

# Distributed File Systems

# Definition of a DFS

- DFS: multiple users, multiple sites, and (possibly) distributed storage of files.
- Benefits
  - File sharing
  - Uniform view of system from different clients
  - Centralized administration
- Goals of a distributed file system
  - Network Transparency (access transparency)
  - Availability

# Goals

- **Network (Access) Transparency**
  - Users should be able to access files over a network as easily as if the files were stored locally.
  - Users should not have to know the physical location of a file to access it.
- Transparency can be addressed through naming and file mounting mechanisms

# Components of Access Transparency

- Location Transparency: file name doesn't specify physical location
- Location Independence: files can be moved to new physical location, no need to change references to them. (A name is independent of its addresses)
- Location independence → location transparency, but the reverse is not necessarily true.

# Goals

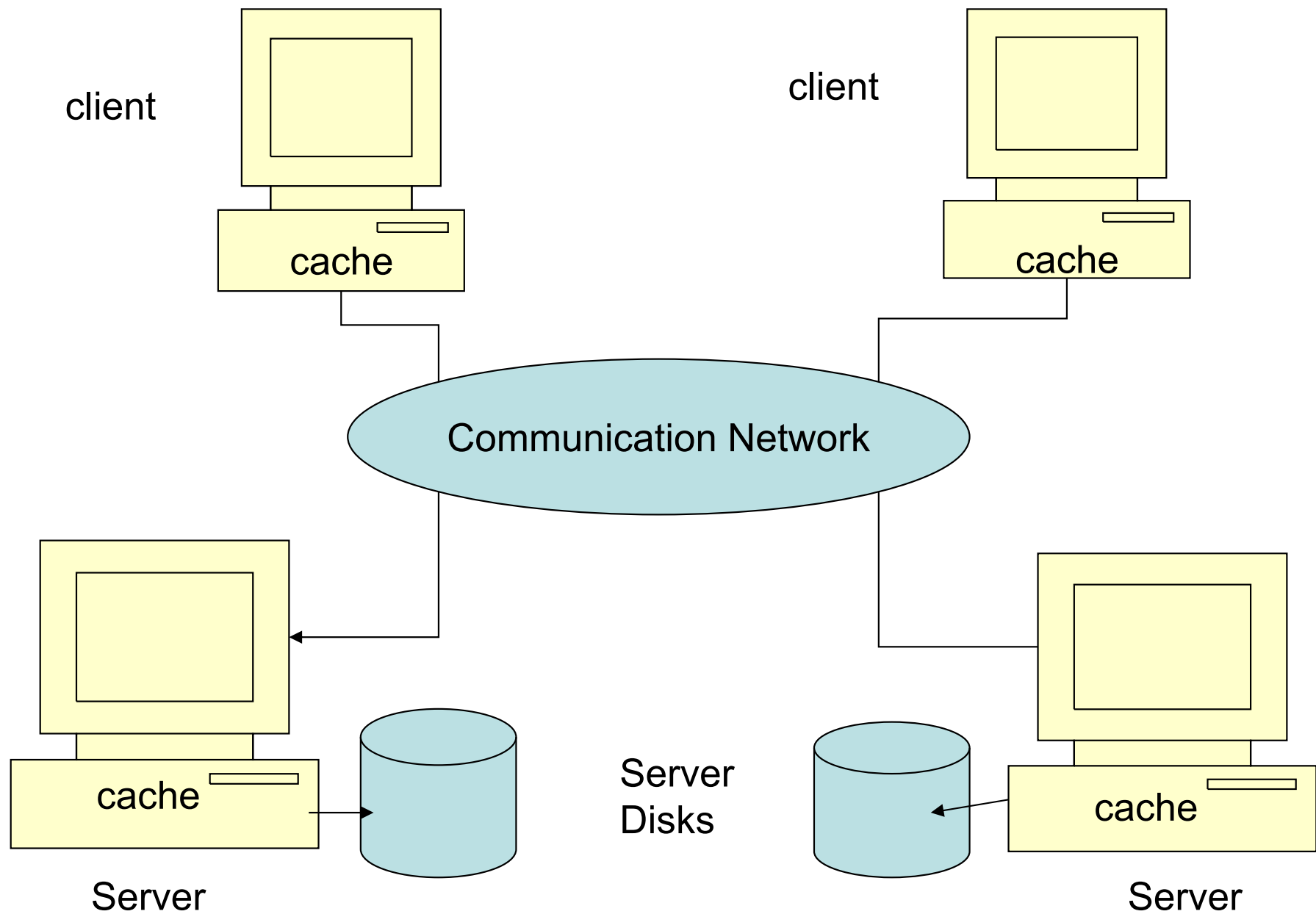
- Availability: files should be easily and quickly accessible.
- The number of users, system failures, or other consequences of distribution shouldn't compromise the availability.
- Addressed mainly through replication.

# Architectures

- Client-Server
  - Traditional; e.g. Sun Microsystem Network File System (NFS)
  - Cluster-Based Client-Server; e.g., Google File System (GFS)
- Symmetric
  - Fully decentralized; based on peer-to-peer technology
  - e.g., Ivy (uses a Chord DHT approach)

# Client-Server Architecture

- One or more machines (file servers) manage the file system.
- Files are stored on disks at the servers
- Requests for file operations are made from clients to the servers.
- Client-server systems centralize storage and management; P2P systems decentralize it.



Architecture of a distributed file system: client-server model



# Sun's Network File System

- Sun's NFS for many years was the most widely used distributed file system.
  - NFSv3: original architecture
  - NFSv4: introduced in 2003

# Overview

- NFS goals:
  - Each file server presents a standard view of its local file system
  - transparent access to remote files
  - compatibility with multiple operating systems and platforms.
  - easy crash recovery at server (at least v1-v3)
- Originally UNIX based; now available for most operating systems.
- NFS communication protocols lets processes running in different environments share a file system.

