INFSCI 2750: Cloud Computing

Lecture 2: Datacenter Filesystems

Dr. Balaji Palanisamy
Associate Professor
School of Computing and Information
University of Pittsburgh
bpalan@pitt.edu

Slides Courtesy: Prof. Bina Ramamurthy, University of Buffalo
Prof. Keke Chen, Wright State University
Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007

Outline

1. Introduction to Distributed File Systems
2. Design Principles of Distributed File Systems
3. Hadoop Distributed File System
4. Google Distributed File System

Client-Server Architectures (1)

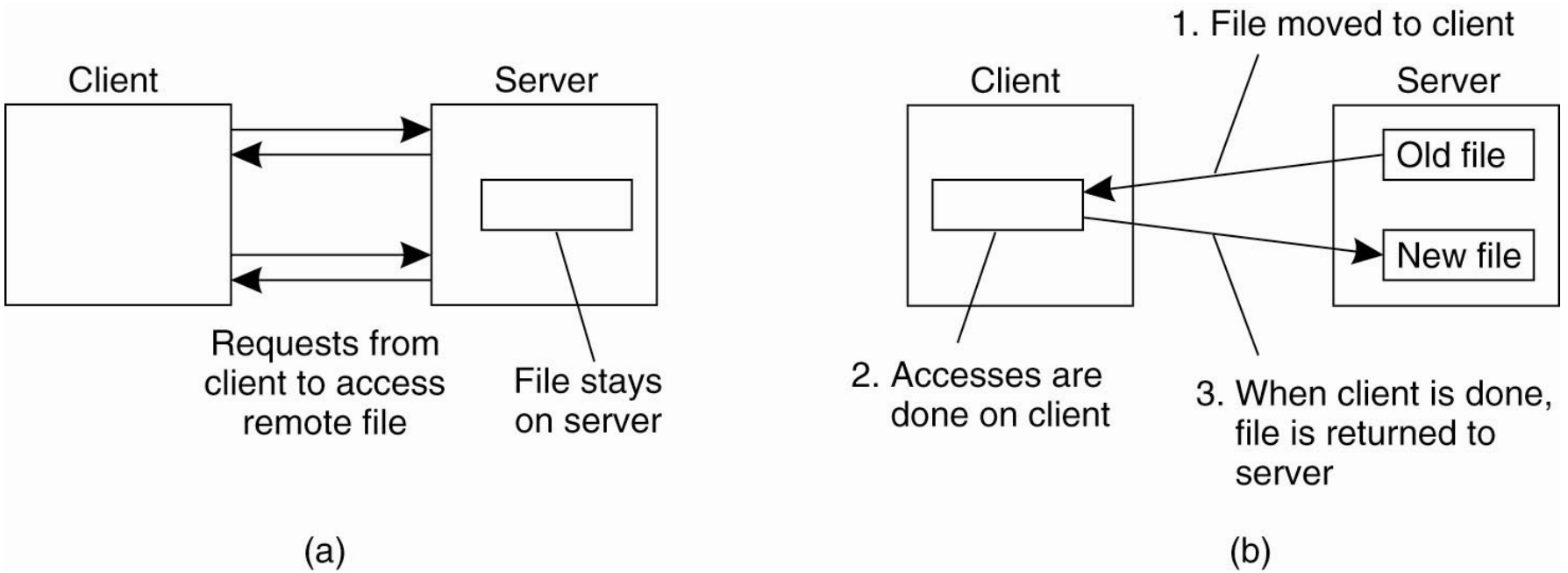


Figure 11-1. (a) The remote access model. (b) The upload/download model.

Client-Server Architectures (2)

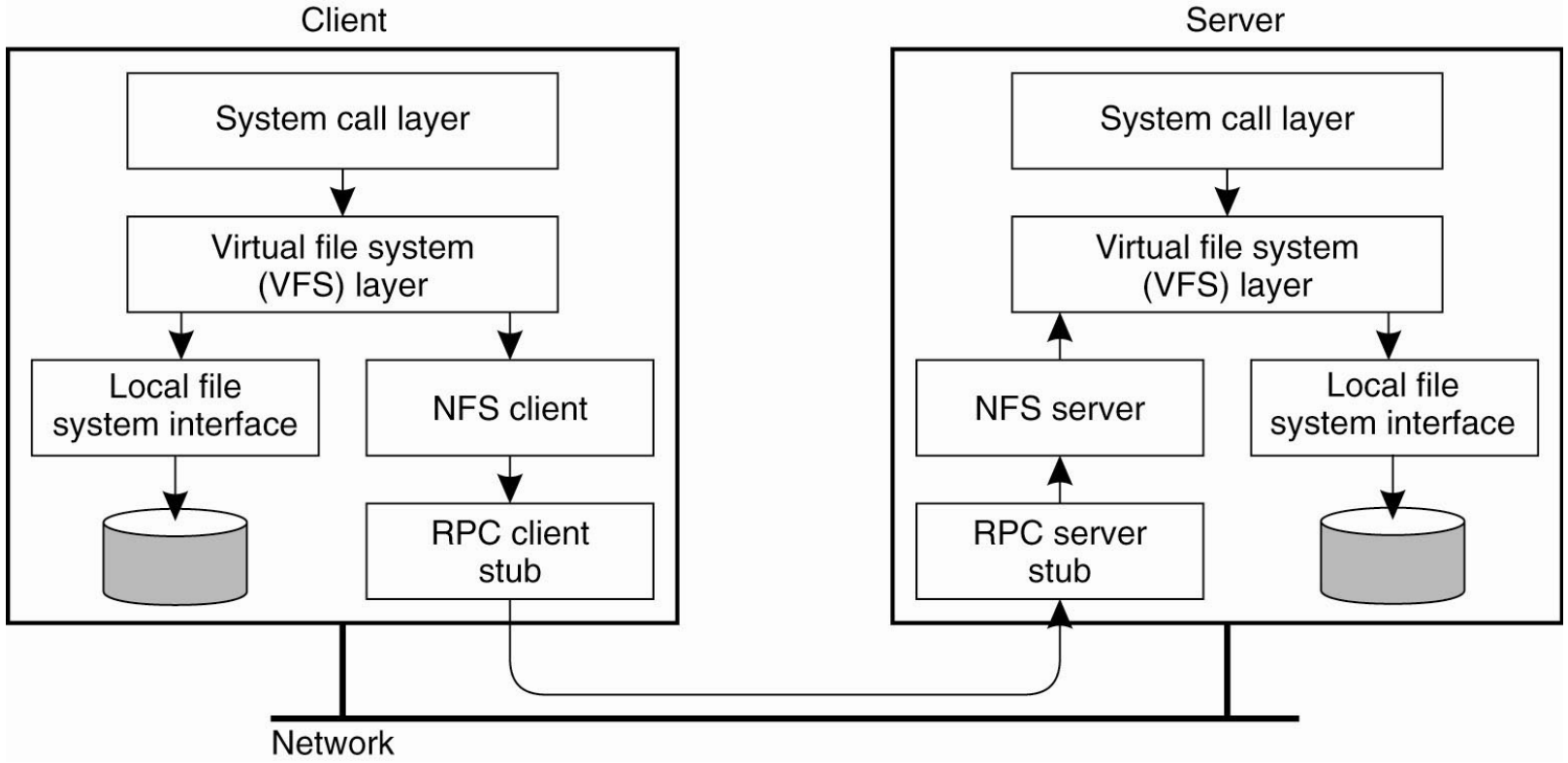
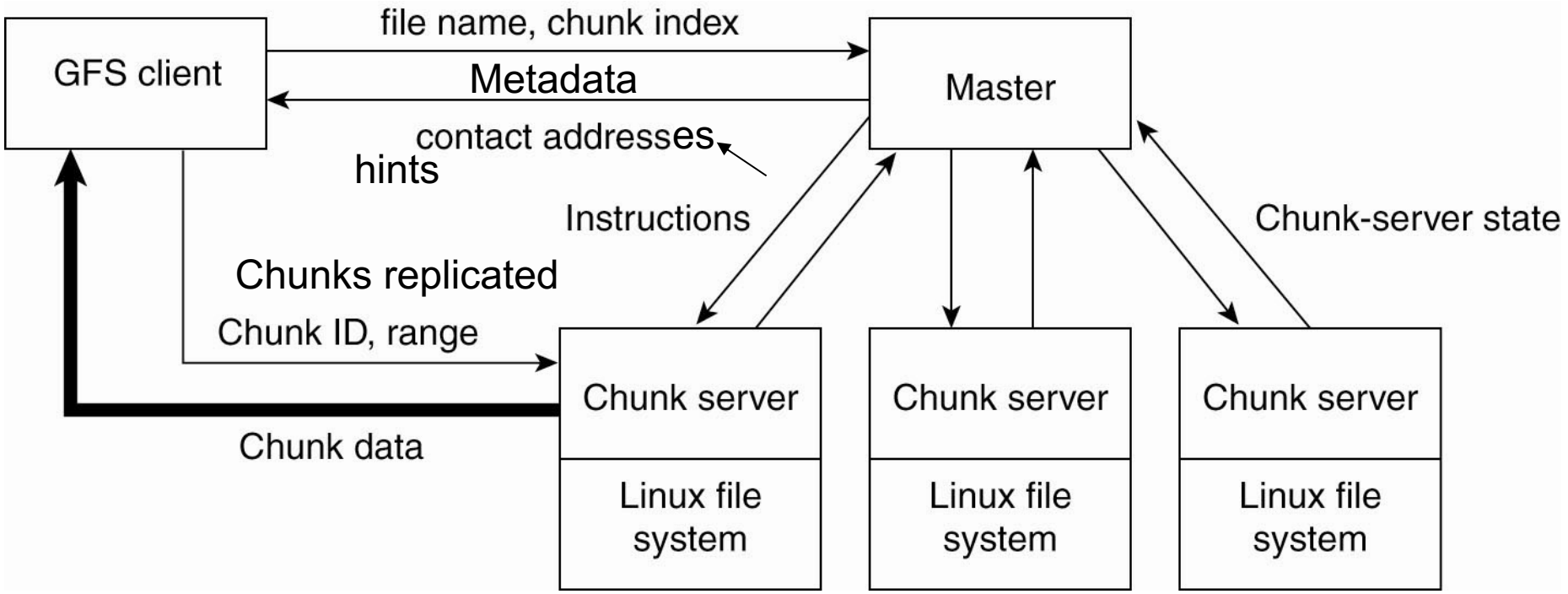


