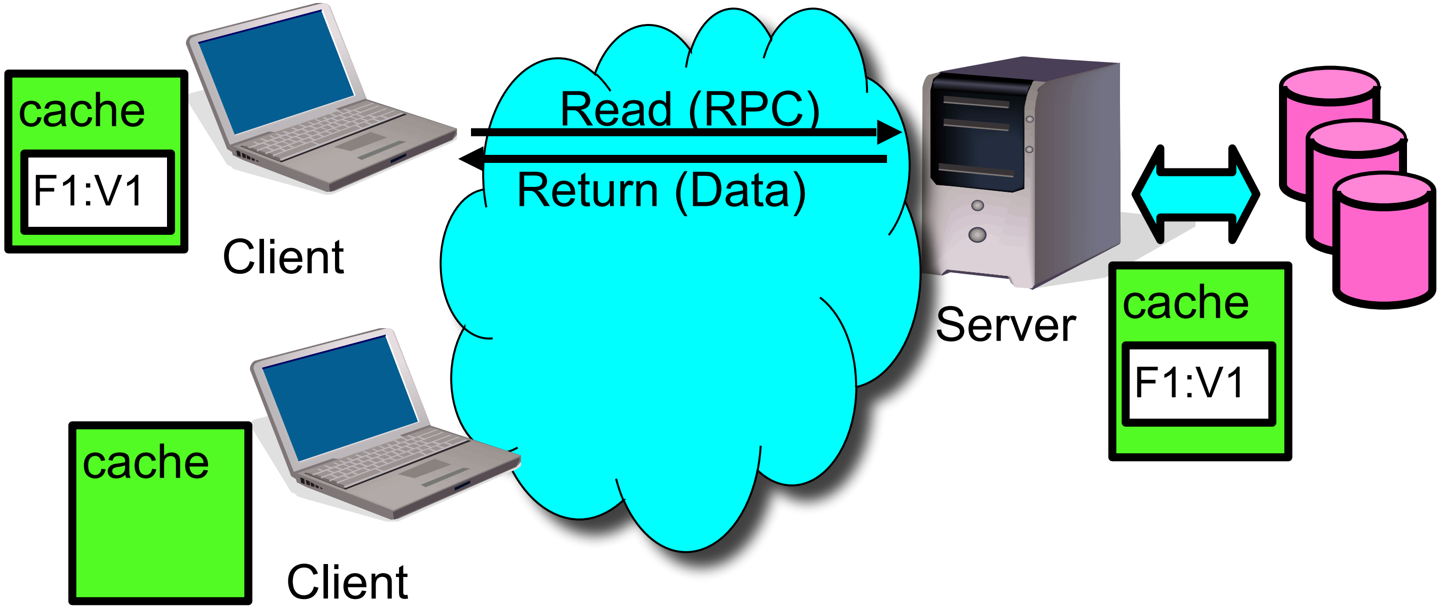
read(f1)→V1
read(f1)→V1
read(f1)→V1



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

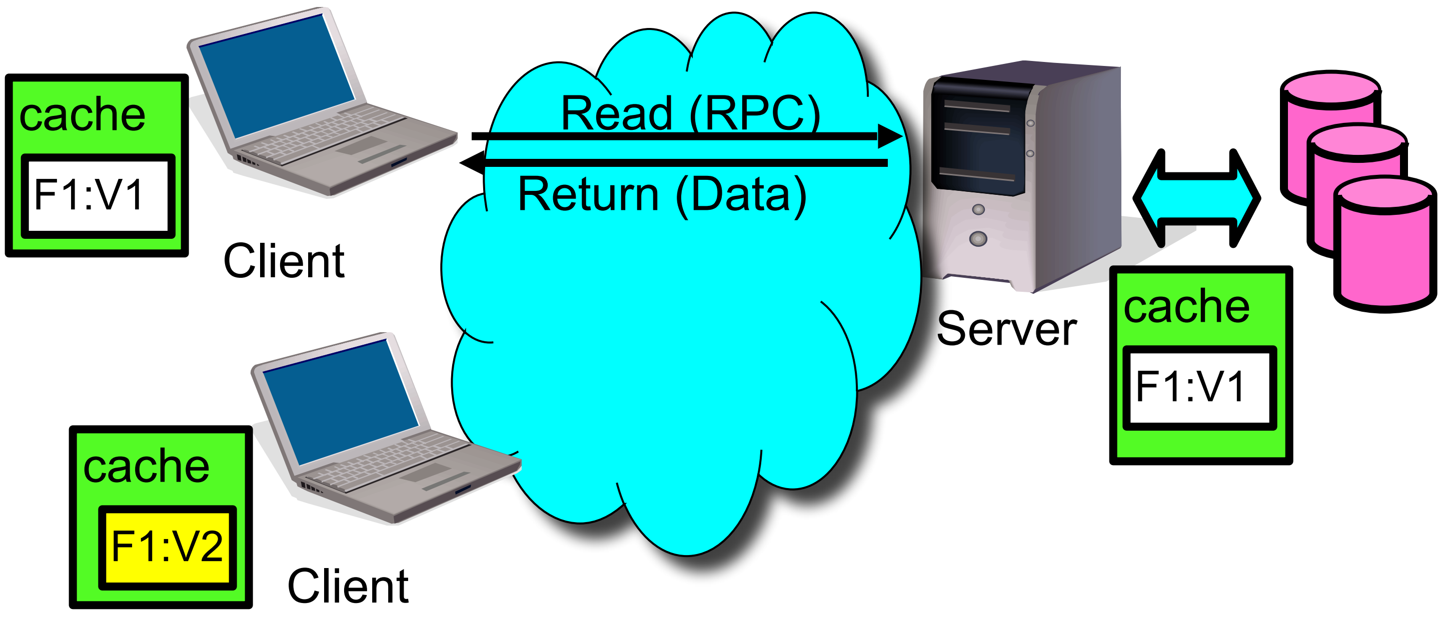
write(f1)



