Distributed File Systems

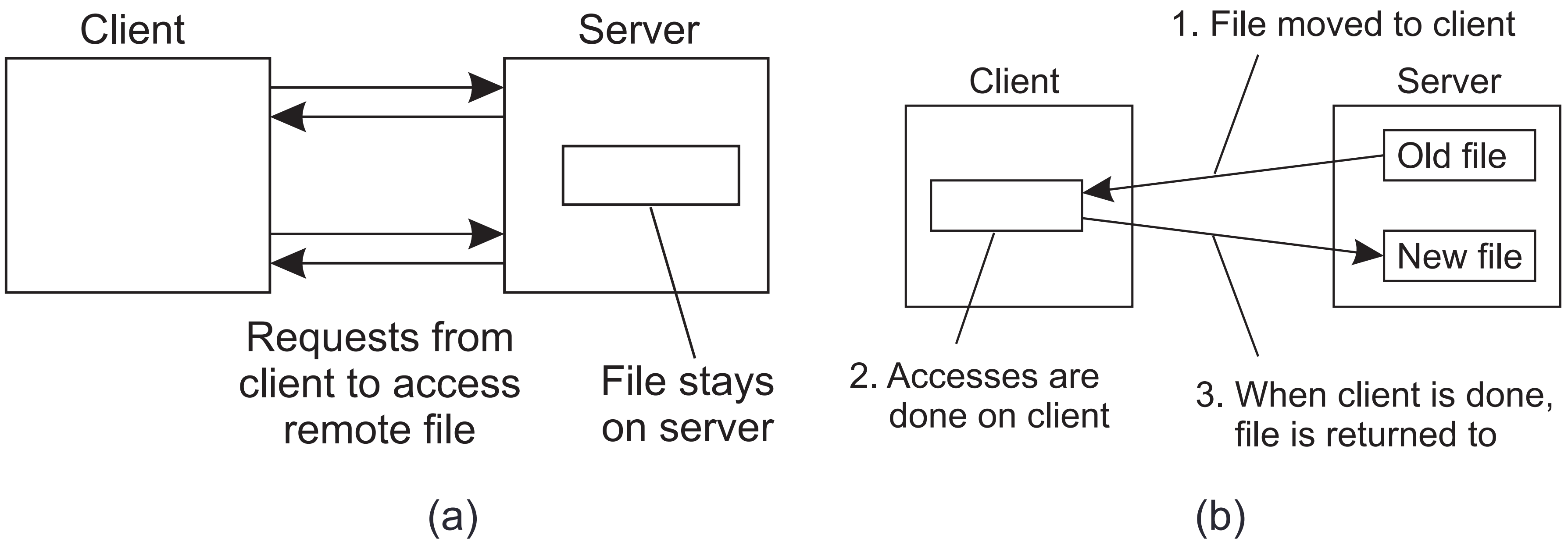
Yao Liu

Distributed file systems

- Desirable features:
    - Transparency: client accesses DFS files as if it were accessing local (say, Unix) files
        - Same API as local files, i.e., client code doesn’t change
        - Need to make location, replication, etc. invisible to client
    - Support concurrent clients
        - Multiple client processes reading/writing the file concurrently
    - Replication: for performance and fault-tolerance

- Distributed, and Sharing

File access model


Client
Server
Requests from client to access remote file
File stays on server
(a)
1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to
(b)

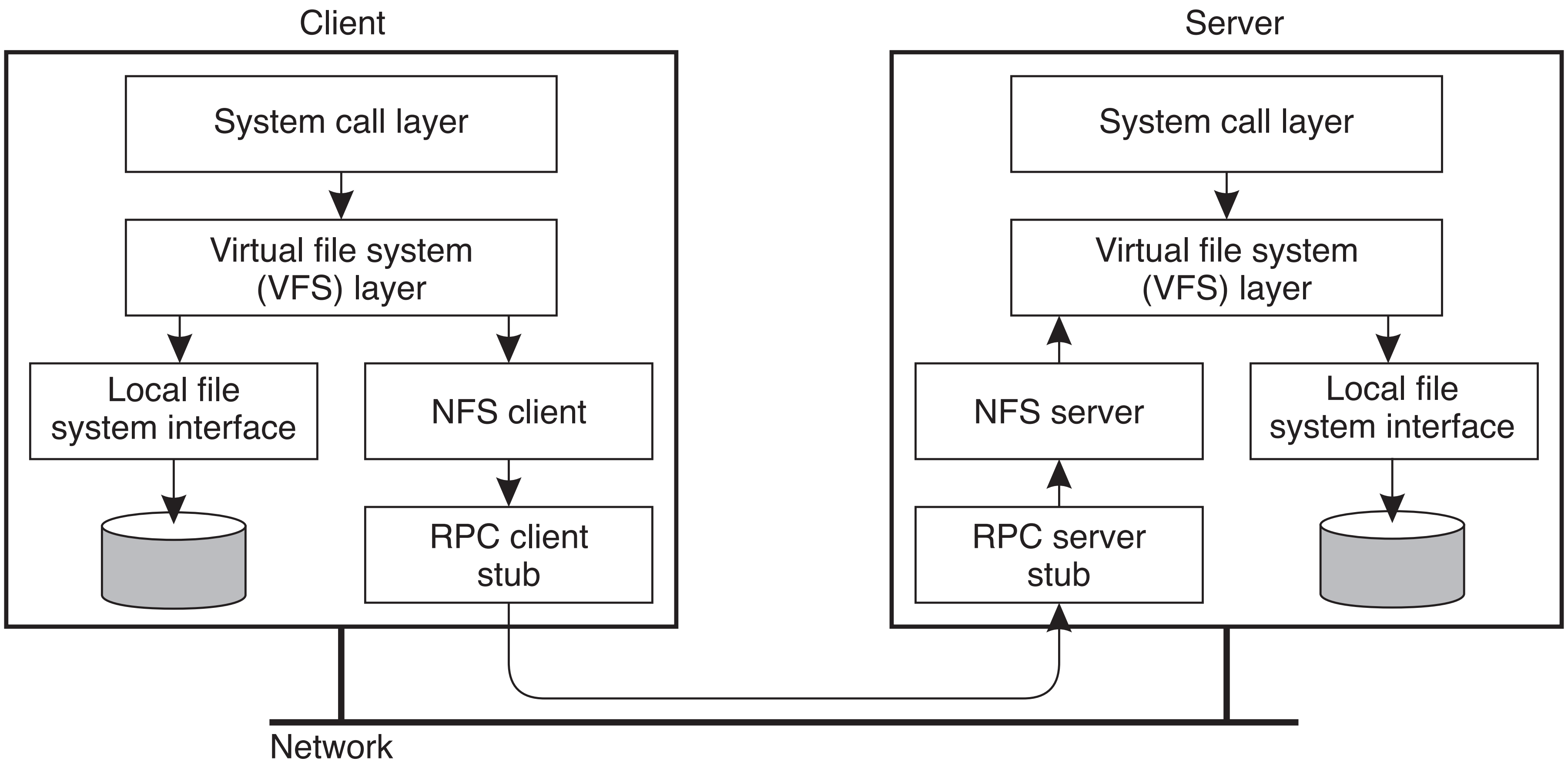
(a) The remote access model. (b) The upload/download model.