Figure 11-2. The basic NFS architecture for UNIX systems.

Cluster-Based Distributed File Systems (2)

Massive numbers of servers = highly likely one or more is down!



Master does not try to stay consistent all the time – update info by polling

Master keeps metainfo in RAM – like logging file system; reduces I/O

Figure 11-5. The organization of a Google cluster of servers.

Symmetric Architectures

Key choice: are files or blocks what is stored on servers?

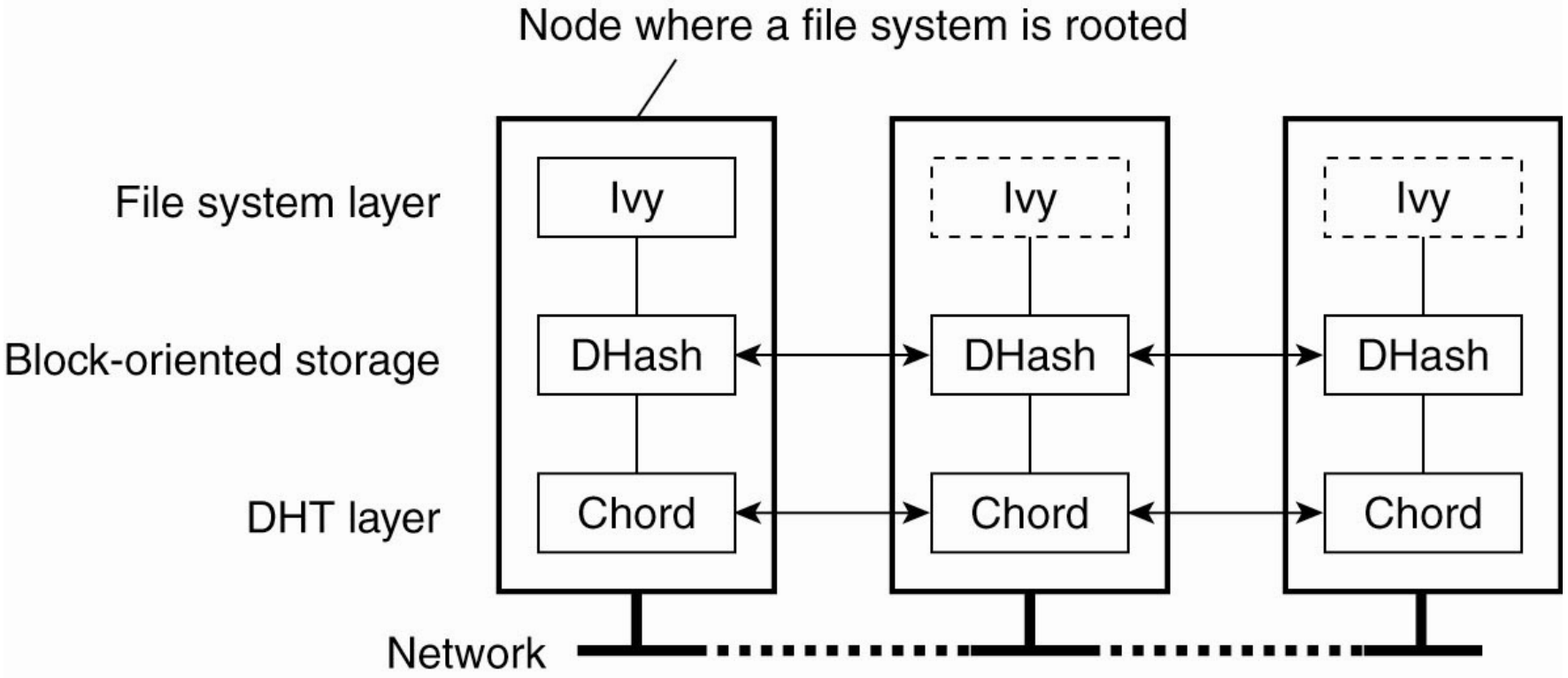


Figure 11-6. The organization of the Ivy distributed file system.

Cluster-Based Distributed File Systems (1)