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

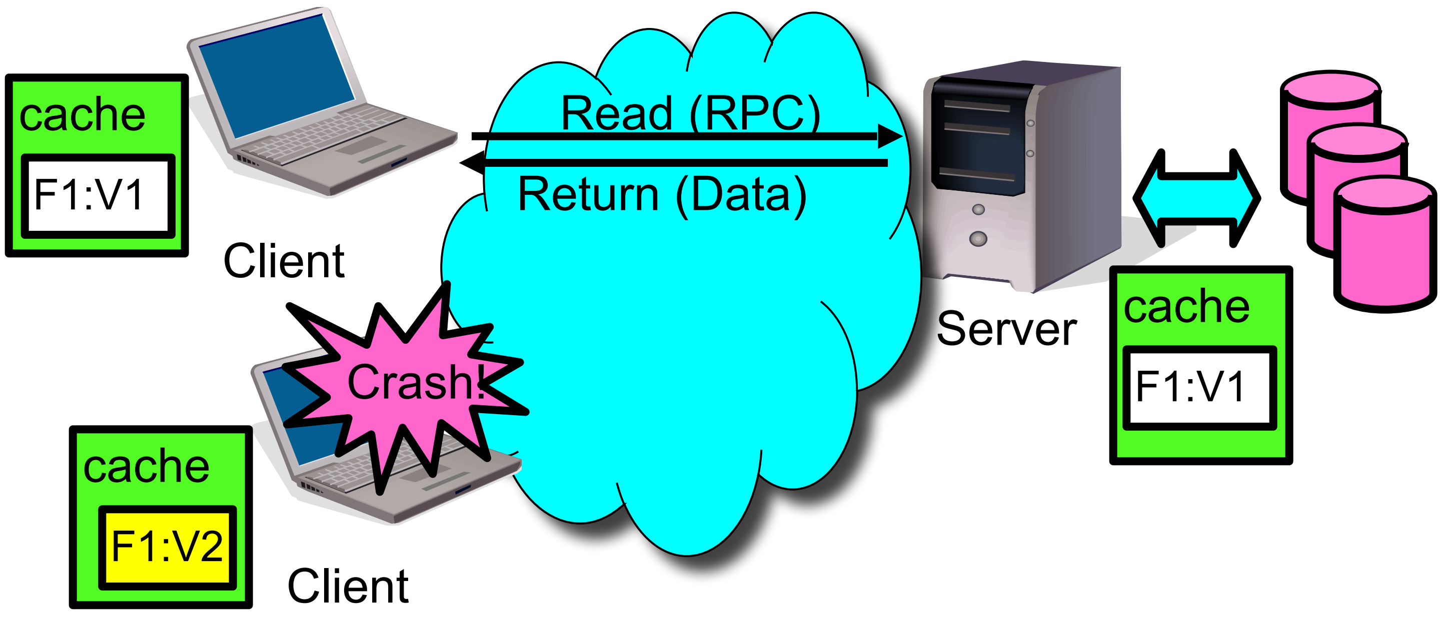
write(f1)



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

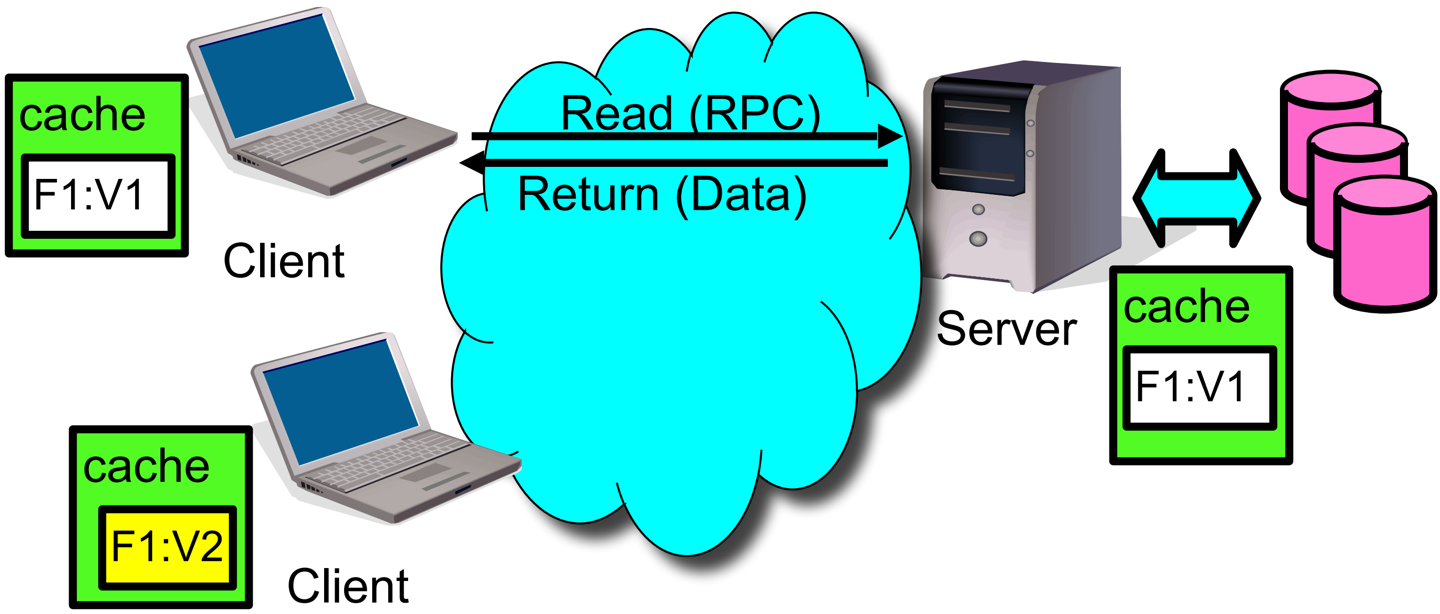
write(f1)