NFS: Network File System

NFS

- Network File System
- Created by SUN in 1985. Protocol defined in
  - RFC 1813 (version 3)
  - RFC 3530 (version 4)
- Still widely used today,
  e.g., our `remote.cs.binghamton.edu`
  environment.

NFS architecture


Client
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
Server
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

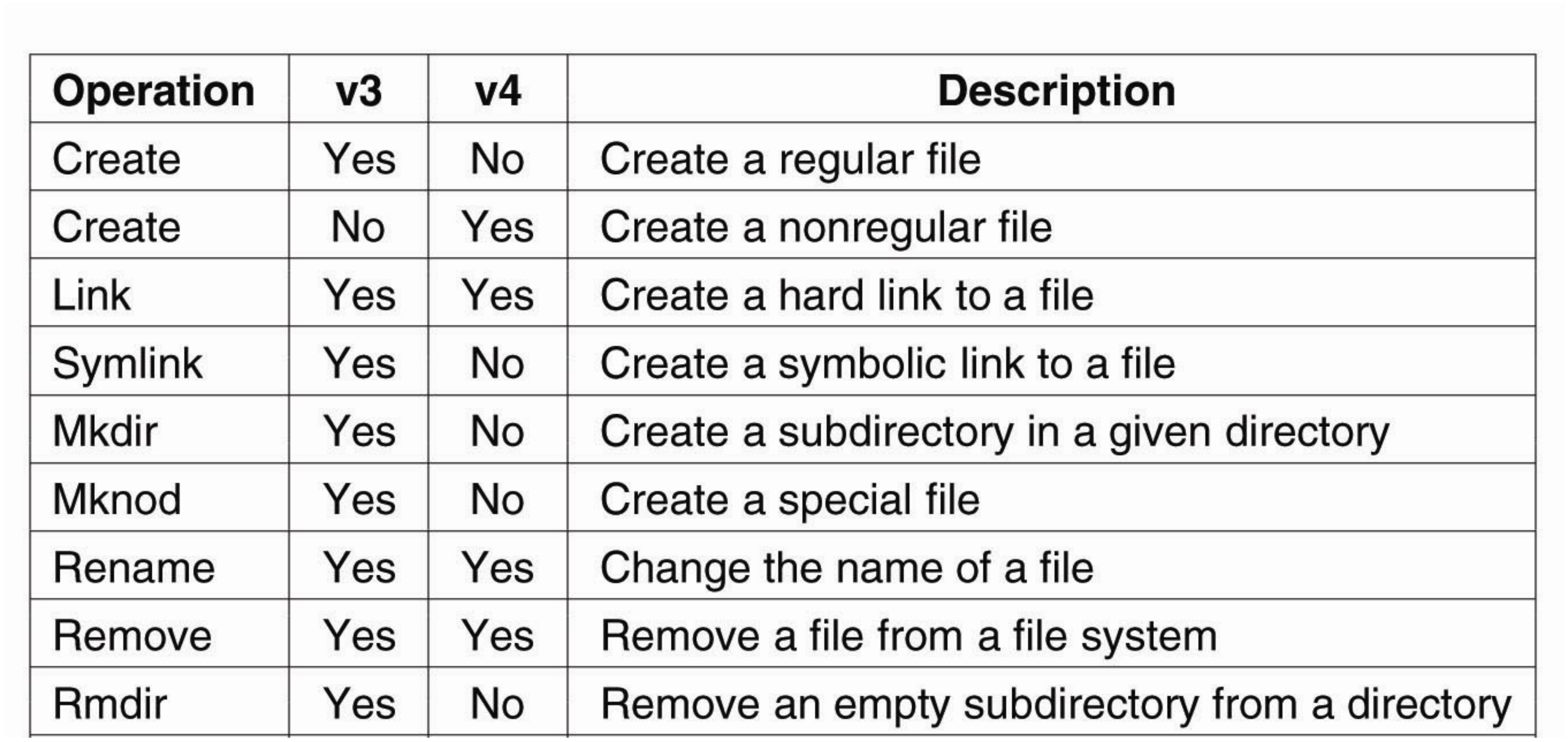
Stateful vs. stateless

- Stateless model: each call contains complete information to execute operation

- Stateful model: server maintain context (info) shared by consecutive operations

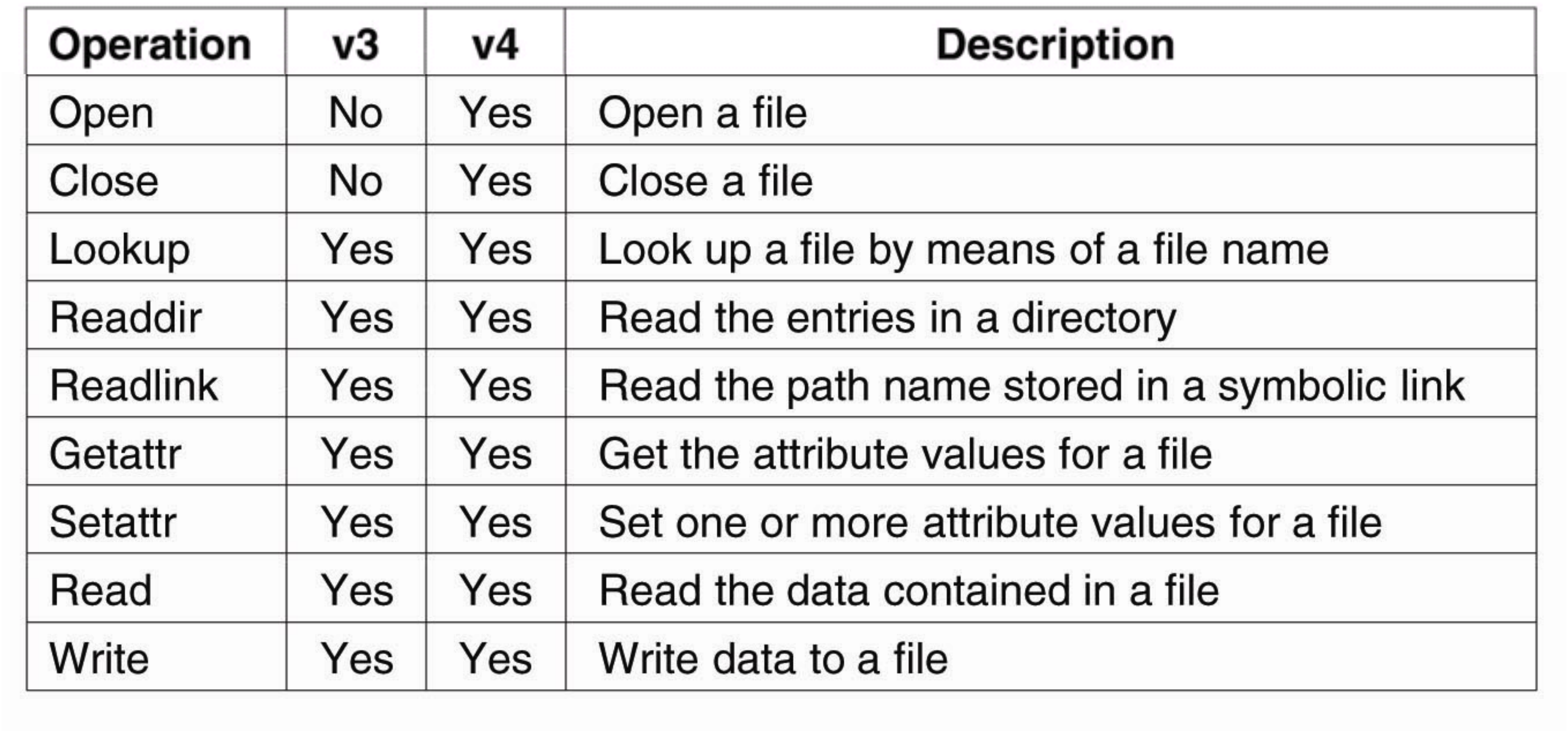
- First three versions of NFS were **stateless**; version 4 is **stateful**