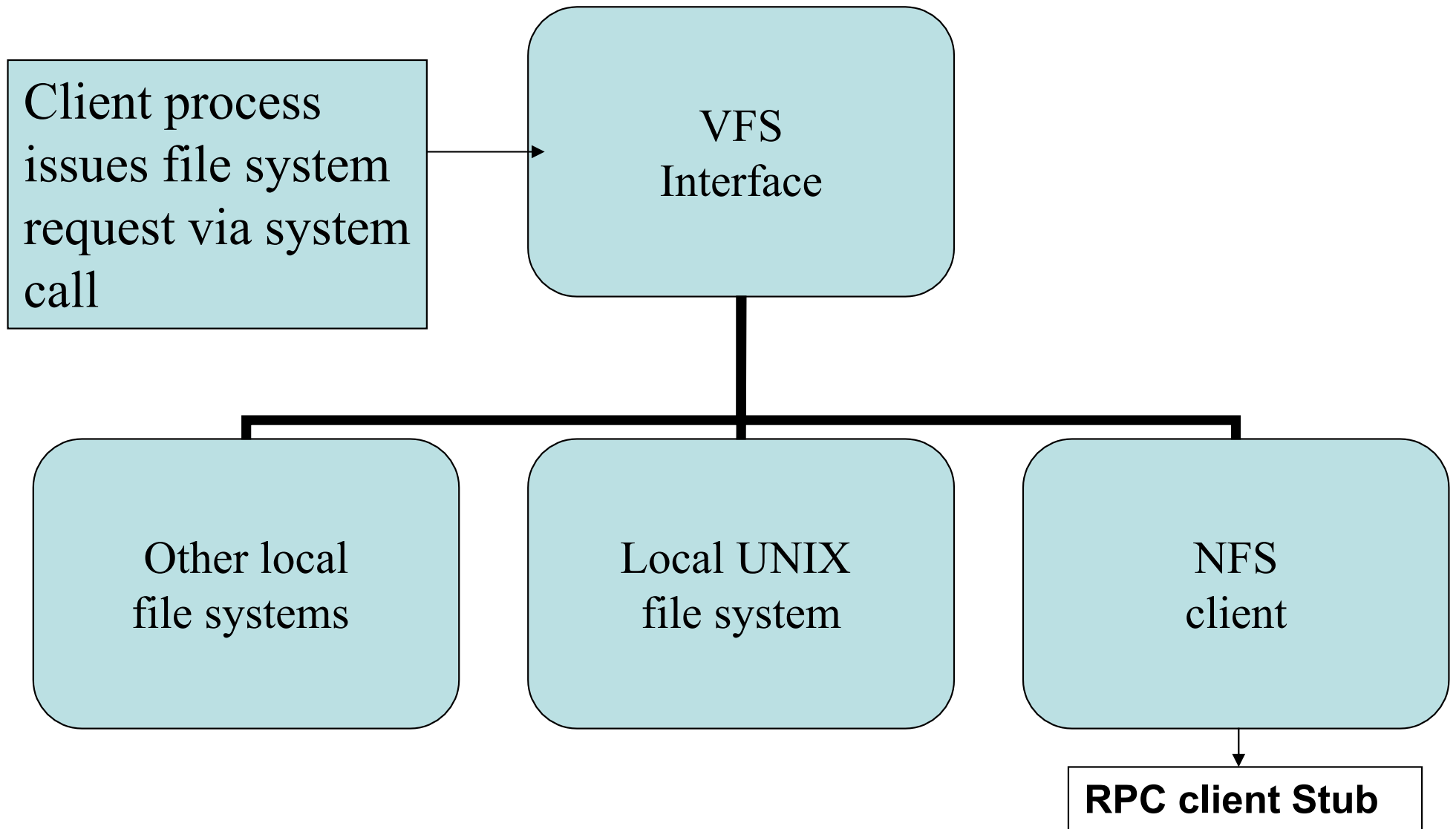
# Access Models

- Clients access the server transparently through an interface similar to the local file system interface
- Client-side caching may be used to save time and network traffic
- Server defines and performs all file operations

# NFS - System Architecture

- Virtual File System (VFS) acts as an interface between the operating system's system call layer and all file systems on a node.
- VFS is used today on virtually all operating systems as the interface to different local and distributed file systems.
- The user interface to NFS is the same as the interface to local file systems. The calls go to the VFS layer, which passes them either to a local file system or to the NFS client

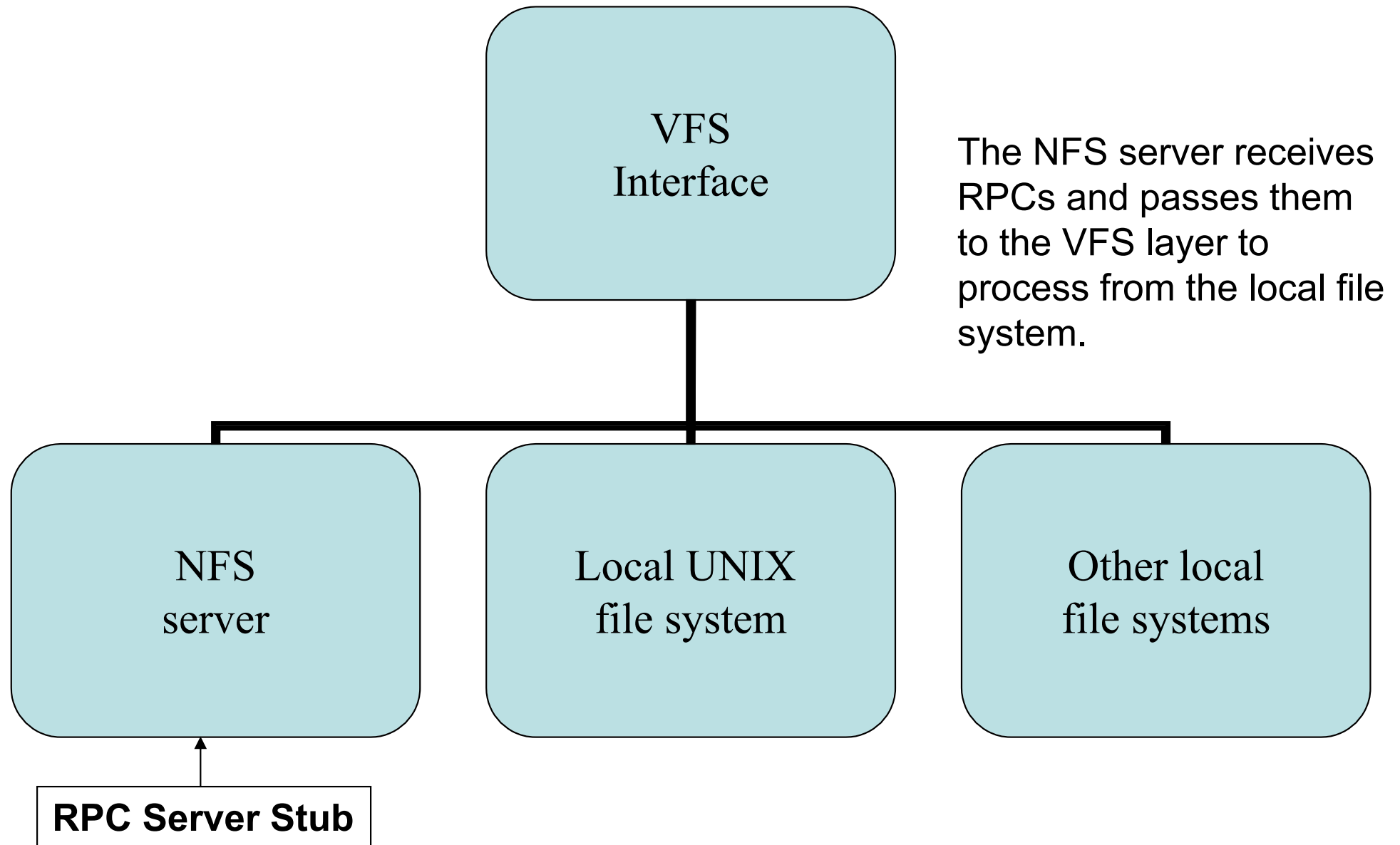
# Client-Side Interface to NFS



# NFS Client/Server Communication

- The NFS client communicates with the server using RPCs
  - File system operations are implemented as remote procedure calls
- At the server: an RPC server stub receives the request, “un-marshalls” the parameters & passes them to the NFS server, which creates a request to the server’s VFS layer.
- The VFS layer performs the operation on the local file system and the results are passed back to the client.

# Server-Side Interface to NFS



# NFS as a **Stateless** Server

- NFS servers historically did not retain any information about past requests.
- Consequence: crashes weren't too painful
  - If server crashed, it had no tables to rebuild – just reboot and go
- Disadvantage: client has to maintain all state information; messages are longer than they would be otherwise.
- NFSv4 is **stateful**



# Advantages/Disadvantages

- Stateless Servers

- Fault tolerant
- No open/close RPC required
- No need for server to waste time or space maintaining tables of state information
- Quick recovery from server crashes

- Stateful Servers

- Messages to server are shorter (no need to transmit state information)
- Supports file locking
- Supports idempotency (don't repeat actions if they have been done)

# File System Model

- NFS implements a file system model that is almost identical to a UNIX system.
  - Files are structured as a sequence of bytes
  - File system is hierarchically structured
  - Supports hard links and symbolic links
  - Implements most file operations that UNIX supports
    - Some differences between NFSv3 and NFSv4

# File Create/Open/Close

- Create: v3 Yes, v4 No;  
Open: v3 No, v4 Yes;  
v4 creates a new file if an *open* operation is executed on a non-existent file
- Close: v3 No and v4 Yes
- Rationale: v3 was stateless; didn't keep information about open files.