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

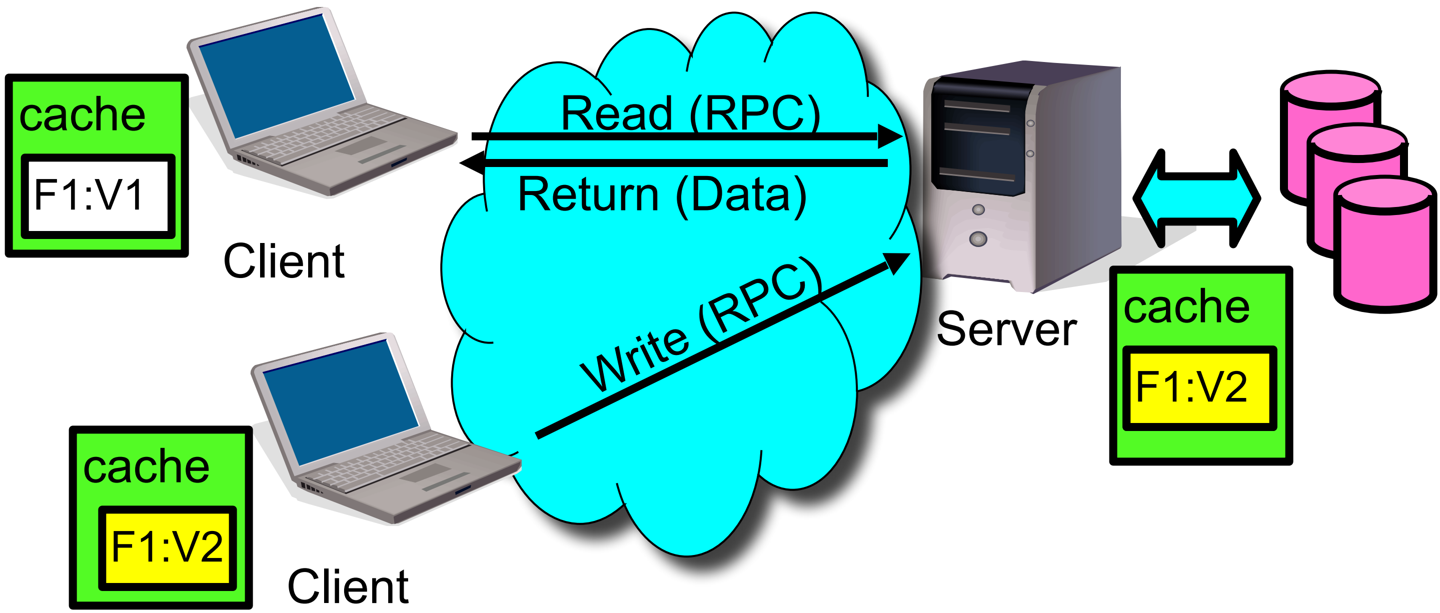
write(f1)