NFS file operations



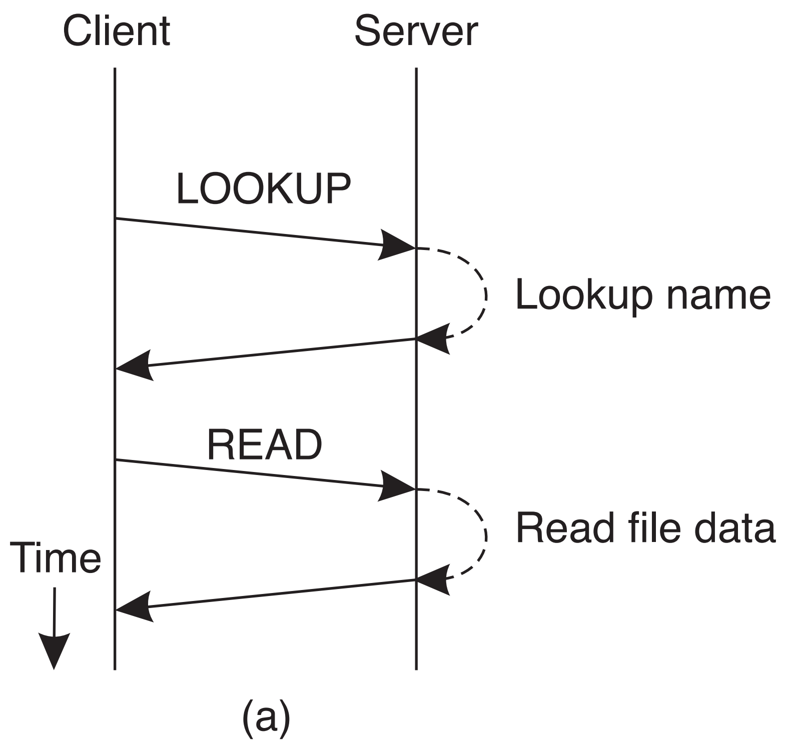
Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory from a directory

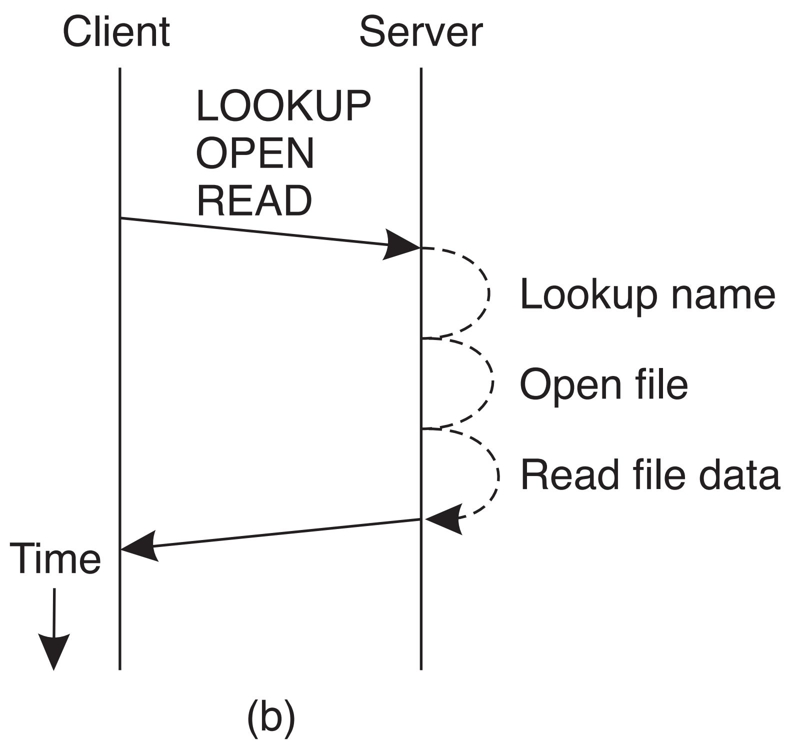
NFS file operations



Operation	v3	v4	Description
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Get the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