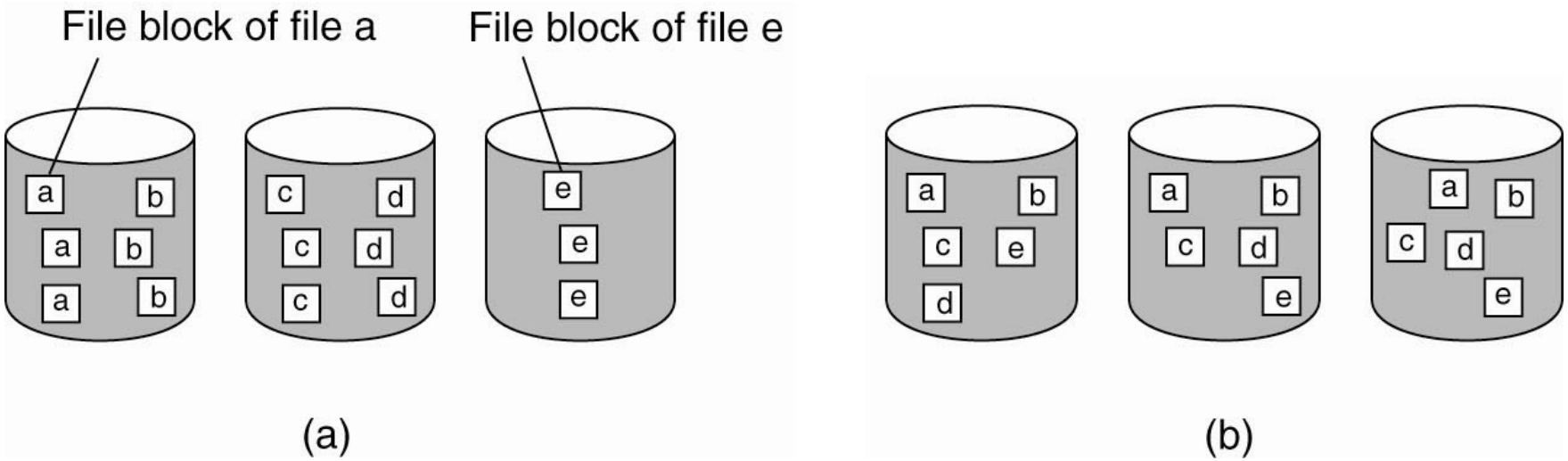
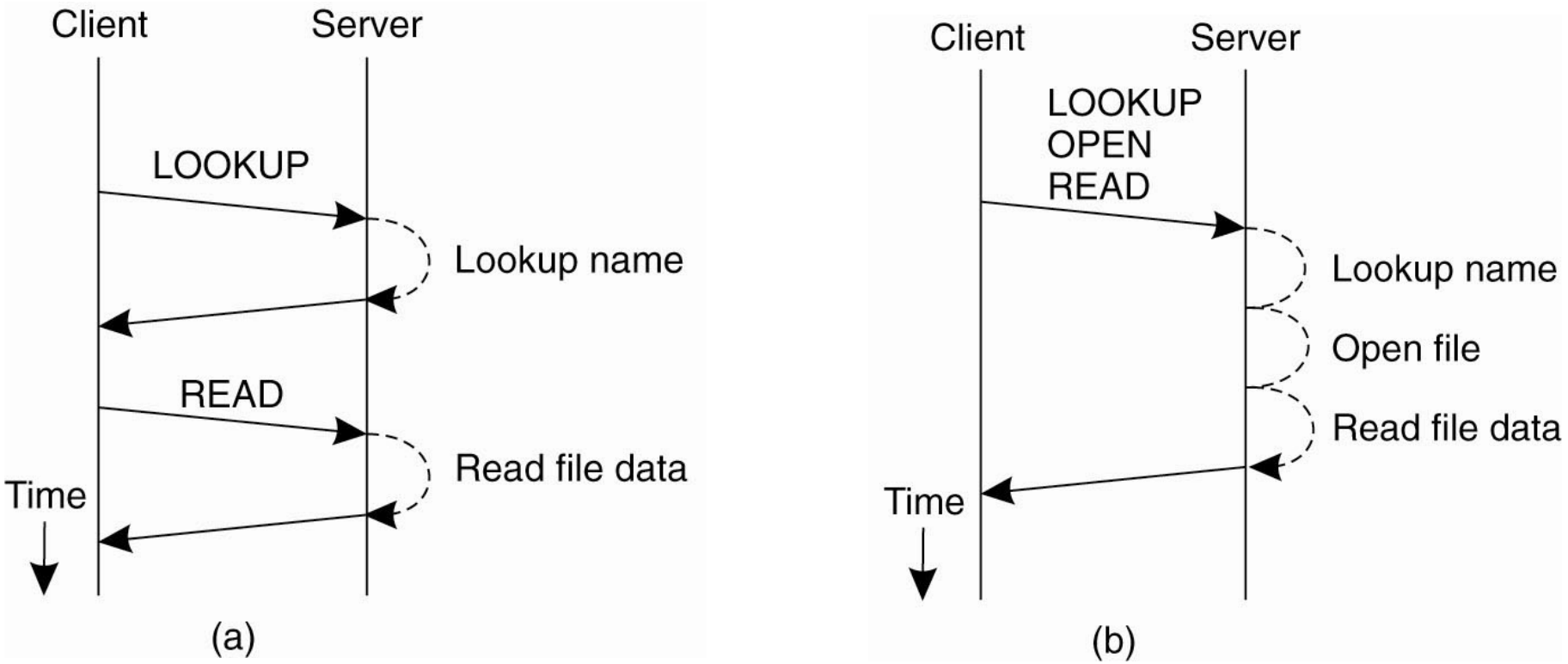


Figure 11-4. The difference between (a) distributing whole files across several servers and (b) striping files for parallel access.

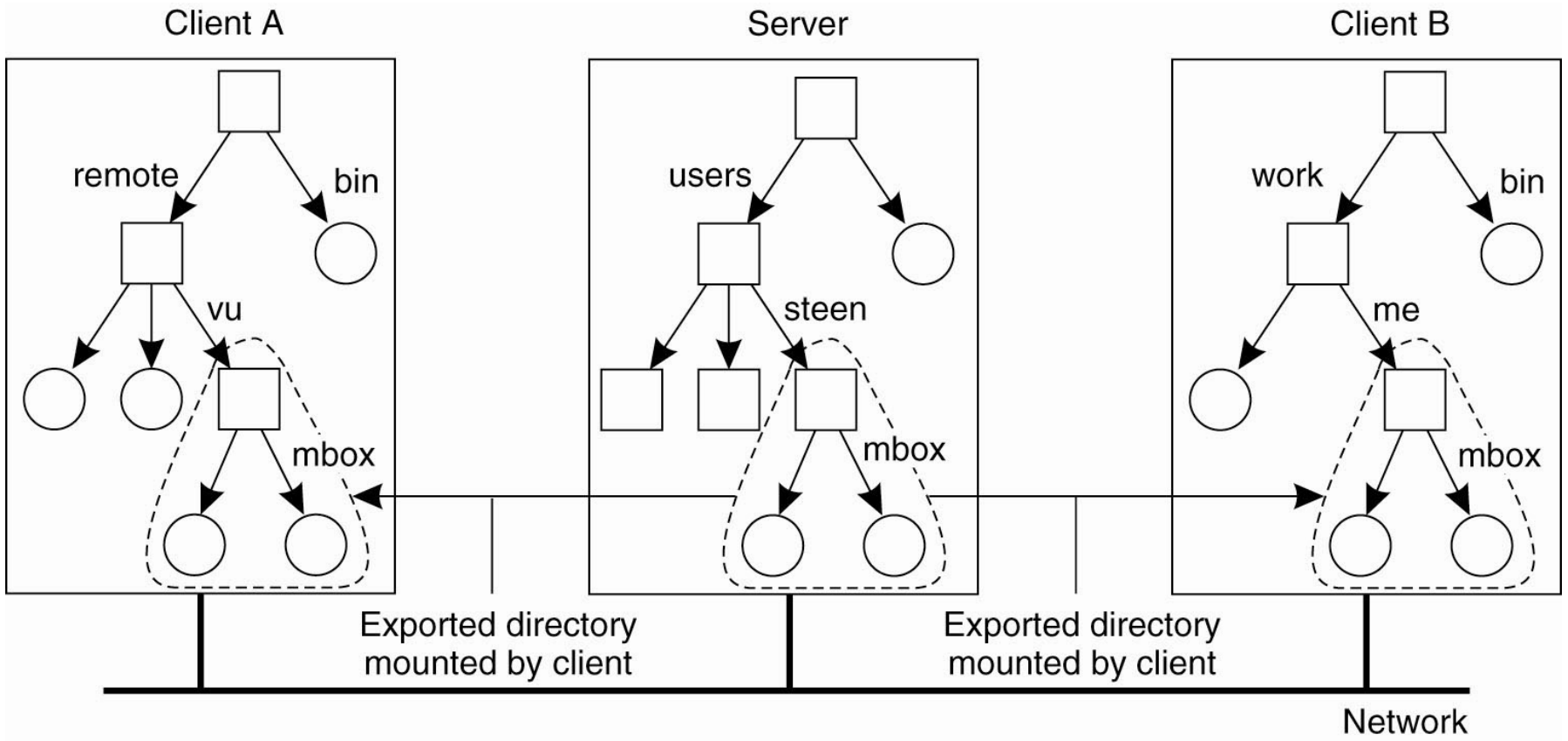
Remote Procedure Calls in NFS



No transaction semantics – just list of requests

Figure 11-7. (a) Reading data from a file in NFS version 3. (b) Reading data using a compound procedure in version 4.