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

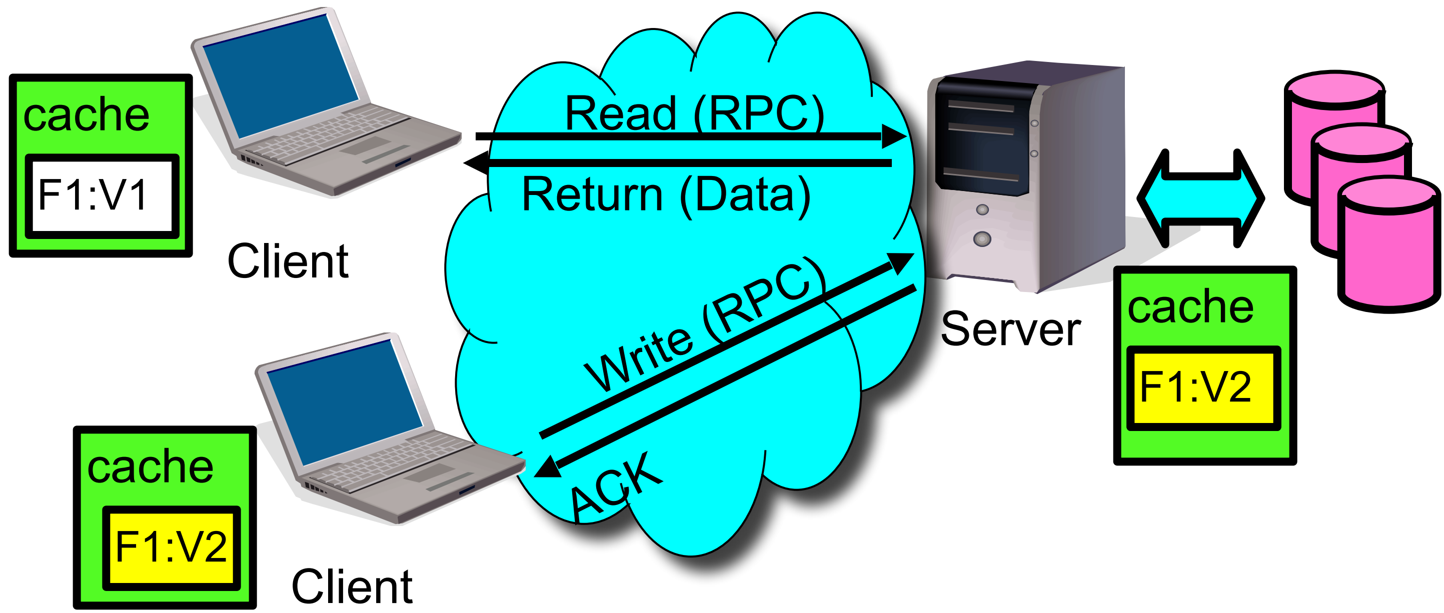
write(f1)



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

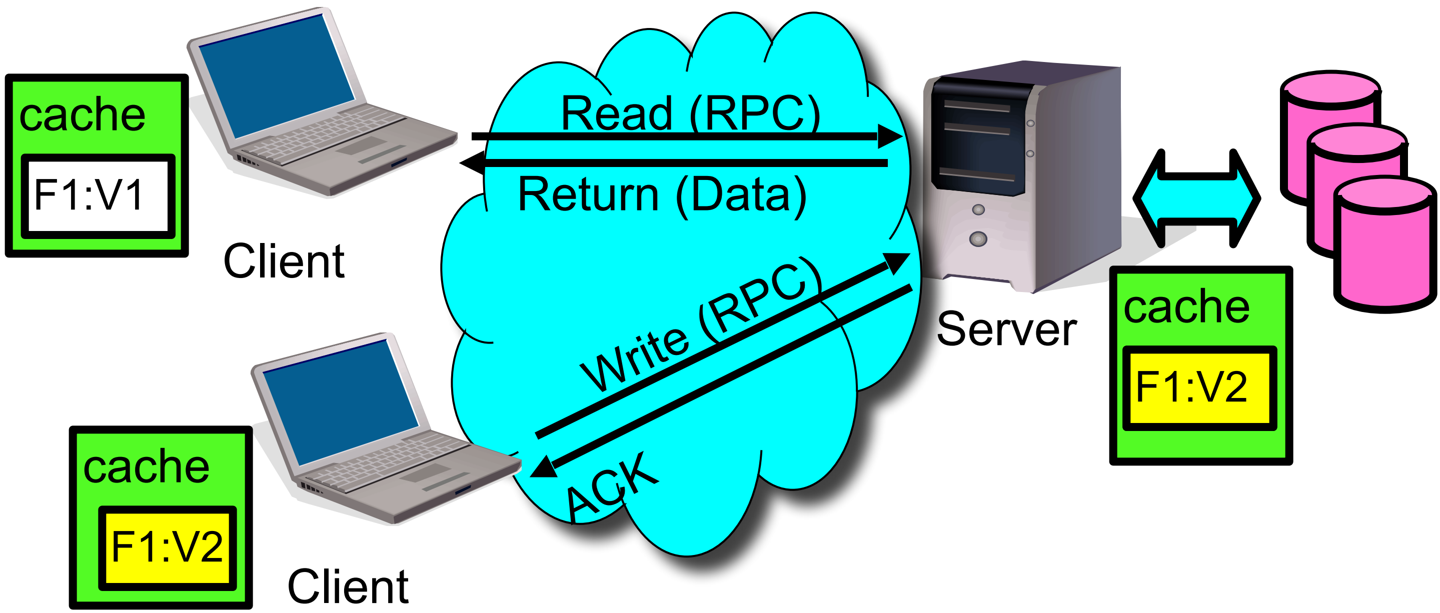
write(f1)