Remote procedure calls in NFS



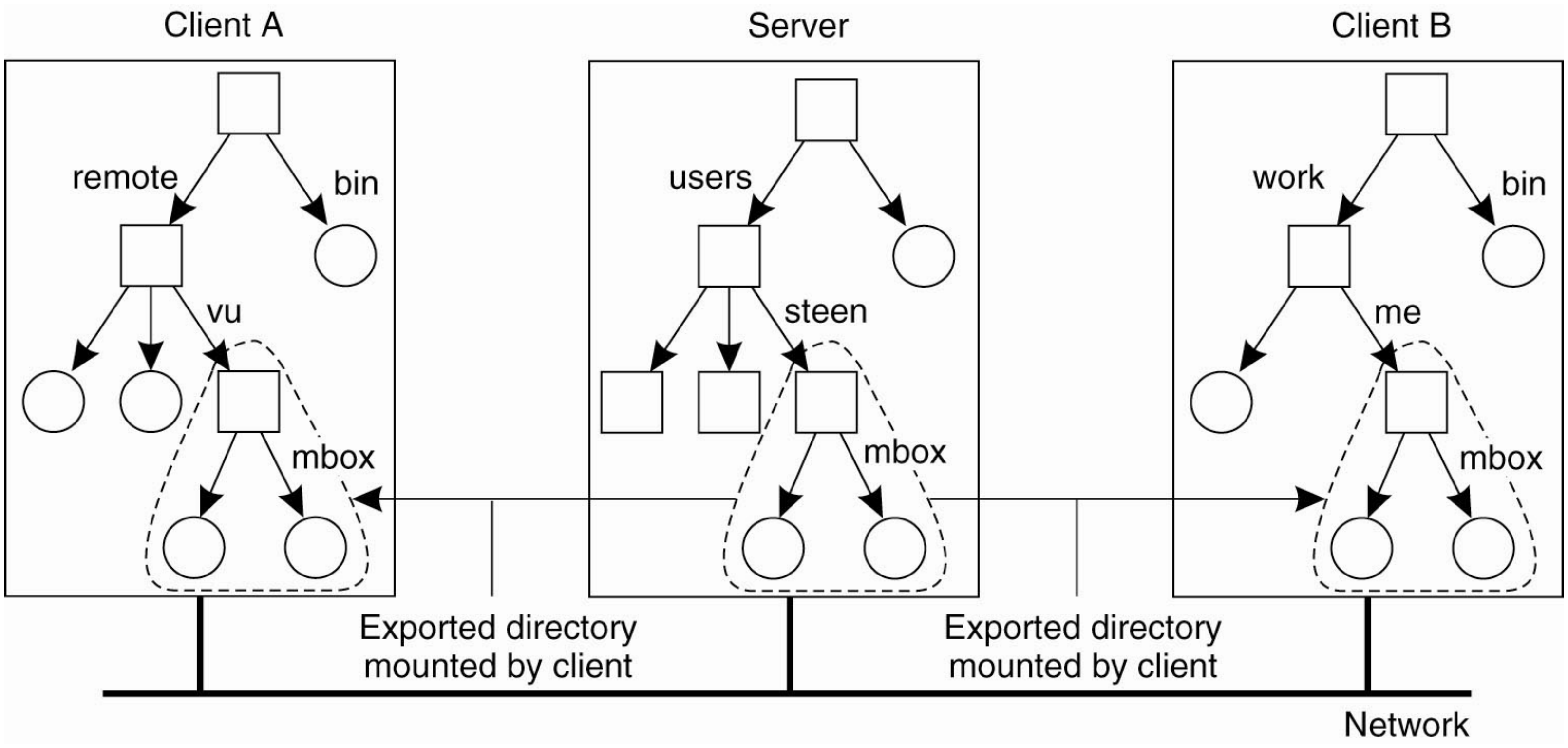


(a) Reading data from a file in NFS version 3.

(b) Reading data using a compound procedure in version 4.

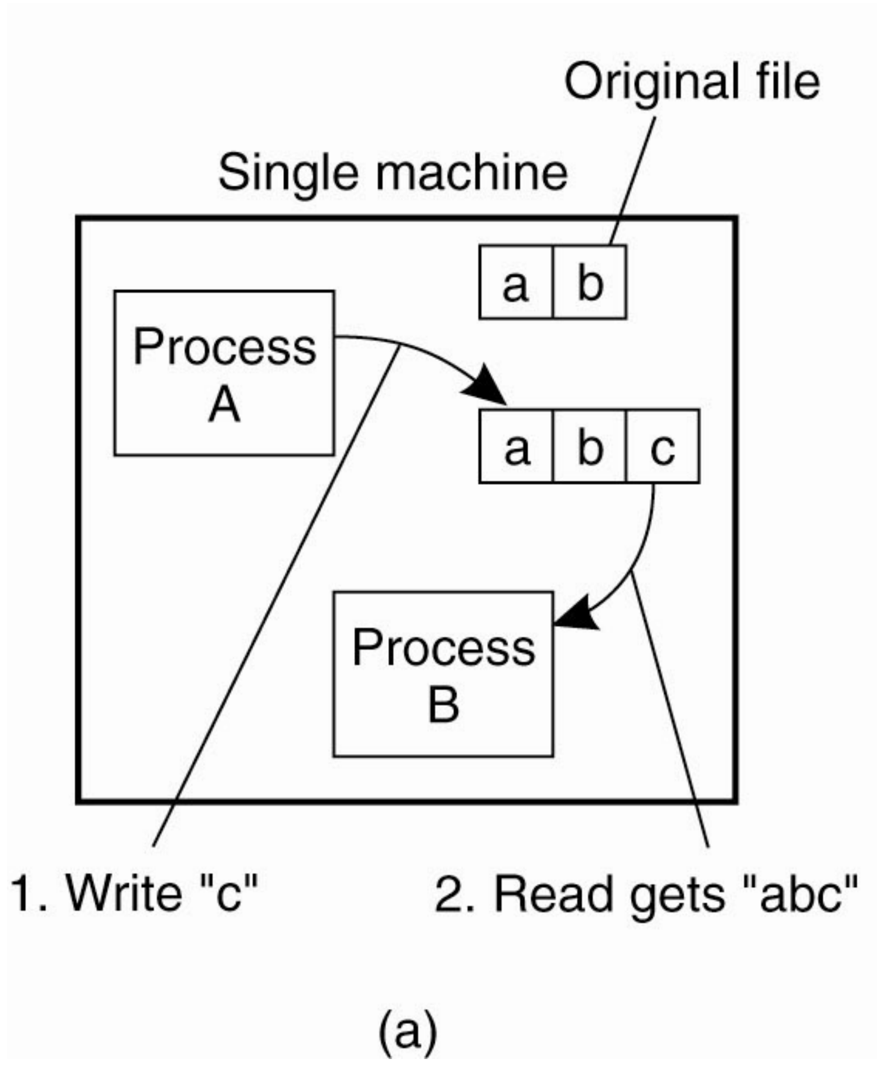
Naming in NFS

- Allow a client to mount a remote file system into its own local file system
- Pathnames are **not** globally unique


Client A
Server
Client B
remote
bin
vu
mbox
users
steen
mbox
work
bin
me
mbox
Exported directory mounted by client
Exported directory mounted by client
Network