Naming in NFS (1)



Only explicitly exported subdirectories can be mounted remotely

Path names depend on client mount points – FS view not uniform

Figure 11-11. Mounting (part of) a remote file system in NFS.

Naming in NFS (2)

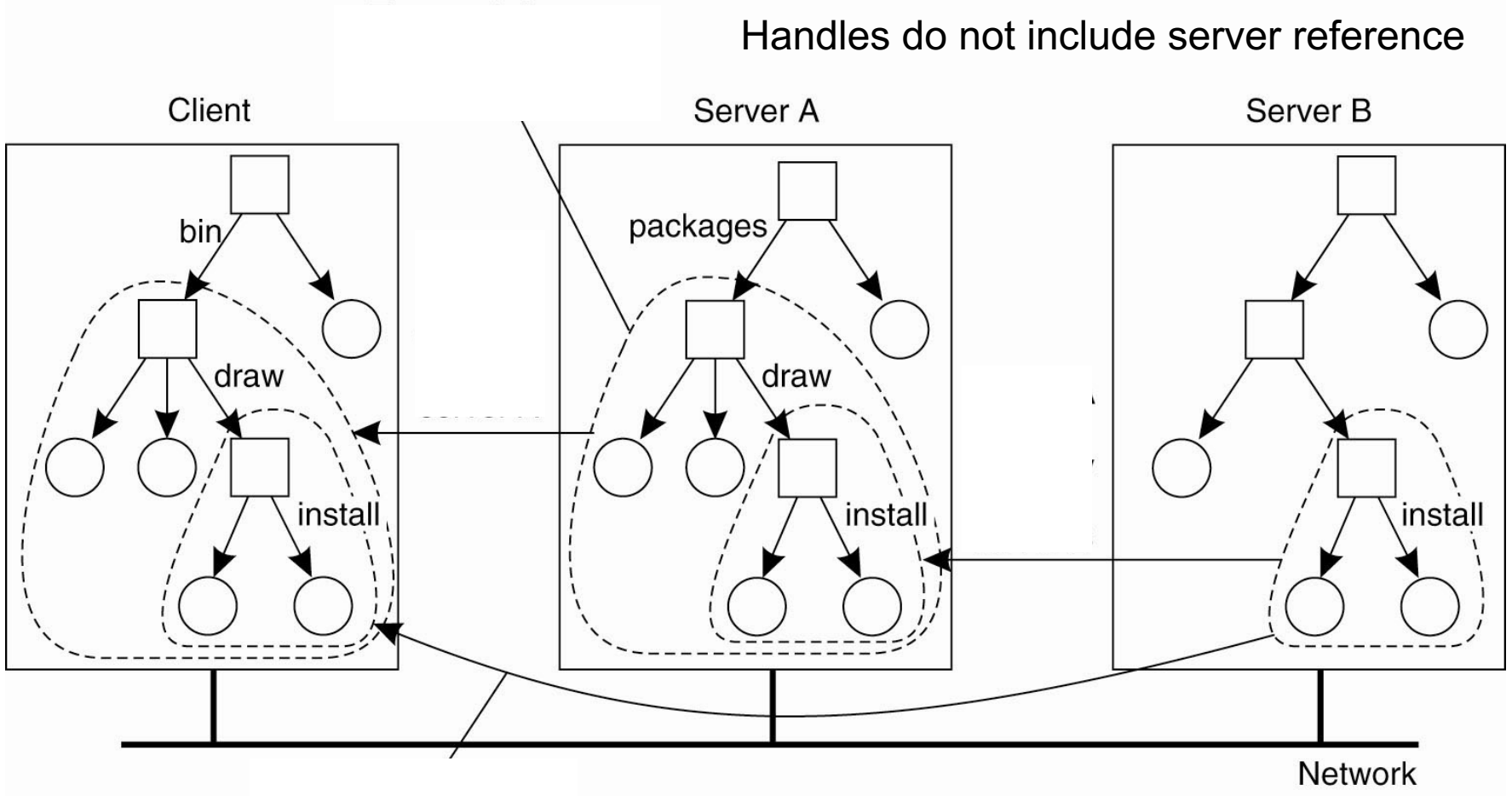


Figure 11-1 ; nested directories from multiple servers in NFS.

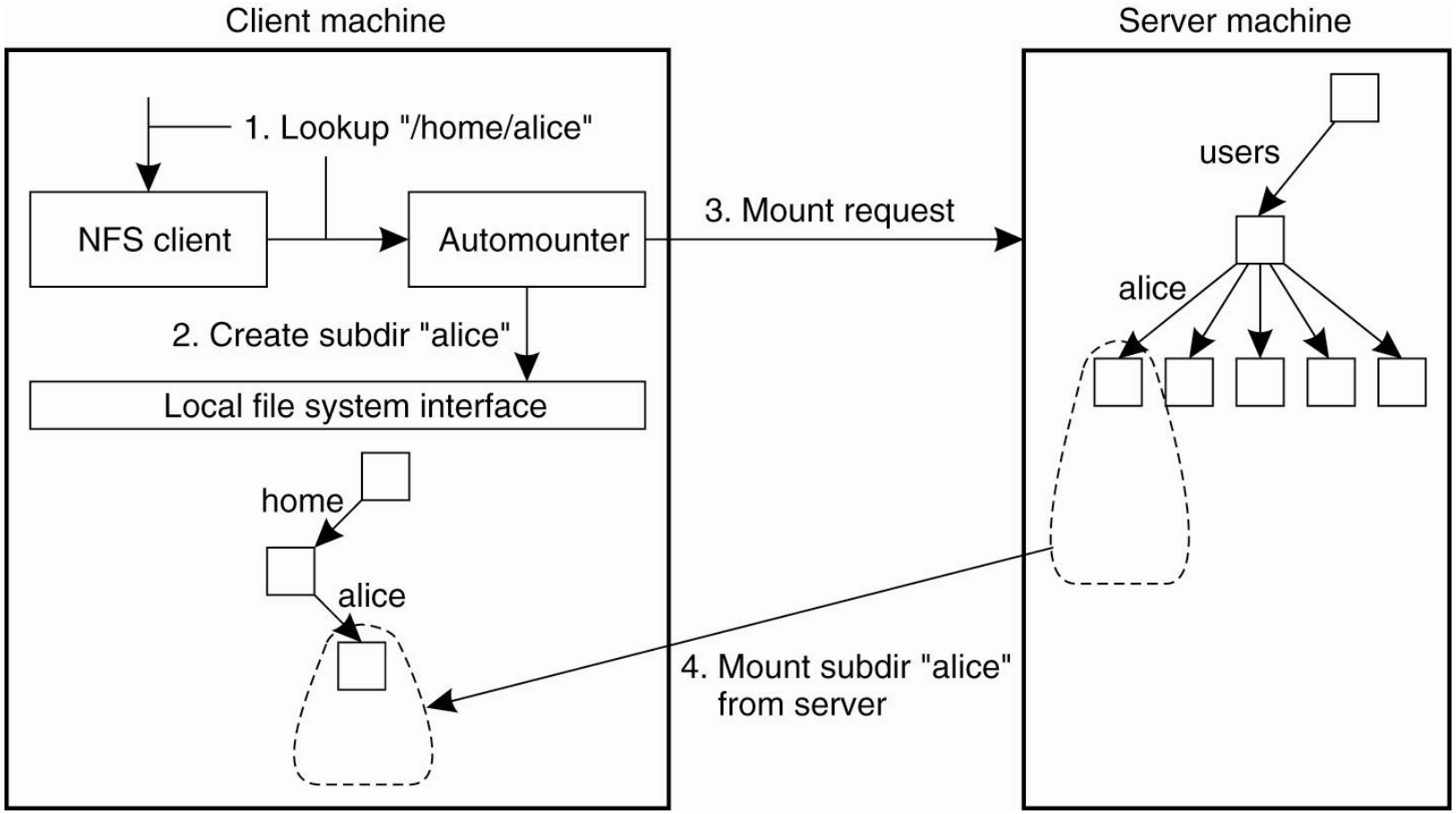
Name resolution iterative in NFSv3

Mount Time

When to mount remote file system?