# Disadvantages of Statelessness

- The server cannot inform the client whether or not a request has been processed.
  - Consider implications for lost request/lost replies when operations are not idempotent
- File locking (to guarantee one writer at a time) is not possible
  - NFS got around this problem by supporting a separate lock manager.

# NFSv4

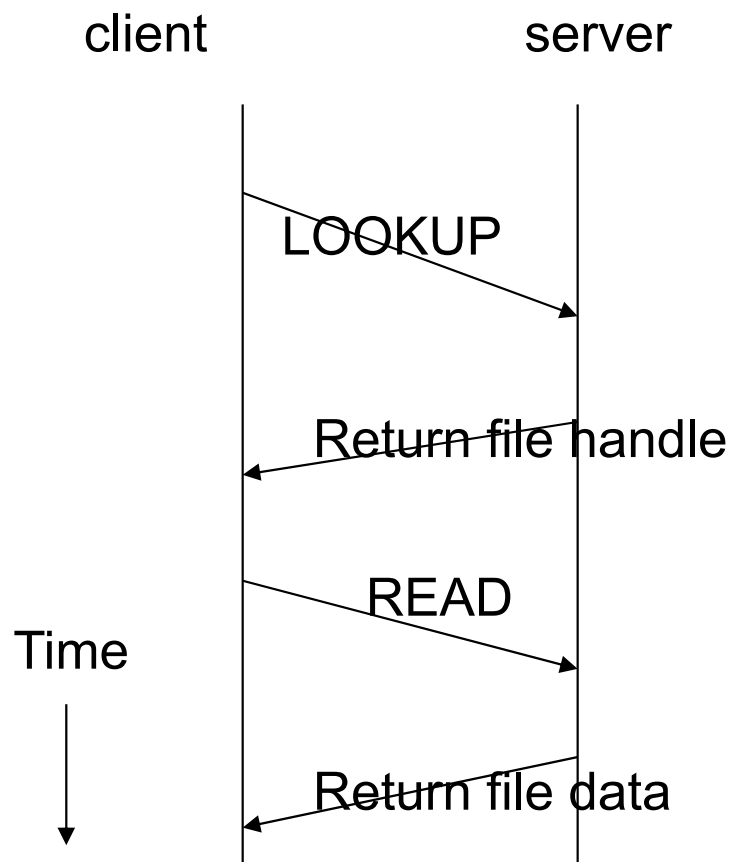
- Maintains some minimal state about its clients; e.g., enough to execute authentication protocols
- Stateful servers are better equipped to run over wide area networks, because they are better able to manage consistency issues that arise when clients are allowed to cache portions of files locally

# Communication

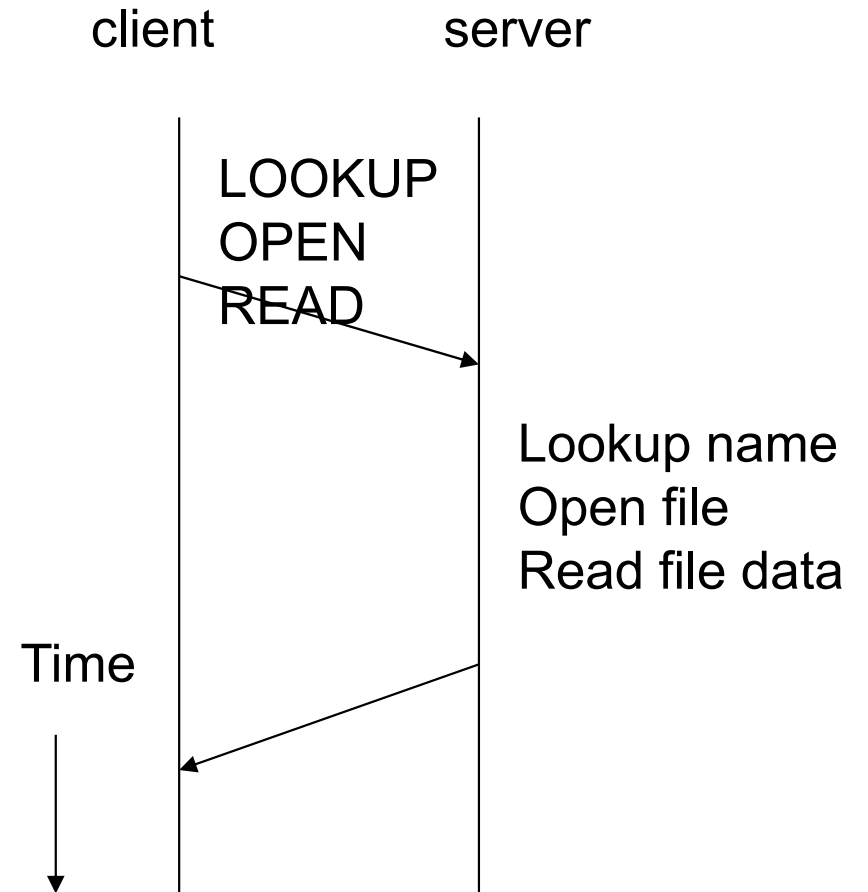
- Usually based on remote procedure calls, or some variation.
- Rationale: RPC communication makes the DFS independent of local operating systems, network protocols, and other issues that distract from the main issue.

# RPC in NFS

- Client-server communication in NFS is based on Open Network Computing RPC (ONC RPC) protocols.
- Each file system operation is represented as an RPC. Pre-version 4 NFS required one RPC at a time, so server didn't have to remember any state.
- NFSv4 supports compound procedures (several RPCs grouped together)