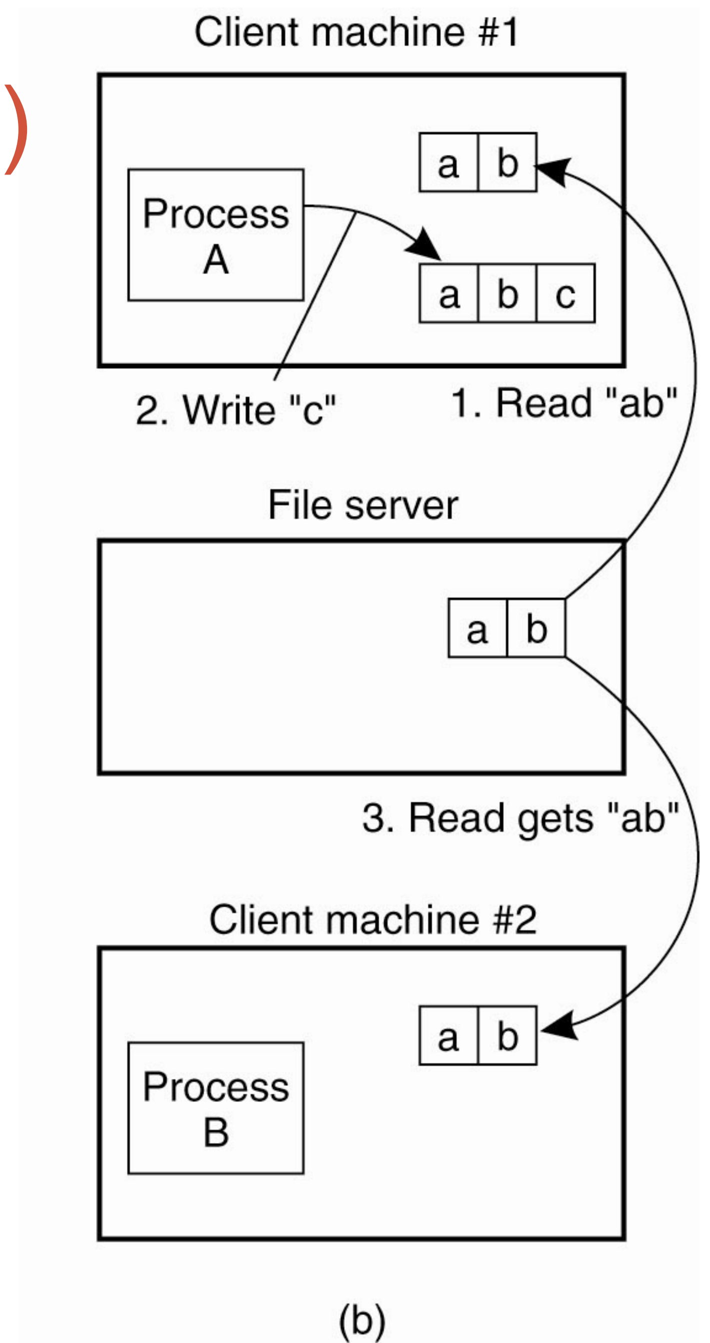
Semantics of file sharing (1)

On a single processor, when a read follows a write, the value returned by the read is the value just written.



Semantics of file sharing (2)

In a distributed system with caching, obsolete values may be returned.

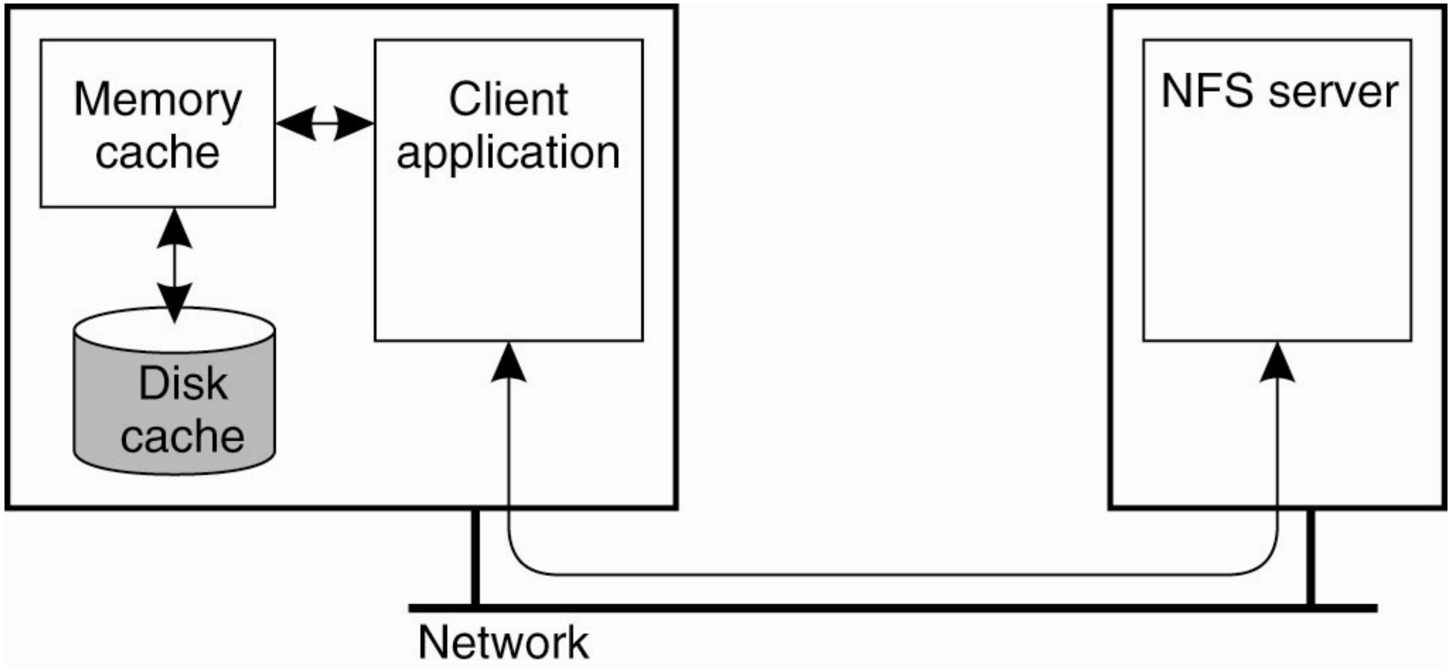


Semantics of file sharing (3)

- NFS implements session semantics
  - Use local caches for performance and provide session semantics
  - No changes are visible to other processes until the file is closed