- Boot time
  - + Consistent view of FS
  - - May do unnecessary work
  - - Takes longer to boot
- On explicit command by user
  - + Give user control
  - - Require user to know & do things
- Automount
  - + “Subdirectories magically appear”
  - - “Subdirectories magically appear”

Automounting (1)



Automount daemon *always* involved in *every* file operation!

Figure 11-13. A simple automounter for NFS.

Semantics of File Sharing (1)

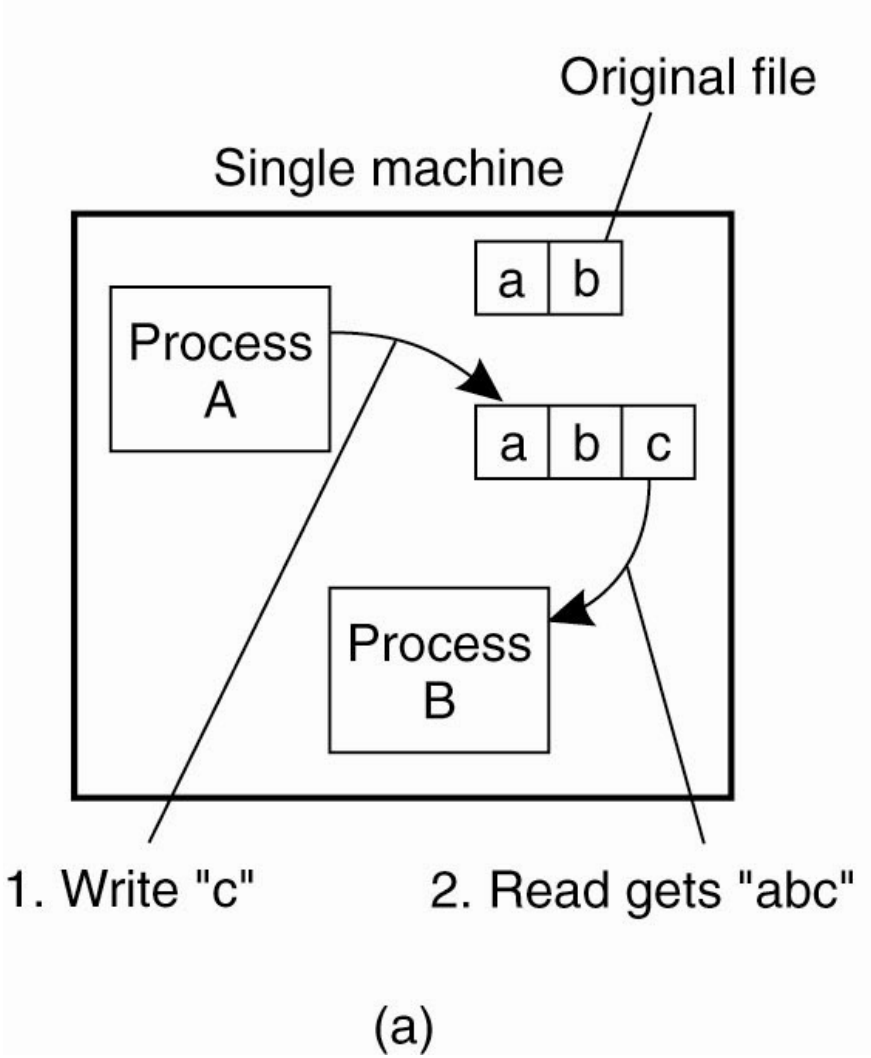


Figure 11-16. (a) On a single processor, when a read follows a write, the value returned by the read is the value just written.

Semantics of File Sharing (2)

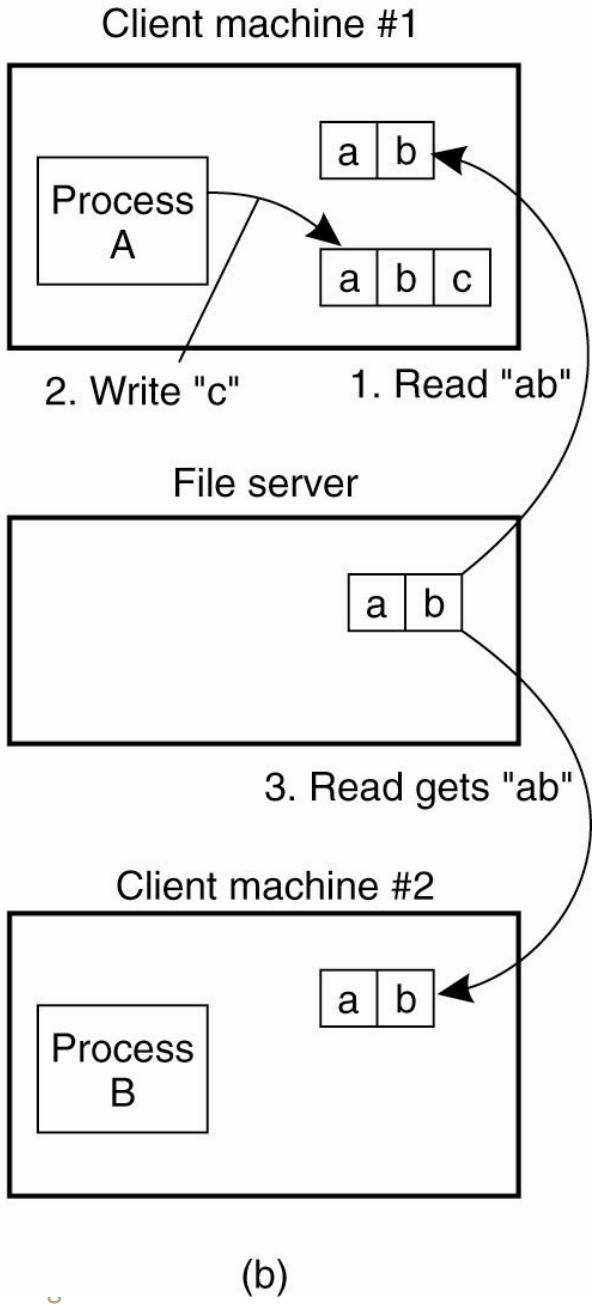


Figure 11-16. (b) In a distributed system with caching, obsolete values may be returned.

Client-Side Caching (1)