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

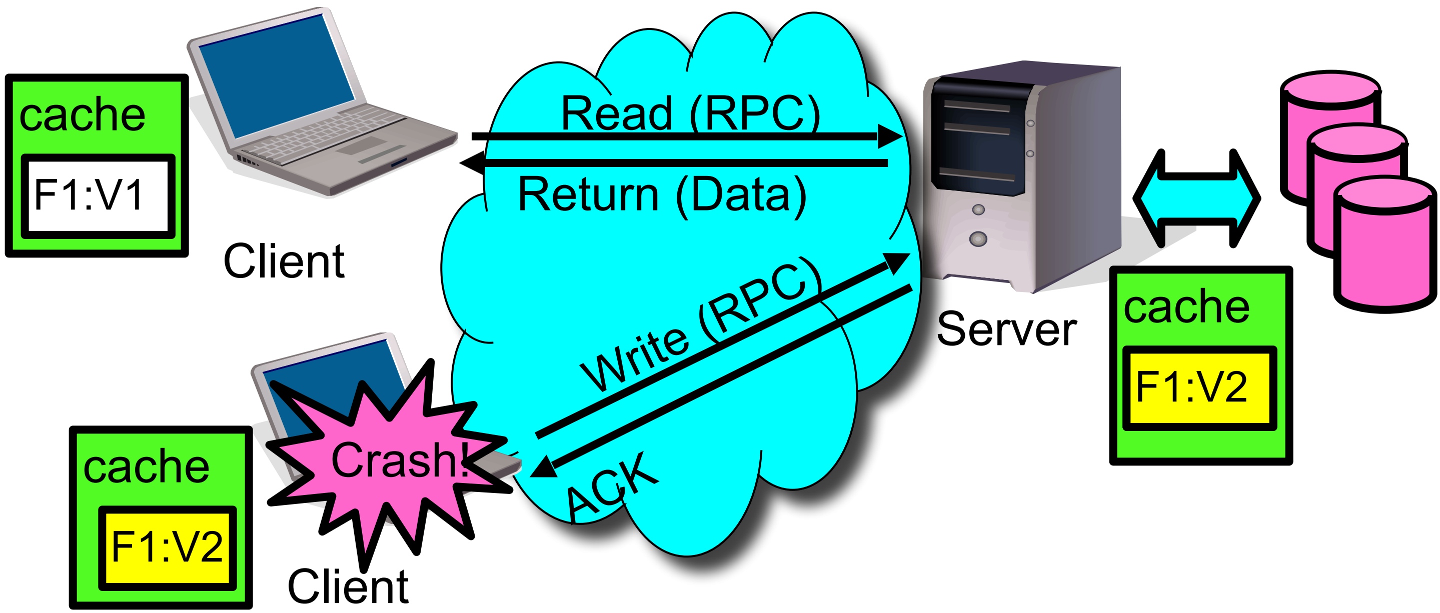
write(f1)→OK



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

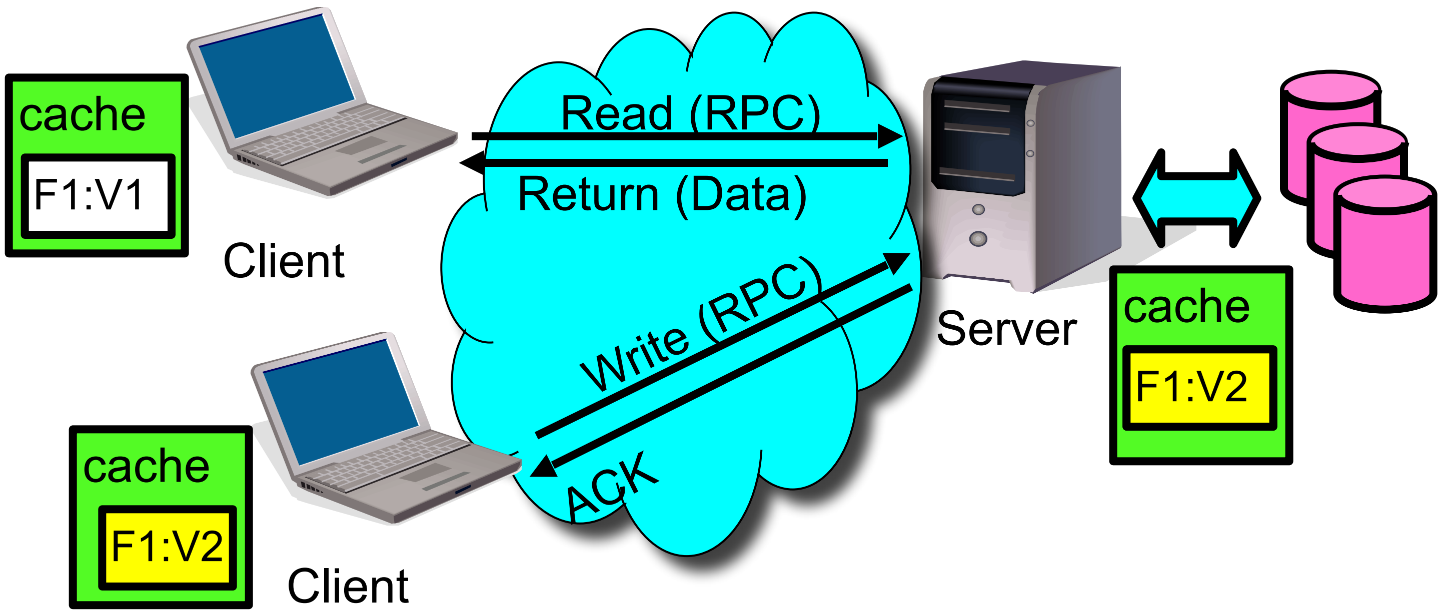
write(f1)→OK



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1

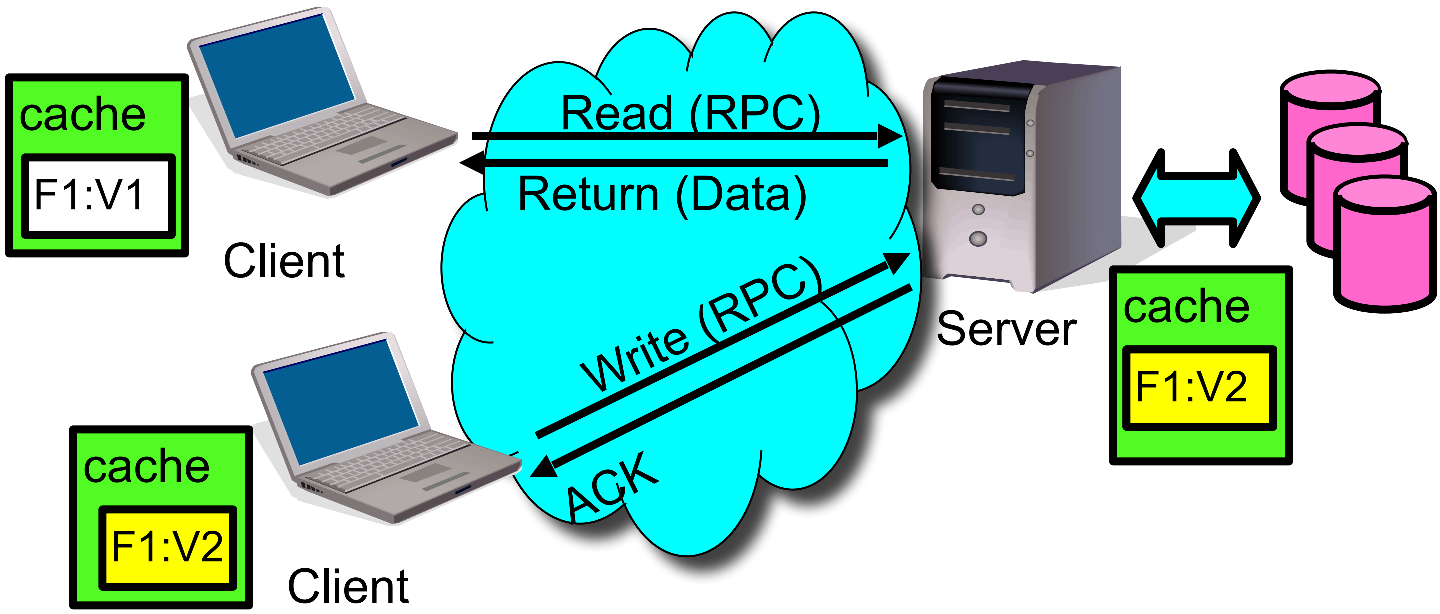
write(f1)→OK



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1
read(f1)→V1

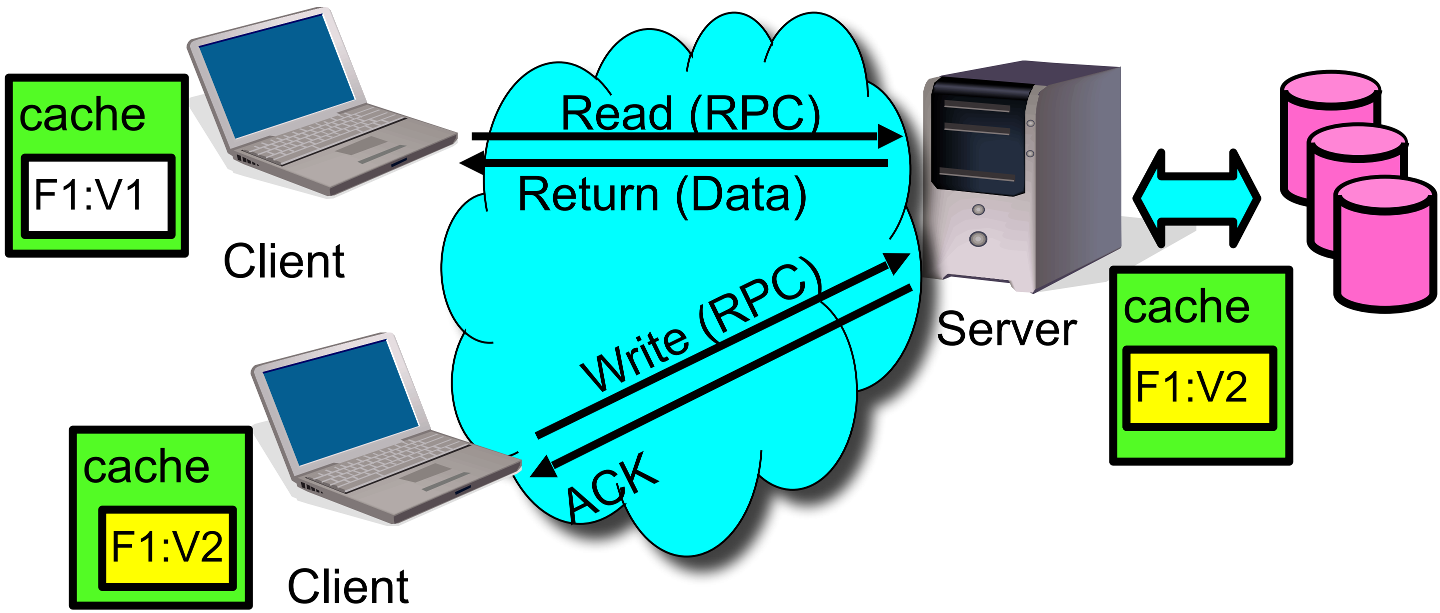
write(f1)→OK



Use of caching to reduce network load

read(f1)→V1
read(f1)→V1
read(f1)→V1
read(f1)→V1

write(f1)→OK
read(f1)→V2



Client Caching in NFS v2

- Cache both clean and dirty file data and file attributes
  - Memory (e.g., DRAM) cache
- File attributes in the client cache expire after 60 seconds (file data doesn't expire)
- File data is checked against the modified-time in file attributes (which could be a cached copy)
  - Changes made on one machine can take up to 60 seconds to be reflected on another machine
- Dirty data are buffered on the client machine until file close or up to 30 seconds (Why 30s?)
  - If the machine crashes before then, the changes are lost

Implication of NFS v2 Client Caching

- Advantage: No network traffic if open/read/write/close can be done locally.
- But…. Data consistency guarantee is very poor
  - Simply unacceptable for some distributed applications
  - Productivity apps tend to tolerate such loose consistency
- Generally clients do not cache data on local disks

NFS’s Failure Handling – Stateless Server

- Files are state, but...
- Server exports files without creating extra state
  - No list of “who has this file open” (permission check on each operation on open file!)
  - No “pending transactions” across crash
- Crash recovery is “fast”
  - Reboot, let clients figure out what happened
- State stashed elsewhere
  - Separate MOUNT protocol
  - Separate NLM locking protocol
- Stateless protocol: requests specify exact state. read() → read([position]). no seek on server.

NFS’s Failure Handling

- Operations are idempotent
  - How can we ensure this?

NFS's Failure Handling

- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories. It's not delete("foo"), it's delete(1337f00f), where that ID won't be reused (e.g., by same/other clients)

NFS’s Failure Handling

- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories. It’s not delete(“foo”), it’s delete(1337f00f), where that ID won’t be reused.
- Write-through caching: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
  - Close failures?
    - retry until things get through to the server
    - return failure to client
  - Most client apps can’t handle failure of close() call.
  - Usual option: hang for a long time trying to contact server

NFS Results

- NFS provides transparent, remote file access
- Simple, portable, *really popular*
  - (it’s gotten a little more complex over time, but...)
- Weak consistency semantics
- Requires hefty server resources to scale (write-through, server queried for lots of operations)

- Design take-away: smart server; dumb clients

AFS Goals

- Global distributed file system
  - “One AFS”, like “one Internet”
    - Why would you want more than one?
- **LARGE** numbers of clients, servers
  - 1000 machines could cache a single file,
  - Most local, some (very) remote
- Goal: O(0) work per client operation
  - O(1) may just be too expensive!

AFS Assumptions

- Client machines are un-trusted
  - Must **prove** they act for a specific user
    - Secure RPC layer
  - Anonymous “system:anyuser”
- Client machines have disks(!!)
  - Can cache whole files over long periods
- Write/write and write/read sharing are rare
  - Most files updated by one user, on one machine

Let’s look back at AFS

- NFS gets us partway there, but
  - Probably doesn’t handle scale (* - you can buy huge NFS appliances today that will, but they’re $$$-y).
  - Is very sensitive to network latency
- How can we improve this?
  - More aggressive caching (AFS caches on disk in addition to just in memory)
  - Prefetching (on open, AFS gets entire file from server, making later ops local & fast).
    - Remember: with traditional hard drives, large sequential reads are much faster than small random writes. So easier to support (client a: read whole file; client B: read whole file) than having them alternate. Improves scalability, particularly if client is going to read whole file anyway eventually.