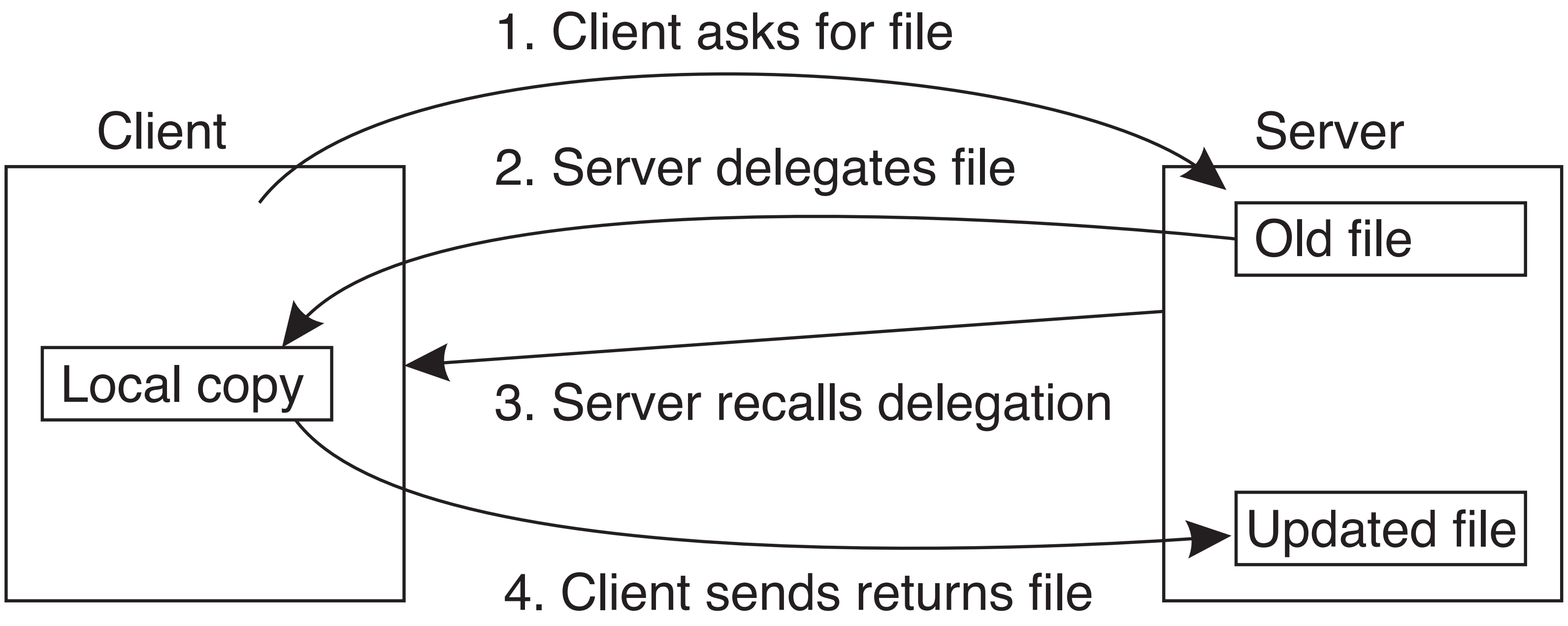
Client-side caching

- Client-side caching is left to the implementation (NFS does not prohibit it)
  - Different implementation use different caching policies
  - e.g., allow cache data to be stale for up to 30 seconds



Client-side caching: delegation

- NFS v4: use file delegation:
  - Server delegates local open and close requests to the NFS client
  - Uses a callback mechanism to recall file delegation.



Reading

- Chapter 11 in TBook
- Section 12.3 in CBook

AFS: Andrew File System

Andrew File System

- Designed at CMU
  - Named after Andrew Carnegie and Andrew Mellon, the “C” and “M” in CMU
- In use today in some clusters (especially university clusters)

Key design characteristics

- Whole-file serving:
  - Entire contents of the directories and files are transmitted to the client
- Whole-file caching:
  - The cache is permanent, survives reboots

- Based on (validated) assumptions that
  - Most file accesses are by a single user
  - Most files are small
  - The local cache can be allocated a substantial proportion of the disk space on each server
  - File reads are much more often than file writes, and typically sequential

AFS details

- Clients system: Venus service
- Server system: Vice service
- Reads and writes
  - Done on local copy of file at client (Venus)
  - When file closed, writes propagated to Vice
- When a client (Venus) opens a file, Vice:
  - Sends it entire file
  - Gives client a callback promise
- Callback promise
  - Promise that if another client modifies then closes the file, a callback will be sent from Vice to Venus
  - Callback state at Venus only binary: valid or canceled

AFS system calls

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus. Open the local file and return the file descriptor to the application.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	→ ←	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

Reading

- Section 12.4 in CBook

GFS: Google File System

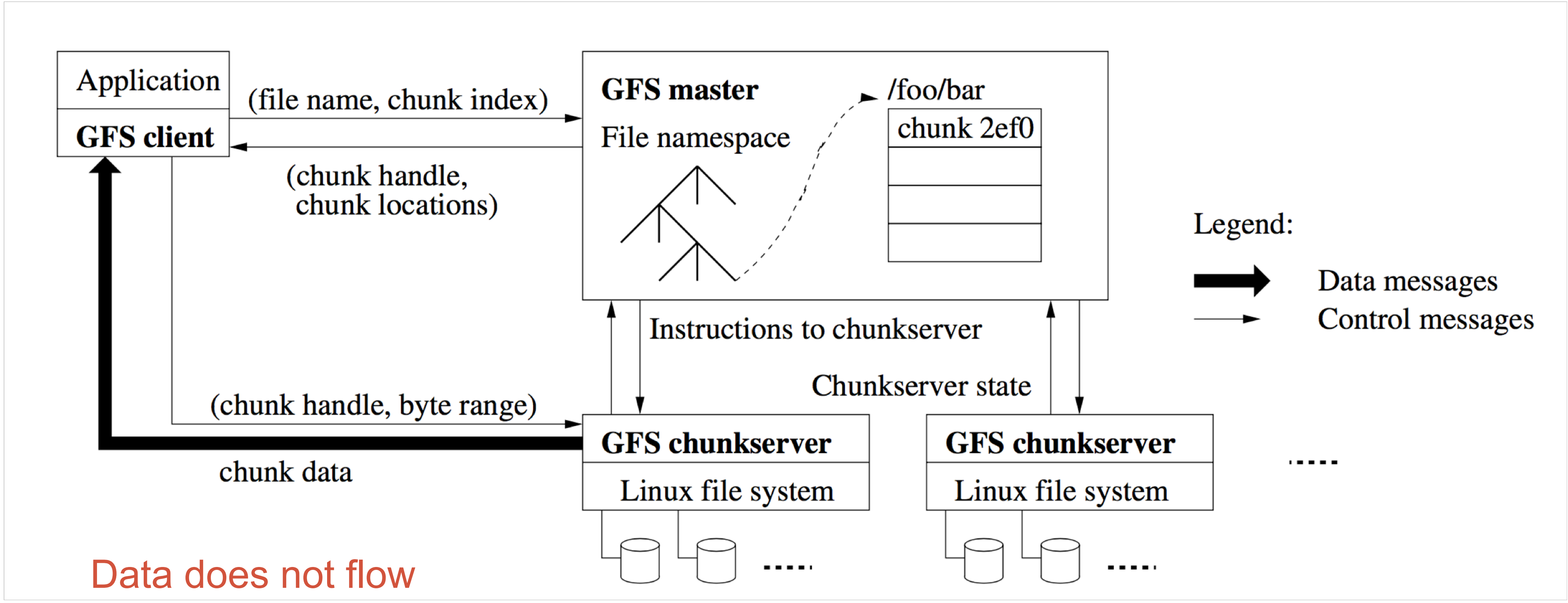
GFS assumptions

- Commodity hardware
  - inexpensive but failures are common
- Files are large (GB/TB), but there aren't that many files
- Workloads
  - Large streaming reads
  - Small random reads
  - Large, sequential writes that append data to files
- Once written, files are seldom modified (!=append) again
  - Random modification in files possible, but not efficient in GFS
- High sustained bandwidth more important than low latency