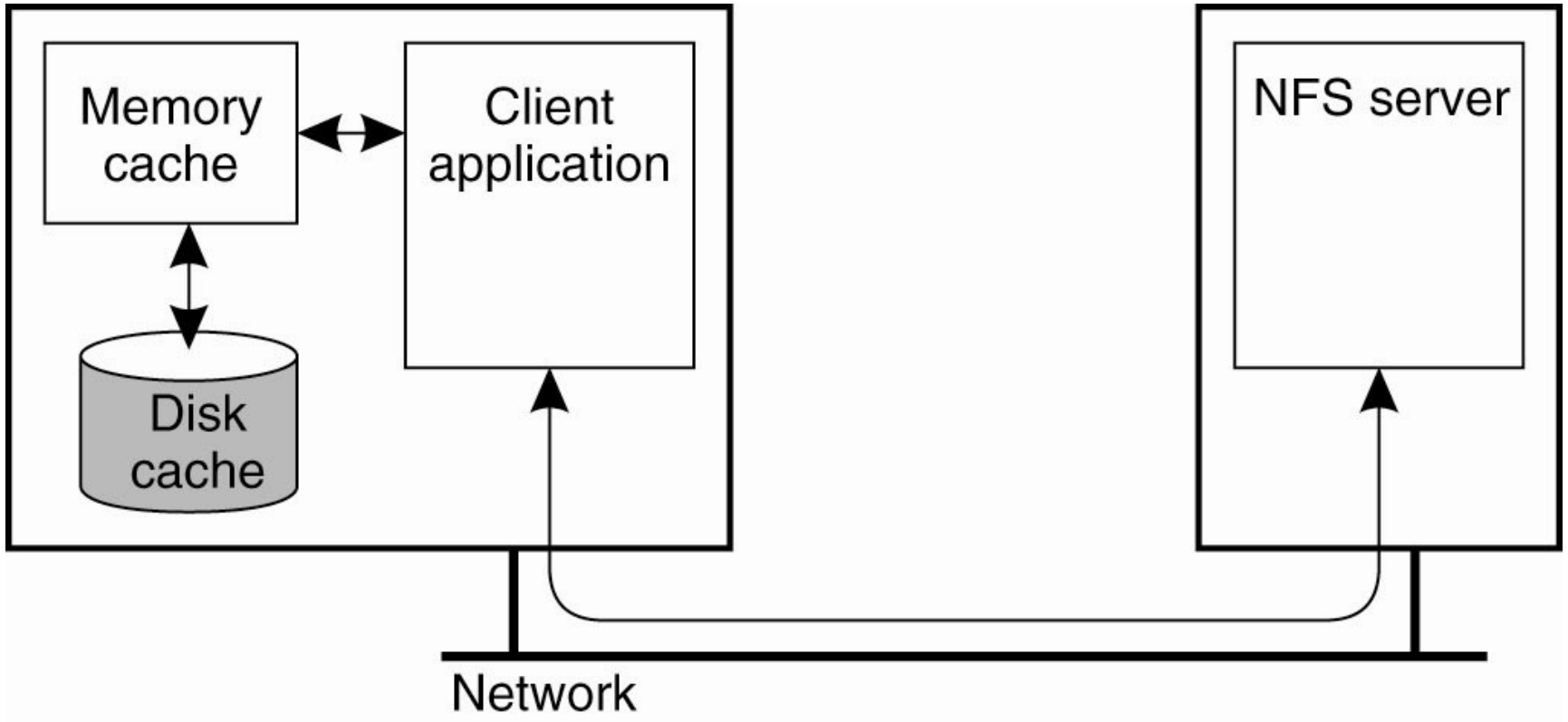


Figure 11-21. Client-side caching in NFS.

Client-Side Caching (2)

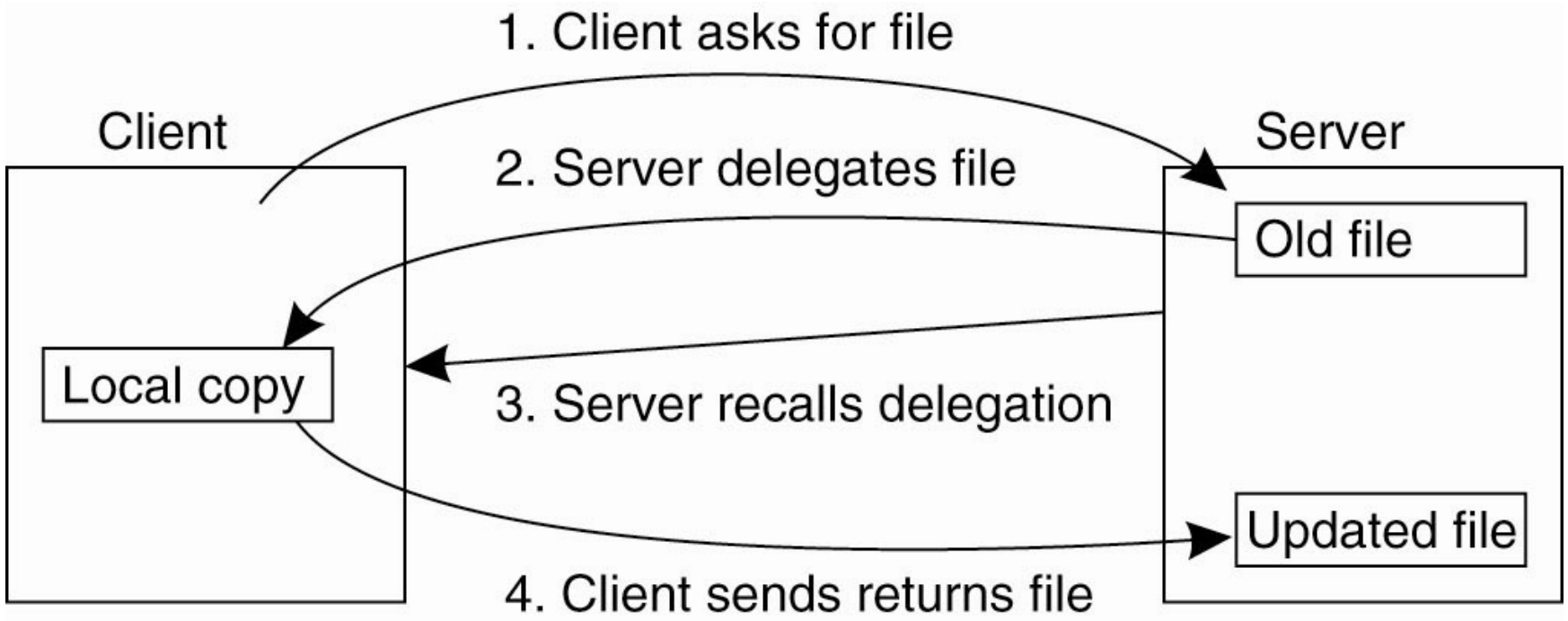


Figure 11-22. Using the NFSv4 callback mechanism to recall file delegation.

Hadoop File System

B. Ramamurthy

Reference

- The Hadoop Distributed File System: Architecture and Design by Apache Foundation Inc.

Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Can be built out of commodity hardware