Client Caching in AFS

- Callbacks! Clients register with server that they have a copy of file;
  - Server tells them: "Invalidate" if the file changed (but only does so on file close!)
  - This trades state for improved consistency
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from clients
    - ask everyone "who has which files cached?"

AFS v2 RPC Procedures

- Procedures that are not in NFS
  - Fetch: return status and optionally data of a file or directory, and place a callback on it
  - RemoveCallBack: specify a file that the client has flushed from the local machine
  - BreakCallBack: from server to client, revoke the callback on a file or directory
    - What should the client do if a callback is revoked?
  - Store: store the status and optionally data of a file
- Rest are similar to NFS calls

Topic 2: File Access Consistency

- In UNIX local file system, concurrent file reads and writes have “sequential” consistency semantics
  - Each file read/write from user-level app is an atomic operation
    - The kernel locks the file vnode
  - Each file write is immediately visible to all file readers
- Neither NFS nor AFS provides such concurrency control
  - NFS: “sometime within 30 seconds”
  - AFS: session semantics for consistency (next slide)

Session Semantics in AFS v2

- What it means:
  - A file write is visible to processes on the same box immediately, but not visible to processes on other machines until the file is closed
  - When a file is closed, changes are visible to new opens, but are not visible to “old” opens
  - All other file operations are visible everywhere immediately
- Implementation
  - Dirty data are buffered at the client machine until file close, then flushed back to server, which leads the server to send “break callback” to other clients

AFS Write Policy

- Writeback cache
  - Opposite of NFS "every write is sacred"
  - Store chunk back to server
    - When cache overflows
    - On last user close()
  - ...or don't (if client machine crashes)
- Is writeback crazy?
  - Write conflicts "assumed rare"
  - Who wants to see a half-written file?