GFS architecture

Multiple clients

One single master



Data does not flow through the master

multiple chunkservers

Files on GFS

- Files are divided into **fixed size chunks** (64MB) with unique 64 bit identifiers
  - IDs assigned by GFS master at chunk creation time
- Advantages of large, fixed-size chunks:
  - Disk seek time small compared to transfer time
  - A single file can be larger than a node’s disk space
  - Fixed size makes allocation computations easy

Files on GFS

- **chunkservers** store chunks on local disk as "normal" Linux files
  - Reading & writing of data specified by the tuple (chunk_handle, byte_range)
- Files are replicated (by default 3 times) across all chunk servers

Files on GFS

- **master** maintains all file system metadata
  - Namespace, access control information, mapping from file to chunks, chunk locations, chunk migration, ...
- **Heartbeat** messages between master and chunkservers
  - Is the chunkserver still alive?
  - What chunks are stored at the chunkserver?
- To read/write data: client communicates with master (for metadata operations) and chunkservers (for data)

Files on GFS

- Clients cache metadata
- Clients do not cache file data (why?)
- **chunkservers** do not cache file data (rely on the underlying file system: Linux’s buffer cache)

Single master

- Single master simplifies the design:
    - global knowledge of chunk placement and replication
- However, from distributed systems we know this is a:
    - single point of failure
    - scalability bottleneck
- GFS solutions:
    - Shadow masters
    - minimize the number of reads and writes at the master
        - never move data through it, only use it for metadata
            - also cache metadata at clients
        - large chunk size (64 MB)
        - master delegates authority to primary replicas in data mutations (chunk leases)

Master's responsibility

- Metadata storage
- Namespace management
  - metadata read/write
- Periodic communication with chunkservers
  - give instructions, collect state, track cluster health
- Garbage collection

Metadata on the master

- **3** types of metadata:
  - Files and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- All metadata is kept in master's memory (fast access)
  - Sets limits on the entire system's capacity
- Operation log is kept on the master's local disk
  - master state can be recovered in case of crash
  - logs the namespaces and mappings
  - does not log chunk locations (why?)

Chunk locations

- Master does NOT keep a persistent record of chunk replica locations
- Instead, it polls all chunkservers about their chunks at startup
- Master keeps its info up to date through periodic HeartBeat messages
    - Chunkserver has the final say over what chunks it has

Chunk placement

- Creation of (initially empty) chunks
  - Use under-utilized chunk servers
  - Limit number of recent creations on each chunk server
- Re-replication
  - Started once the number of available chunk replicas falls below threshold
  - Master instructs chunkserver to copy chunk data directly from existing valid replica
- Re-balancing
  - Chunkservers serving popular files may become a hotspot
  - Changes in replica distribution for better load balancing
  - Gradually fills new chunkservers

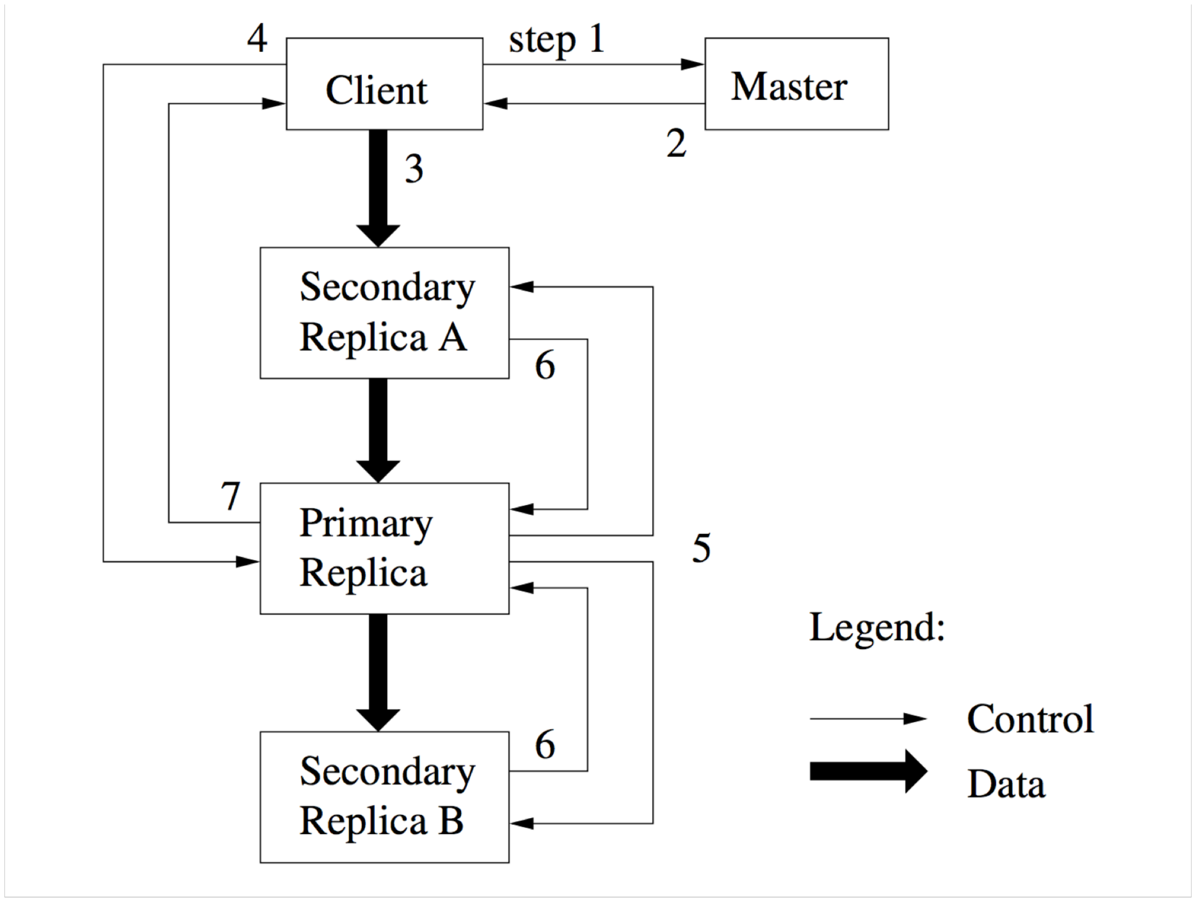
Garbage collection

- Deletion logged by master
- File renamed to a hidden file, keep deletion timestamp
- Periodic scan of the master's file system namespace
  - Hidden files older than 3 days are deleted from the master
- Periodic scan of the chunk namespace
  - Orphaned chunks (not reachable from any file) are identified, their metadata deleted
- HeartBeat messages between chunkserver and master
  - used for master to reply the identity of all chunks that can be deleted