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



(b)  
Reading data from a file  
in NFS version 4



# Naming

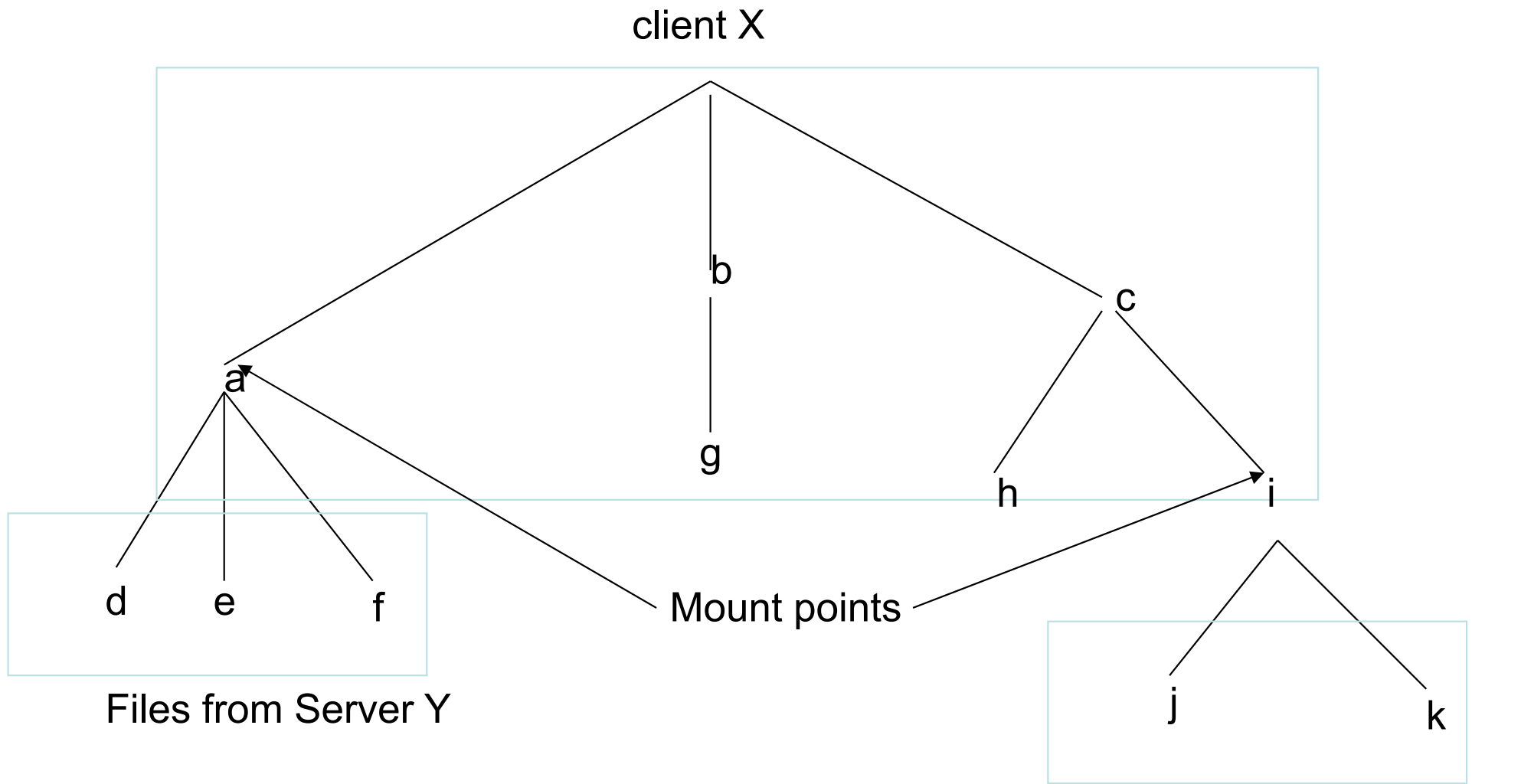
- NFS is used as a typical example of naming in a DFS.
- Virtually all support a hierarchical namespace organization.
- NFS naming model strives to provide transparent client access to remote file systems.

# Goal

- Network (Access) Transparency
  - Users should be able to access files over a network as easily as if the files were stored locally.
  - Users should not have to know the location of a file to access it.
- Transparency can be addressed through naming and file mounting mechanisms

# Mounting

- Servers *export* file systems; i.e, make them available to clients
- Client machines can attach a remote FS (directory or subdirectory) to the local FS at any point in its directory hierarchy.
- When a FS is mounted, the client can reference files by the local path name – no reference to remote host location, although files remain physically located at the remote site.
- *Mount tables* keep track of the actual physical location of the files.



Files d, e, and f are on server Y; files j and k are on server Z, but from the perspective of server X all are part of the file system at that location

# File Handles

- A file handle is a reference to a file that is created by the server when the file is created.
  - It is independent of the actual file name
  - It is not known to the client (although the client must know the size)
  - It is used by the file system for all internal references to the file.

# Benefits of File Handles

- There is a uniform format for the file identifier inside the file system (128 bytes, in NFSv4)
- Clients can store the handle locally after an initial reference and avoid the lookup process on subsequent file operations

# Cluster-based or Clustered File System

- A distributed file system that consists of several servers that share the responsibilities of the system, as opposed to a single server (possibly replicated).
- The design decisions for a cluster-based systems are mostly related to how the data is distributed across the cluster and how it is managed.

# Cluster-Based DFS

- Some cluster-based systems organize the clusters in an application specific manner
- For file systems used primarily for parallel applications, the data in a file might be striped across several servers so it can be read in parallel.
- Or, it might make more sense to partition the file system itself – some portion of the total number of files are stored on each server.
- For systems that process huge numbers of requests; e.g., large data centers, reliability and management issues take precedence.
  - e.g., Google File System



# Google Operations

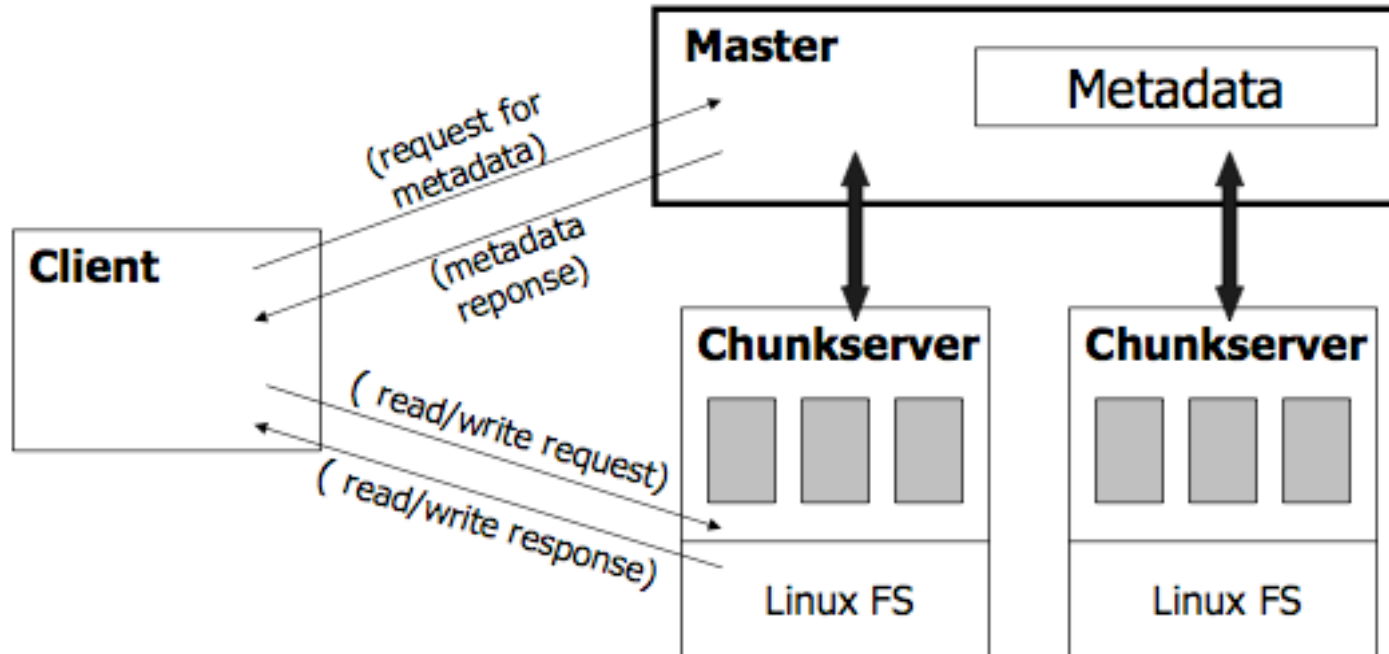
- More than 15K commodity class PCs
- Multiple clusters distributed worldwide
- Thousands of queries served per second
- One query reads 100's of MB of data
- One query consumes 10's of billions of CPU cycles
- Google stores dozens of copy of the entire web
- **Conclusion: Need large, distributed, highly fault tolerant file system, i.e. GFS (Google File System)**

# Motivation behind GFS

- Fault tolerance and auto-recovery need to be built into the systems (monitoring, error detection, fault tolerance, automatic recovery)
- because problems are very often caused by application bugs, OS bugs, human errors, and the failure of disks, memory, connectors, networking, and power supplies.
- Standard I/O assumptions (e.g. block size) has to be re-examined
- Record appends are the frequent form of writing
- Google applications and GFS should be co- designed