Results for AFS

- Lower server load than NFS
  - More files cached on clients
  - Callbacks: server not busy if files are read-only (common case)
- But maybe slower: Access from local disk is much slower than from another machine's memory over LAN
- For both:
  - Central server is bottleneck: all reads and writes hit it at least once;
  - is a single point of failure.
  - is costly to make them fast, beefy, and reliable servers.

Topic 3: Name-Space Construction and Organization

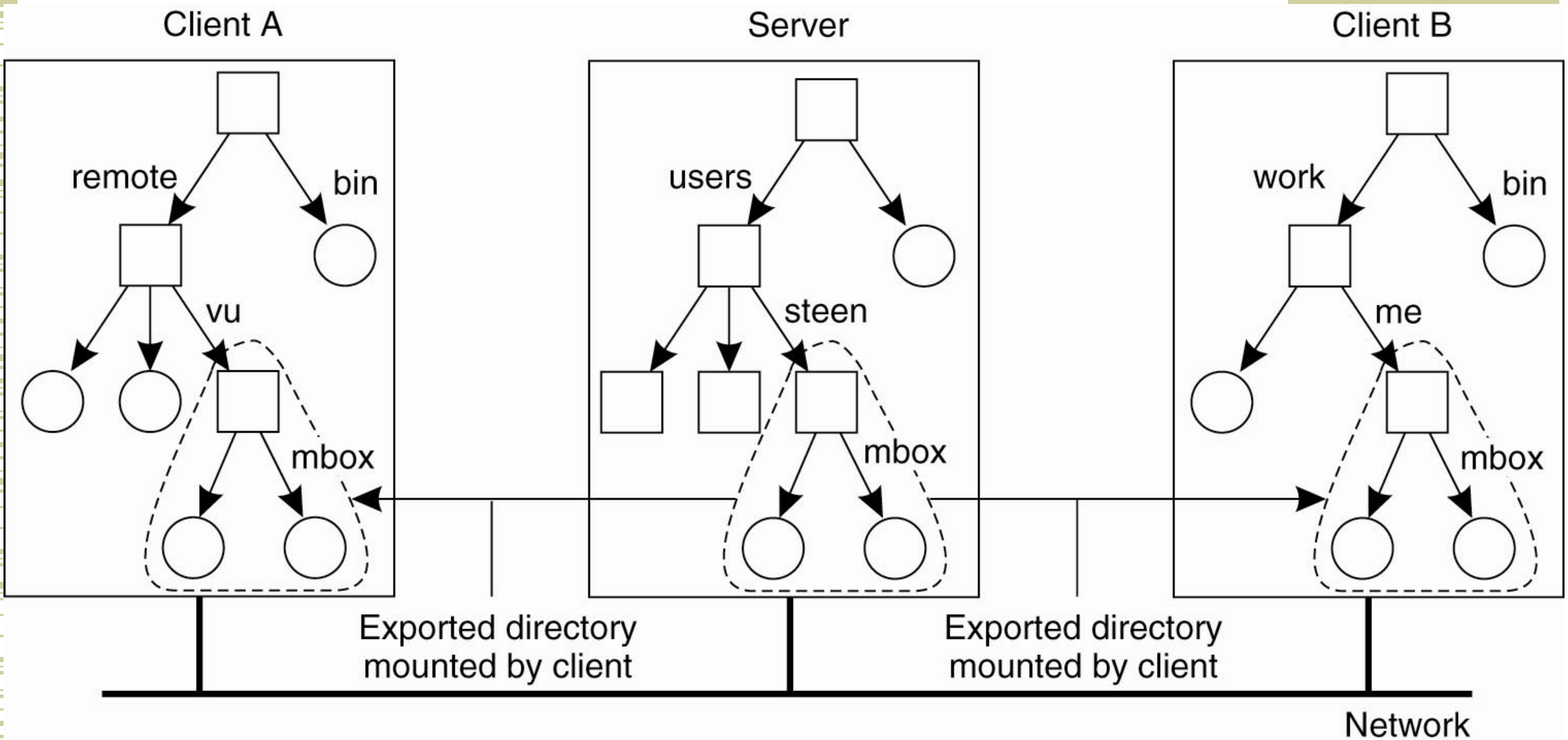
- NFS: per-client linkage
  - Server: export /root/fs1/
  - Client: mount server:/root/fs1 /fs1
- AFS: global name space
  - Name space is organized into Volumes
    - Global directory /afs;
    - /afs/cs.wisc.edu/vol1/…; /afs/cs.stanford.edu/vol1/…
  - Each file is identified as fid = <vol_id, vnode #, unique identifier>
  - All AFS servers keep a copy of “volume location database”, which is a table of vol_id→ server_ip mappings

Implications on Location Transparency

- NFS: no transparency
  - If a directory is moved from one server to another, client must remount

- AFS: transparency
  - If a volume is moved from one server to another, only the volume location database on the servers needs to be updated

Naming in NFS (1)



Naming in NFS (2)