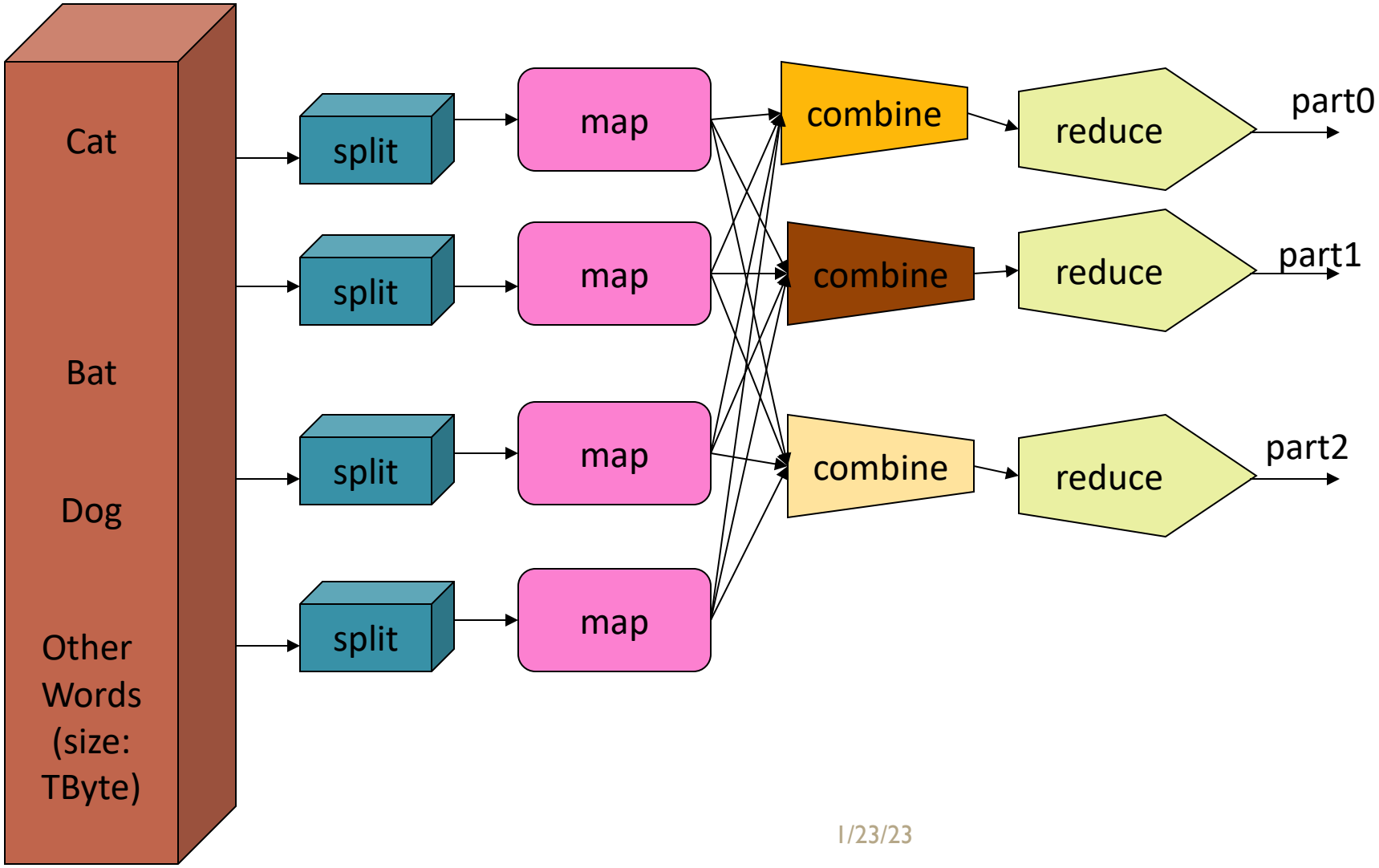
Fault tolerance

- Failure is the norm rather than exception
- A HDFS instance may consist of thousands of server machines, each storing part of the file system's data.
- Since we have huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Data Characteristics

- Streaming data access
- Applications need streaming access to data
- Batch processing rather than interactive user access.
- Large data sets and files: gigabytes to terabytes size
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
- Tens of millions of files in a single instance
- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency
- A map-reduce application or web-crawler application fits perfectly with this model.

MapReduce

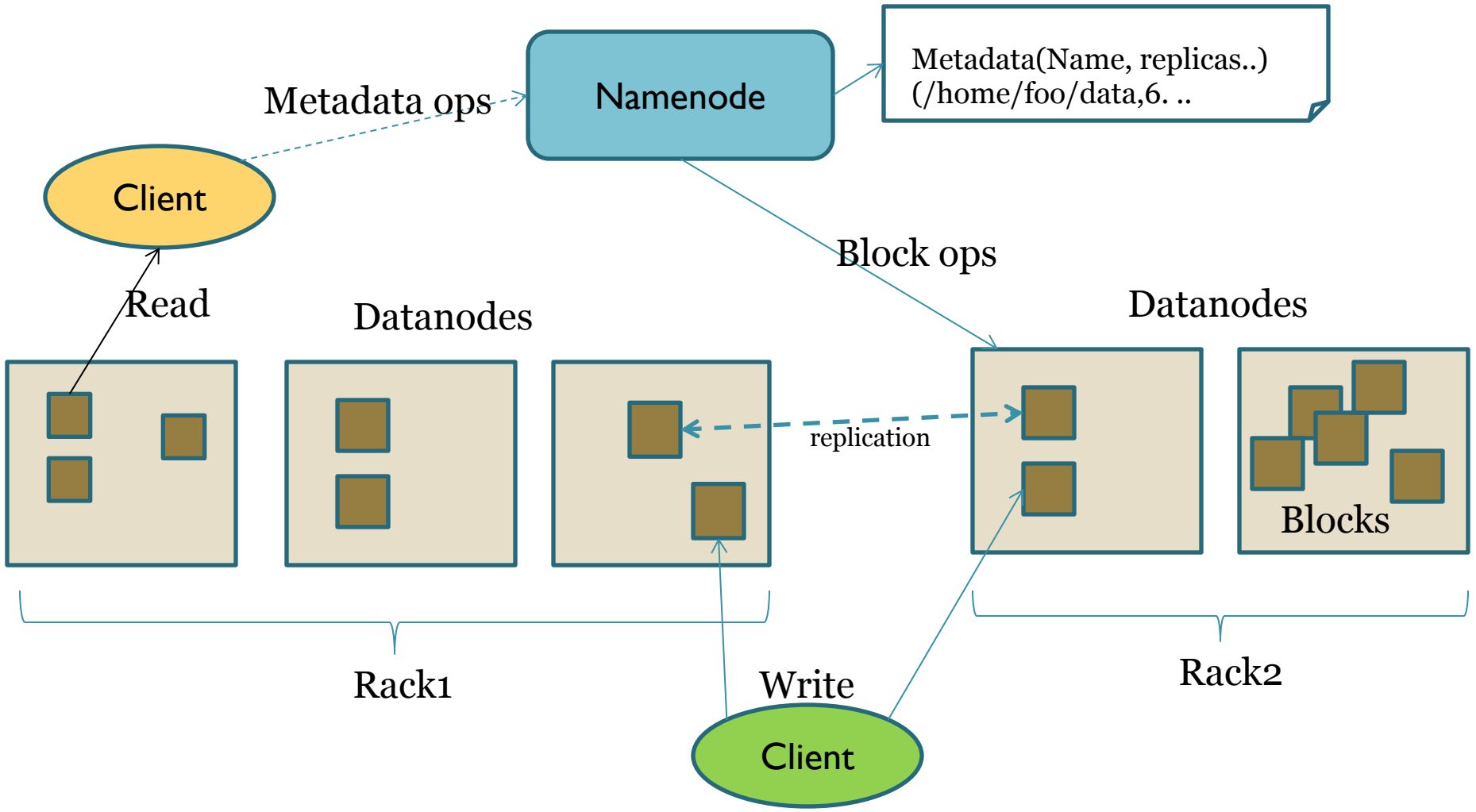

Cat
Bat
Dog
Other Words (size: TByte)
split
split
split
split
map
map
map
map
combine
combine
combine
reduce
reduce
reduce
part0
part1
part2

ARCHITECTURE

Namenode and Datanodes

- Master/slave architecture
- HDFS cluster consists of a single **Namenode**, a master server that manages the file system namespace and regulates access to files by clients.
- There are a number of **DataNodes** usually one per node in a cluster.
- The DataNodes manage storage attached to the nodes that they run on.
- HDFS exposes a file system namespace and allows user data to be stored in files.
- A file is split into one or more blocks and set of blocks are stored in DataNodes.
- DataNodes: serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode.

HDFS Architecture



File system Namespace

- Hierarchical file system with directories and files
- Create, remove, move, rename etc.
- Namenode maintains the file system
- Any meta information changes to the file system recorded by the Namenode.
- An application can specify the number of replicas of the file needed: replication factor of the file. This information is stored in the Namenode.

Data Replication

- HDFS is designed to store very large files across machines in a large cluster.
- Each file is a sequence of blocks.
- All blocks in the file except the last are of the same size.
- Blocks are replicated for fault tolerance.
- Block size and replicas are configurable per file.
- The Namenode receives a Heartbeat and a BlockReport from each DataNode in the cluster.
- BlockReport contains all the blocks on a Datanode.

Replica Selection

- Replica selection for READ operation: HDFS tries to minimize the bandwidth consumption and latency.
- If there is a replica on the Reader node then that is preferred.
- HDFS cluster may span multiple data centers: replica in the local data center is preferred over the remote one.

Safemode Startup

- On startup Namenode enters Safemode.
- Replication of data blocks do not occur in Safemode.
- Each DataNode checks in with Heartbeat and BlockReport.
- Namenode verifies that each block has acceptable number of replicas
- After a configurable percentage of safely replicated blocks check in with the Namenode, Namenode exits Safemode.
- It then makes the list of blocks that need to be replicated.
- Namenode then proceeds to replicate these blocks to other Datanodes.