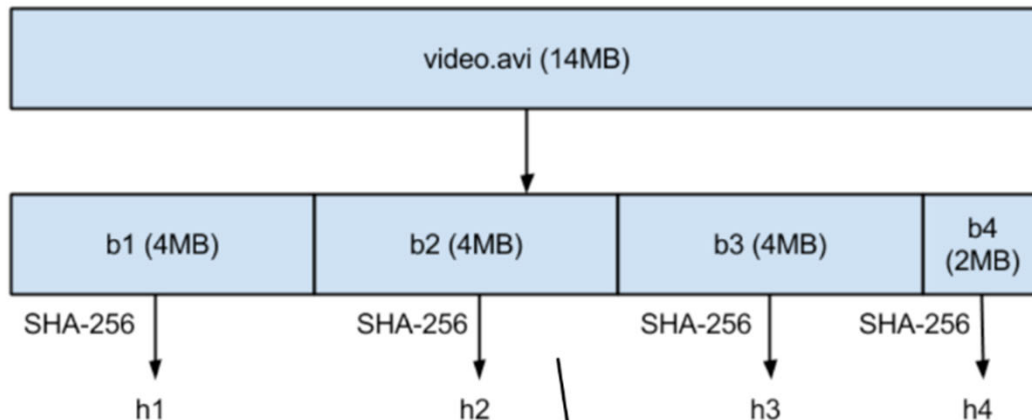
# GFS Architecture

- In the GFS
  - A master process maintains the metadata
  - A lower layer (i.e. a set of chunkservers) stores the data in unit called *chunks*

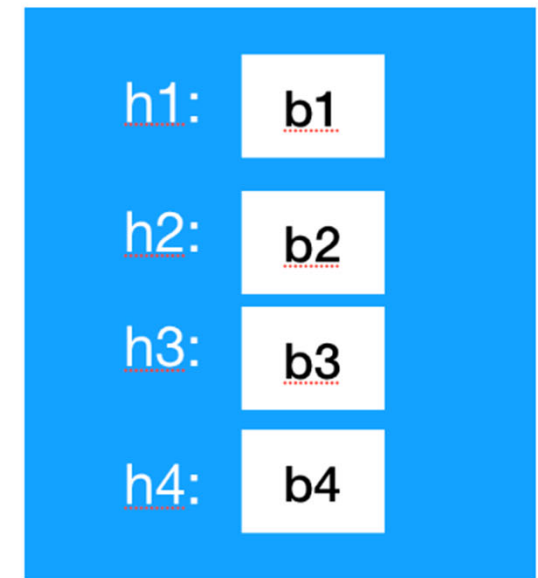


# Dropbox architecture

sheldon/folder1/video.avi



Block Server



Metadata Server

Namespace ID	sheldon
Namespace relative path	folder1
Blocklist	h1,h2,h3,h4
Journal ID	1

# GFS Architecture: Chunk

- **Chunk:**
  - Similar to block, much larger than typical file system block size
  - **Size: 64 MB!**
    - Why so big, compare to few KBs block size of OS file systems in general?
      - Reduces client's need to contact with the master
      - On a large chunk a client can perform many operations
      - Reduces the size of the metadata stored in the master (less chunks less metadata in the master!)
      - No internal fragmentation due to lazy space allocation
    - Disadvantages:
      - Some small file consists of a small number of chunks can be accessed so many times!
      - In practice: not a major issue, as google applications mostly read large multi-chunk files sequentially
      - Solution: Fixed by storing such files with a high replication factor

# Chunk contd.

## – Chunk

- Stored in chunkserver as file
- Chunk handle (~ chunk file name) used to reference link
- Chunk replicas across multiple chunkservers
- Note: There are hundreds of chunkservers in a GFS cluster distributed over multiple racks

# GFS Architecture: Master

- Master:
  - A single process running on a separate machine
  - Stores all metadata
    - File and chunk namespace
    - File to chunk mappings
    - Chunk location information
    - Access control information
    - Chunk version numbers
    - Etc

# GFS Architecture:

## Master <-> Chunkserver communication

- Master and chunkserver communicate regularly to obtain state:
  - Is chunkserver down?
  - Are there disk failure on chunkserver?
  - Are any replicas corrupted?
  - Which chunk replicas does chunkserver store?
- Master sends instructions to chunkserver
  - Delete existing chunk
  - Create new chunk



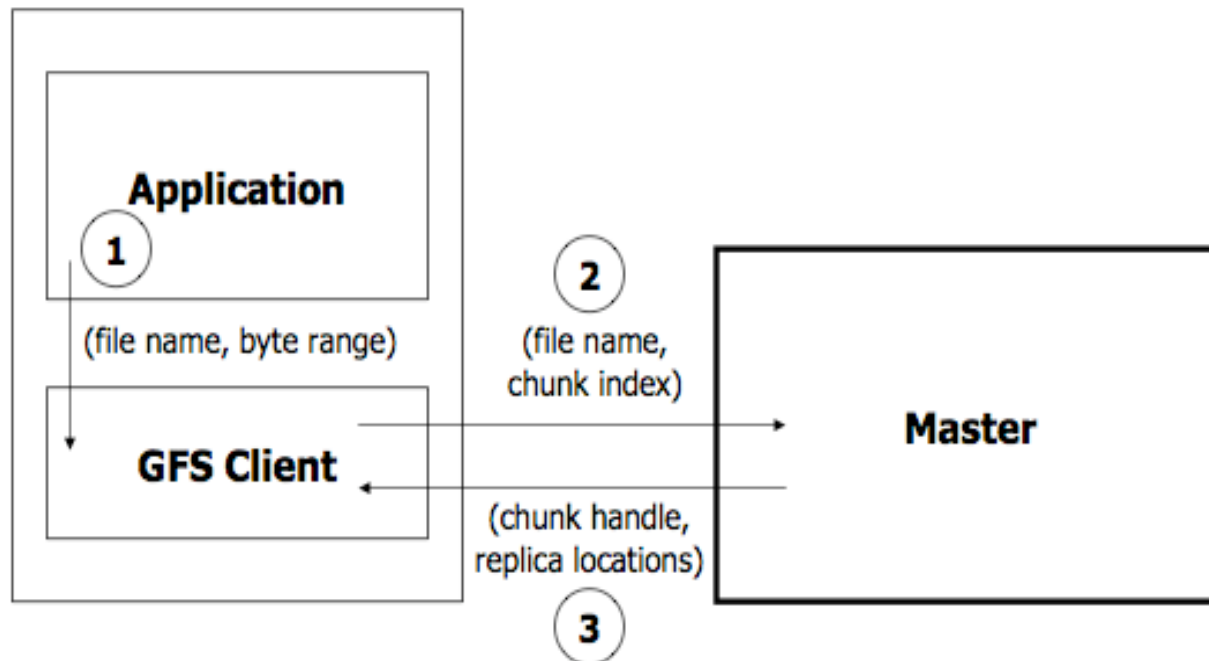
# GFS Architecture:

## Master <-> Chunkserver communication

- Server Requests
  - Client retrieves metadata for operation from master
  - Read/Write data flows between client and chunkserver
  - Single master is not bottleneck, because its involvement with read/write operations is minimized
    - Metadata is stored in master's memory. The master maintains less than 64 bytes of metadata for each 64 MB chunk.