Mutations

- Mutation can be **write** or **append**
  - Must be done for all replicas
- Need to minimize master involvement
- **Lease** mechanism:
  - Master picks one replica as the “primary”, gives it a “lease” for mutations
  - If multiple mutations, the primary determines a serial order
  - all replicas follow this order
- Data flow decoupled from the control flow

Control and data flow of write



Stale replica detection

- If a chunkserver misses a “mutation” applied to a chunk, it becomes a stale replica
- Master maintains a **chunk version number** to distinguish up-to-date and stale replicas
- Master ensures the version number is advanced before an operation on a chunk
- Stale replicas are removed during garbage collection

Data corruption

- chunkservers use **checksums** to detect data corruption
- Chunk is broken into 64 KB blocks, each has a 32 bit checksum.
  - Keep in memory and store on persistent storage
- Read requests: checksum is verified for every block that's in the read range
  - never sends corrupted data to clients

Reading

- The Google File System
- http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

HDFS: Hadoop Distributed File System

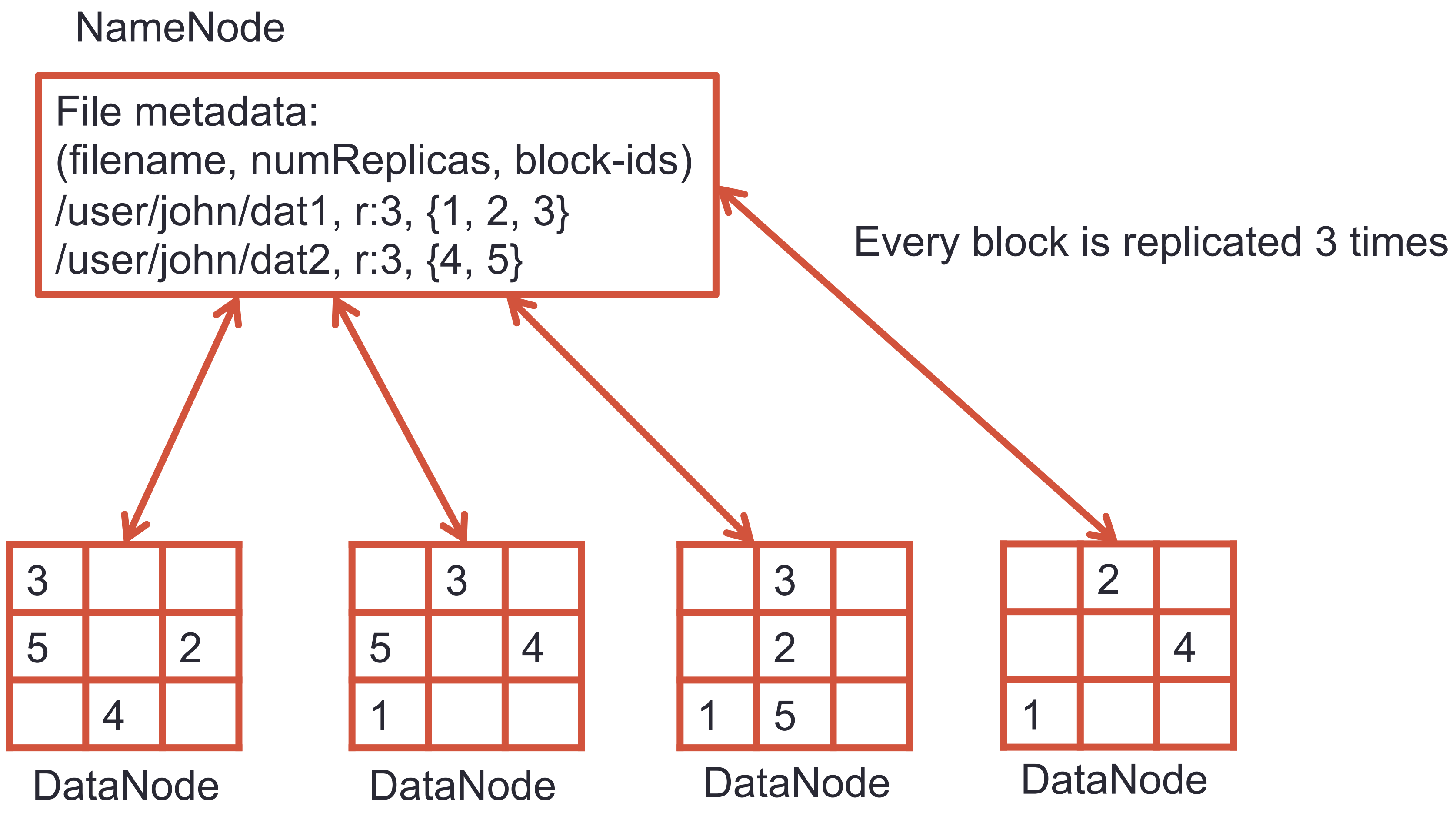
GFS vs. HDFS

GFS	HDFS
master	NameNode
chunkserver	DataNode
operation log	journal
chunk	block
random write is possible	only append is allowed
multiple-writer, multiple-reader	single-writer, multiple-reader
32 bit checksum for every 64 KB data block in every chunk	For every block, DataNode stores both the data file and the metadata file. The metadata file contains checksum and timestamp
default chunk size: 64 MB	default block size: 128 MB

HDFS architecture

- NameNode
  - Master of HDFS, directs the slave DataNode
  - Keeps track of file splitting into blocks, replication, block location, etc.
  - Single point of failure
- Secondary NameNode: takes snapshots of the NameNode
  - Not a standby secondary, cannot take over as the NameNode if the current NameNode fails
- DataNode

NameNode and DataNodes


NameNode
File metadata:
(filename, numReplicas, block-ids)
/user/john/dat1, r:3, {1, 2, 3}
/user/john/dat2, r:3, {4, 5}
Every block is replicated 3 times
3
5
2
4
DataNode
3
5
4
1
DataNode
3
2
1
5
DataNode
2
4
1
DataNode

Single-writer, multiple-reader model

- Only one client can write to a file at a time
- HDFS client that opens a file for writing is granted a lease
- HDFS client can renew the lease by sending a heartbeat message to the NameNode
- The lease is revoked when the file is closed

Reading

- The Hadoop Distributed File System:
- http://www.aosabook.org/en/hdfs.html

Acknowledgement

• These slides contain material developed by Indranil Gupta (UIUC) and Claudia Hauff (TUDelft)