Namenode

- Keeps image of entire file system namespace and file Blockmap in memory.
- 4GB of local RAM is sufficient to support the above data structures that represent the huge number of files and directories.
- When the Namenode starts up it gets the FsImage and Editlog from its local file system, update FsImage with EditLog information and then stores a copy of the FsImage on the filesytstem as a checkpoint.
- Periodic checkpointing is done. So that the system can recover back to the last checkpointed state in case of a crash.

Datanode

- A Datanode stores data in files in its local file system.
- Datanode has no knowledge about HDFS filesystem
- It stores each block of HDFS data in a separate file.
- Datanode does not create all files in the same directory.
- It uses heuristics to determine optimal number of files per directory and creates directories appropriately:

- When the filesystem starts up it generates a list of all HDFS blocks and send this report to Namenode: Blockreport.

ROBUSTNESS

Objectives

- Primary objective of HDFS is to store data reliably in the presence of failures.
- Three common failures are: Namenode failure, Datanode failure and network partition.

DataNode failure and heartbeat

- A network partition can cause a subset of Datanodes to lose connectivity with the Namenode.
- Namenode detects this condition by the absence of a Heartbeat message.
- Namenode marks Datanodes without Hearbeat and does not send any IO requests to them.
- Any data registered to the failed Datanode is not available to the HDFS.
- Also the death of a Datanode may cause replication factor of some of the blocks to fall below their specified value.

Re-replication

- The necessity for re-replication may arise due to:
  - A Datanode may become unavailable,
  - A replica may become corrupted,
  - A hard disk on a Datanode may fail, or
  - The replication factor on the block may be increased.

Data Integrity

- Consider a situation: a block of data fetched from Datanode arrives corrupted.
- This corruption may occur because of faults in a storage device, network faults, or buggy software.
- When a clients retrieves the contents of file, it verifies that the corresponding checksums match.
- If does not match, the client can retrieve the block from a replica.

Application Programming Interface

- HDFS provides Java API for application to use.
- Python access is also used in many applications.
- A C language wrapper for Java API is also available.
- A HTTP browser can be used to browse the files of a HDFS instance.

FS Shell, Admin and Browser Interface

- HDFS organizes its data in files and directories.
- It provides a command line interface called the FS shell that lets the user interact with data in the HDFS.
- The syntax of the commands is similar to bash and csh.
- Example: to create a directory /foodir

/bin/hadoop dfs –mkdir /foodir

- There is also DFSAdmin interface available
- Browser interface is also available to view the namespace.

Google File System

Motivation

- Store big data reliably
- Allow parallel processing of big data

Assumptions

- Inexpensive components that often fail
- Large files
- Large streaming reads and small random reads
- Large sequential writes
- Multiple users append to the same file
- High bandwidth is more important than low latency.

Architecture

- Chunks
  - File → chunks → location of chunks (replicas)
- Master server
  - Single master
  - Keep metadata
  - accept requests on metadata
  - Most management activities
- Chunk servers
  - Multiple
  - Keep chunks of data
  - Accept requests on chunk data

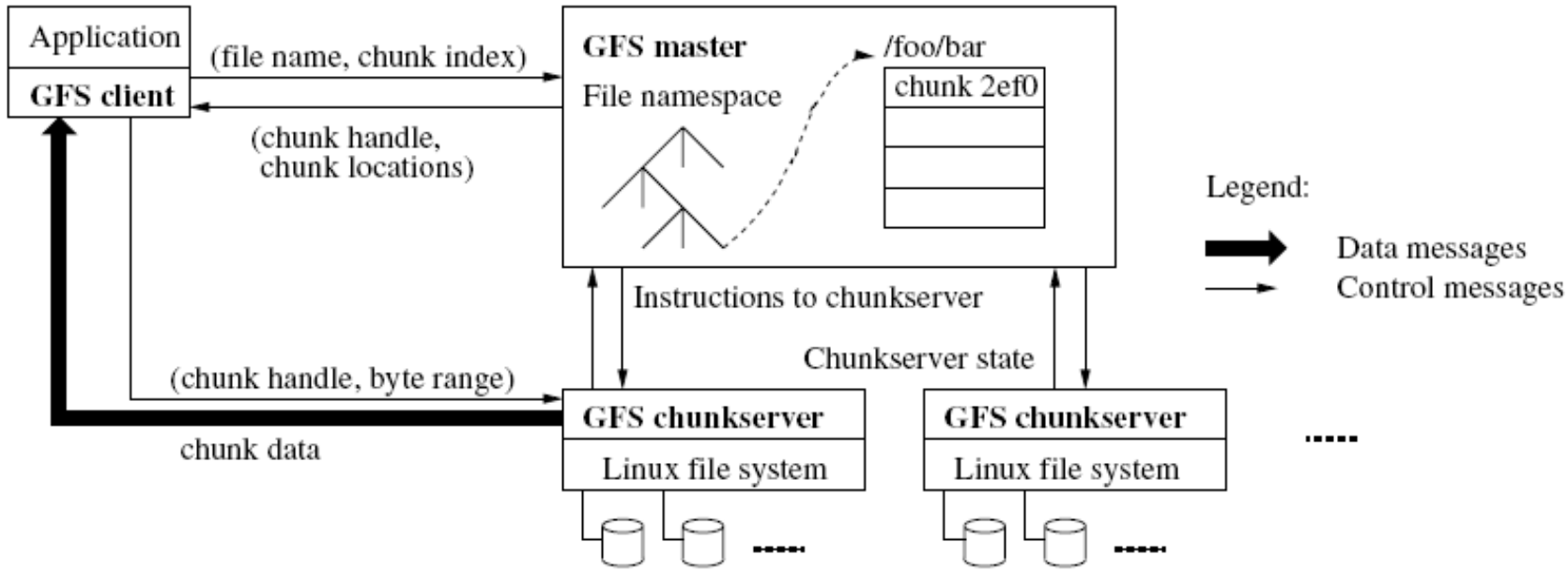


Figure 1: GFS Architecture

Design decisions

- Single master
  - Simplify design
  - Single point-of-failure
  - Limited number of files
    - Meta data kept in memory
- Large chunk size: e.g., 64M
  - advantages
    - Reduce client-master traffic
    - Reduce network overhead – less network interactions
    - Chunk index is smaller
  - Disadvantages
    - Not favor small files
    - hot spots

Master: meta data

- Metadata is stored in memory
- Namespaces
  - Directory → physical location
- Files → chunks → chunk locations
- Chunk locations
  - Not stored by master, sent by chunk servers
- Operation log

Master Operations

- All namespace operations
  - Name lookup
  - Create/remove directories/files, etc
- Manage chunk replicas
  - Placement decision
  - Create new chunks & replicas
  - Balance load across all chunkservers
  - Garbage claim

Master: chunk replica placement

- Goals: maximize reliability, availability and bandwidth utilization
- Physical location matters
  - Lowest cost within the same rack
  - "Distance": # of network switches
- In practice (hadoop)
  - If we have 3 replicas
  - Two chunks in the same rack
  - The third one in another rack
- Choice of chunkservers
  - Low average disk utilization
  - Limited # of recent writes → distribute write traffic

- Re-replication
  - Lost replicas for many reasons
  - Prioritized: low # of replicas, live files, actively used chunks
  - Following the same principle to place

- Rebalancing
  - Redistribute replicas periodically
    - Better disk utilization
    - Load balancing

Consistency

- It is expensive to maintain strict consistency
  - duplicates, distributed
- GFS uses a relaxed consistency
  - Better support for appending
  - Checkpointing

Fault Tolerance

- High availability
  - Fast recovery
  - Chunk replication
  - Master replication: inactive backup
- Data integrity
  - Checksumming
  - Incremental update checksum to improve performance
    - A chunk is split into 64K-byte blocks
    - Update checksum after adding a block

Discussion

- Advantages
  - Works well for large data processing
  - Using cheap commodity servers
- Tradeoffs
  - Single master design
  - Reads most, appends most

- Next class
  - MapReduce programming model
  - Hadoop
  - Hadoop programming