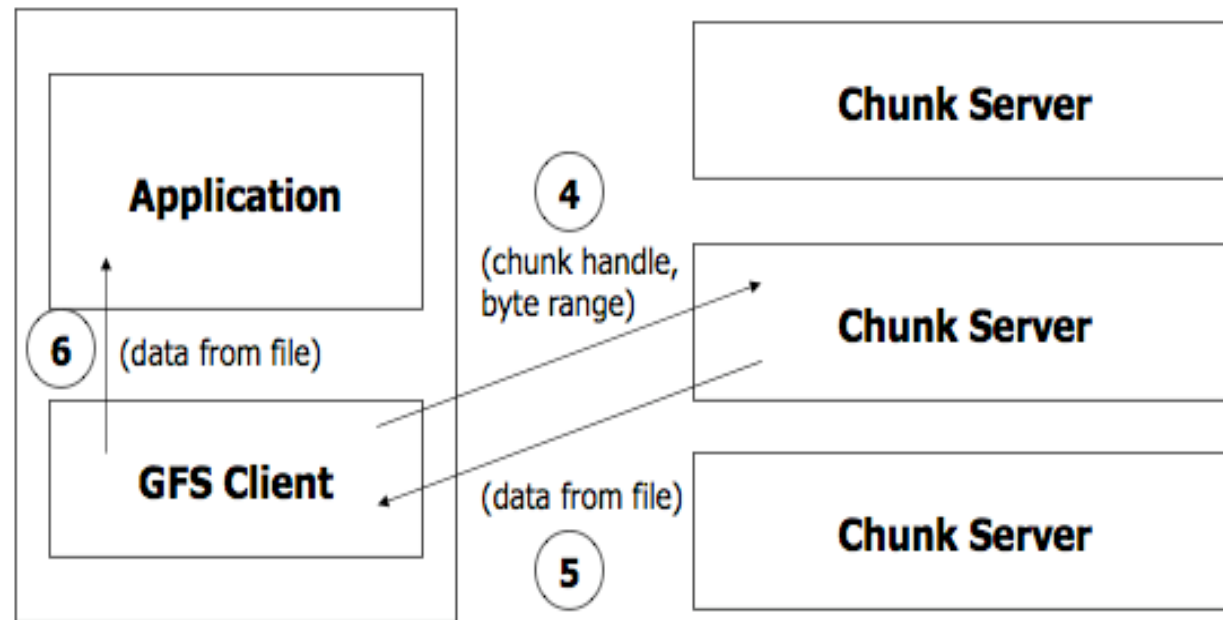
# Read Algorithm

1. Application originates the read request
2. GFS client translates the request form (filename, byte range) - > (filename, chunk index), and sends it to master
3. Master responds with chunk handle and replica locations (i.e. chunkservers where the replicas are stored)

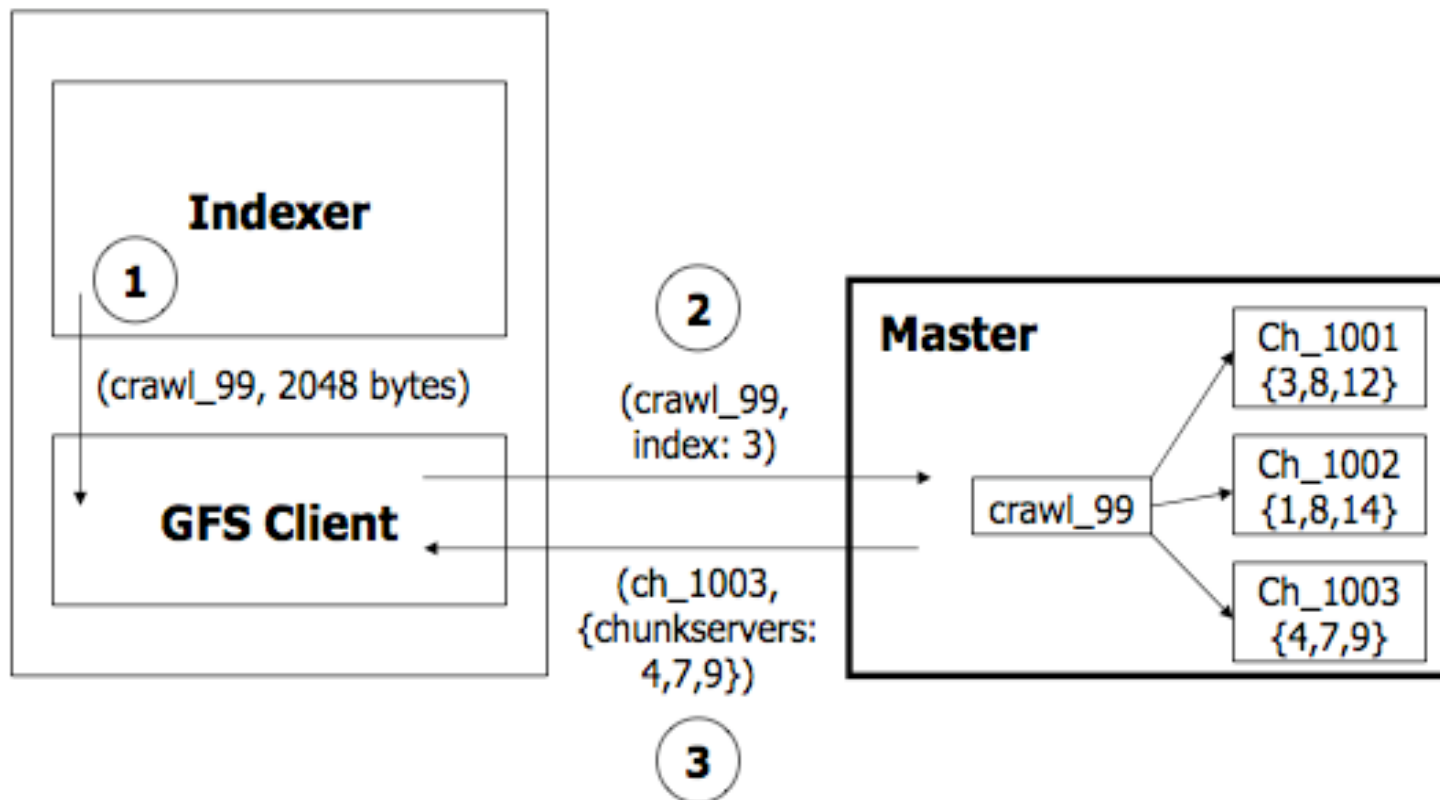


# Read Algorithm

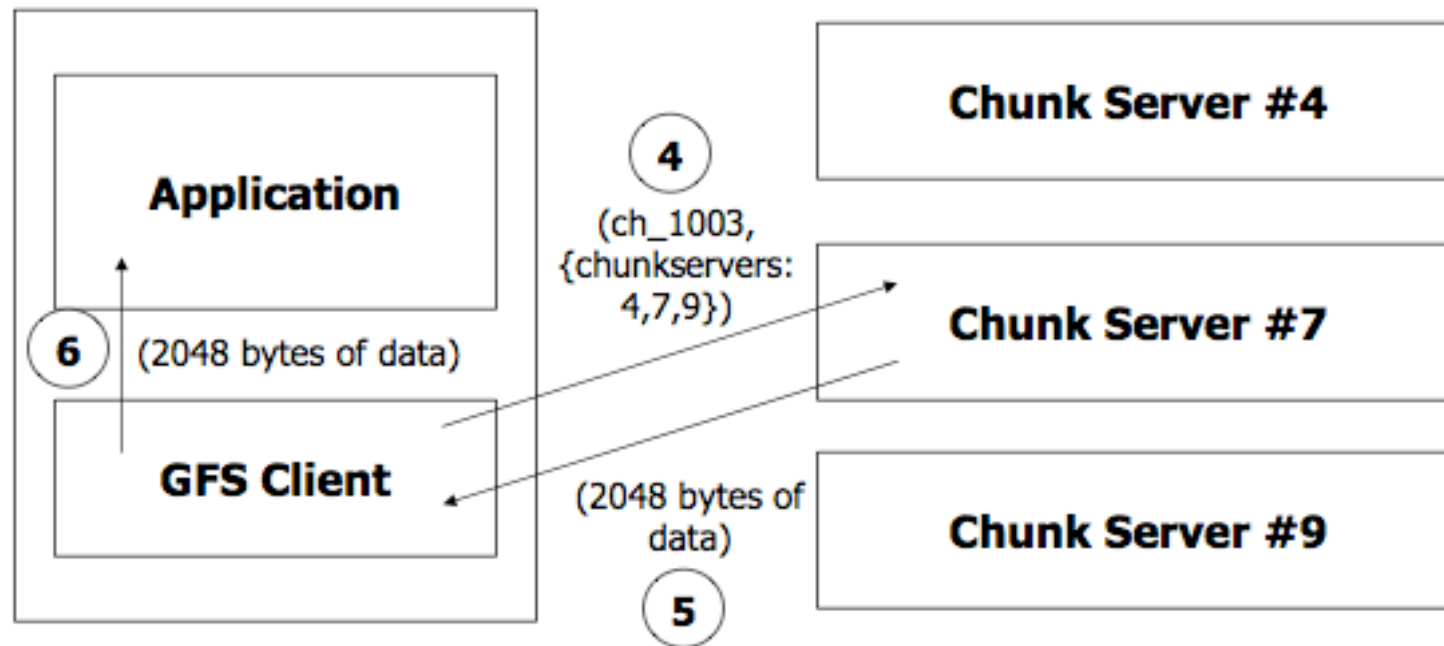
4. Client picks a location and sends the (chunk handle, byte range) request to the location
5. Chunkserver sends requested data to the client
6. Client forwards the data to the application



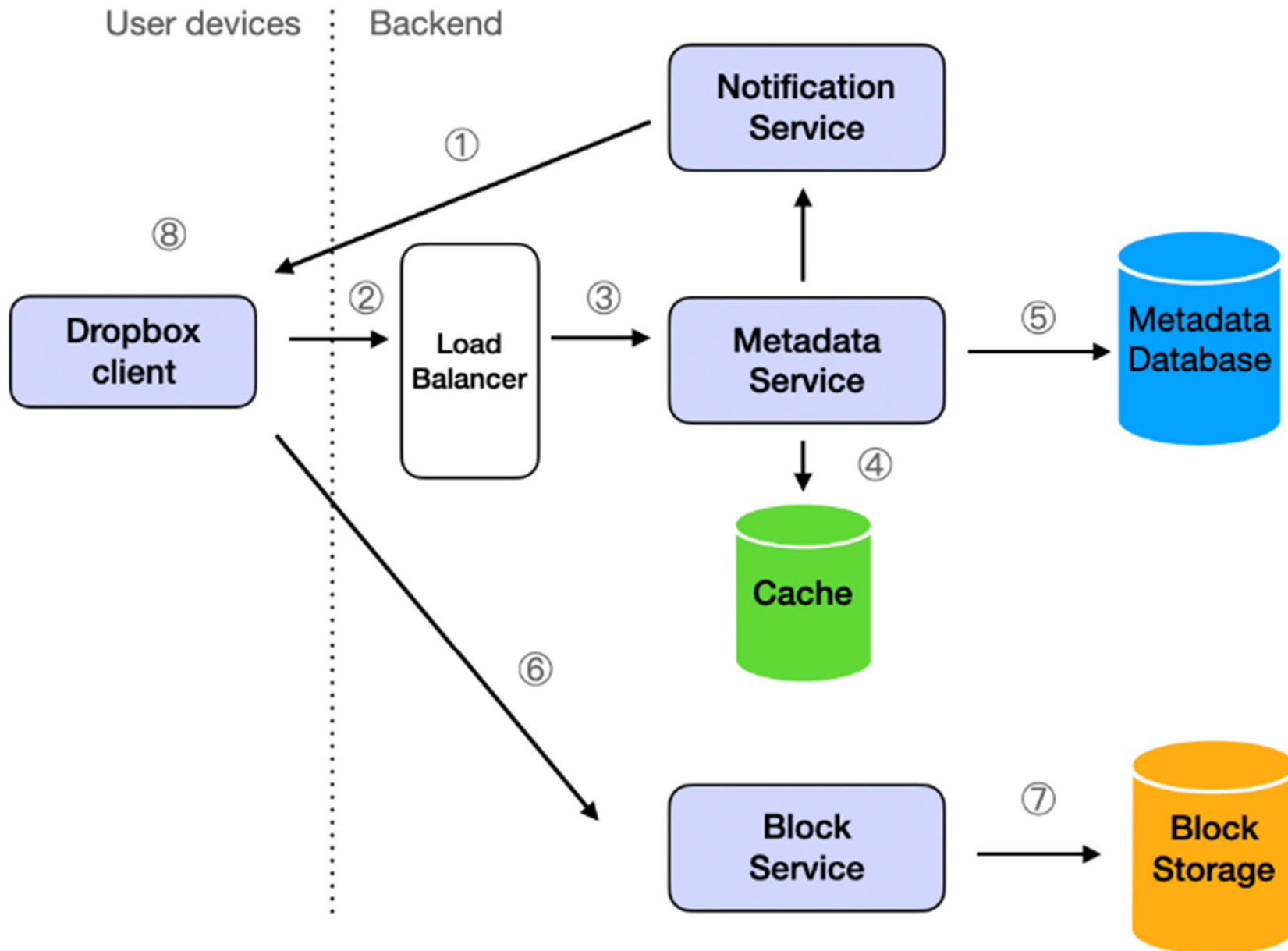
# Read Algorithm (example)



# Read Algorithm (example)

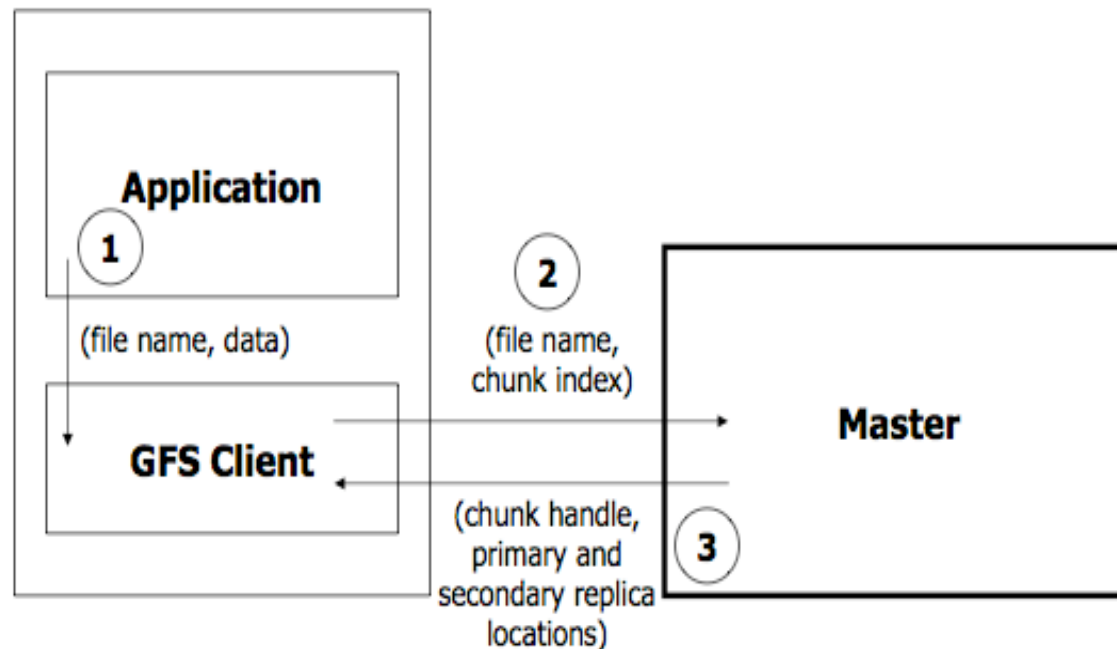


# Dropbox read path



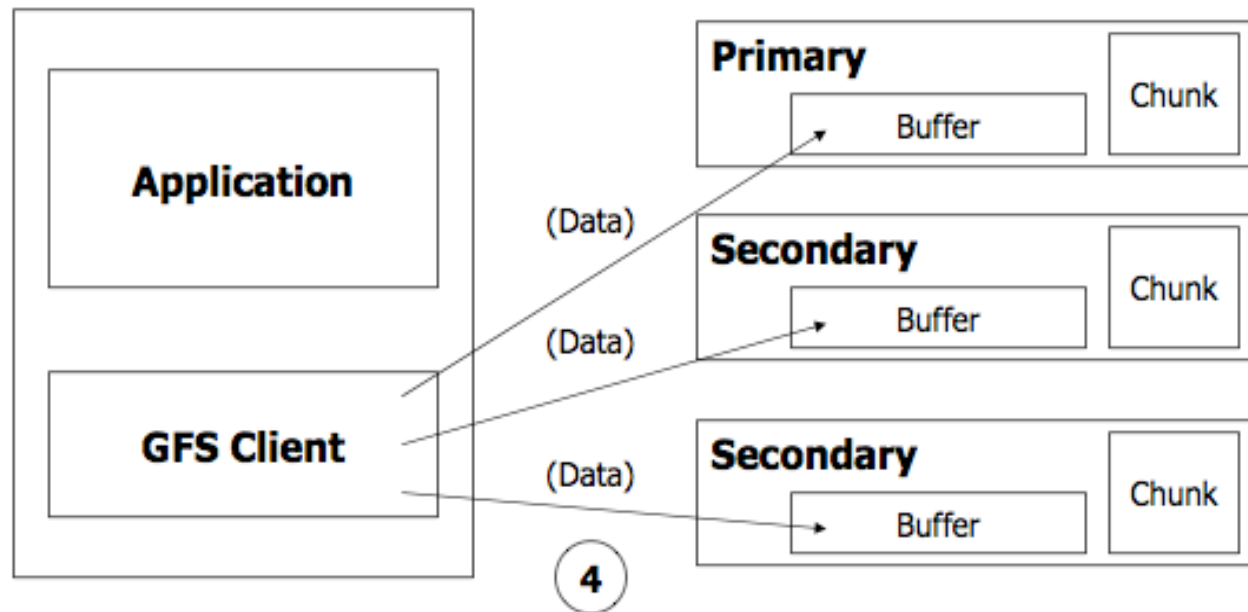
# Write Algorithm

1. Application originates the request
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master
3. Master responds with chunk handle and (primary+secondary) replica locations



# Write Algorithm

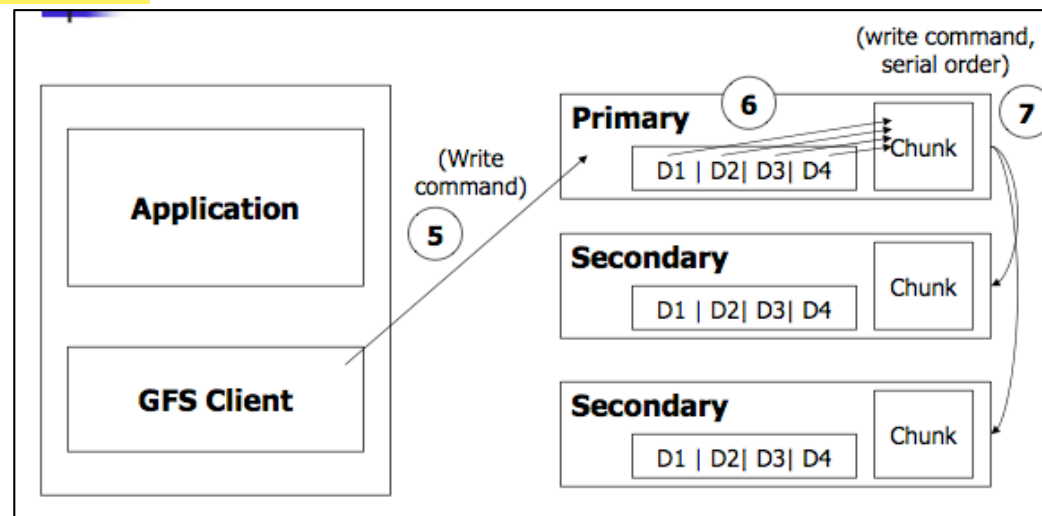
4. Client pushes write data to all locations. Data is stored in chunkservers' internal buffers





# Write Algorithm

5. Client sends write command to primary
6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk
7. Primary send the serial order to the secondaries and tell them to perform the write

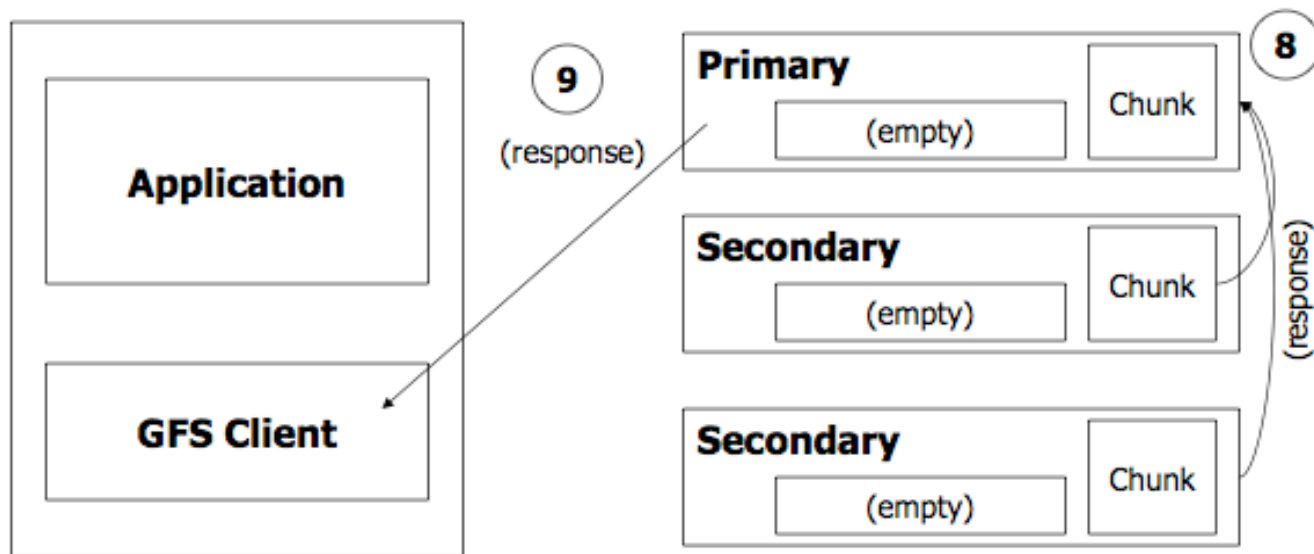


# Write Algorithm

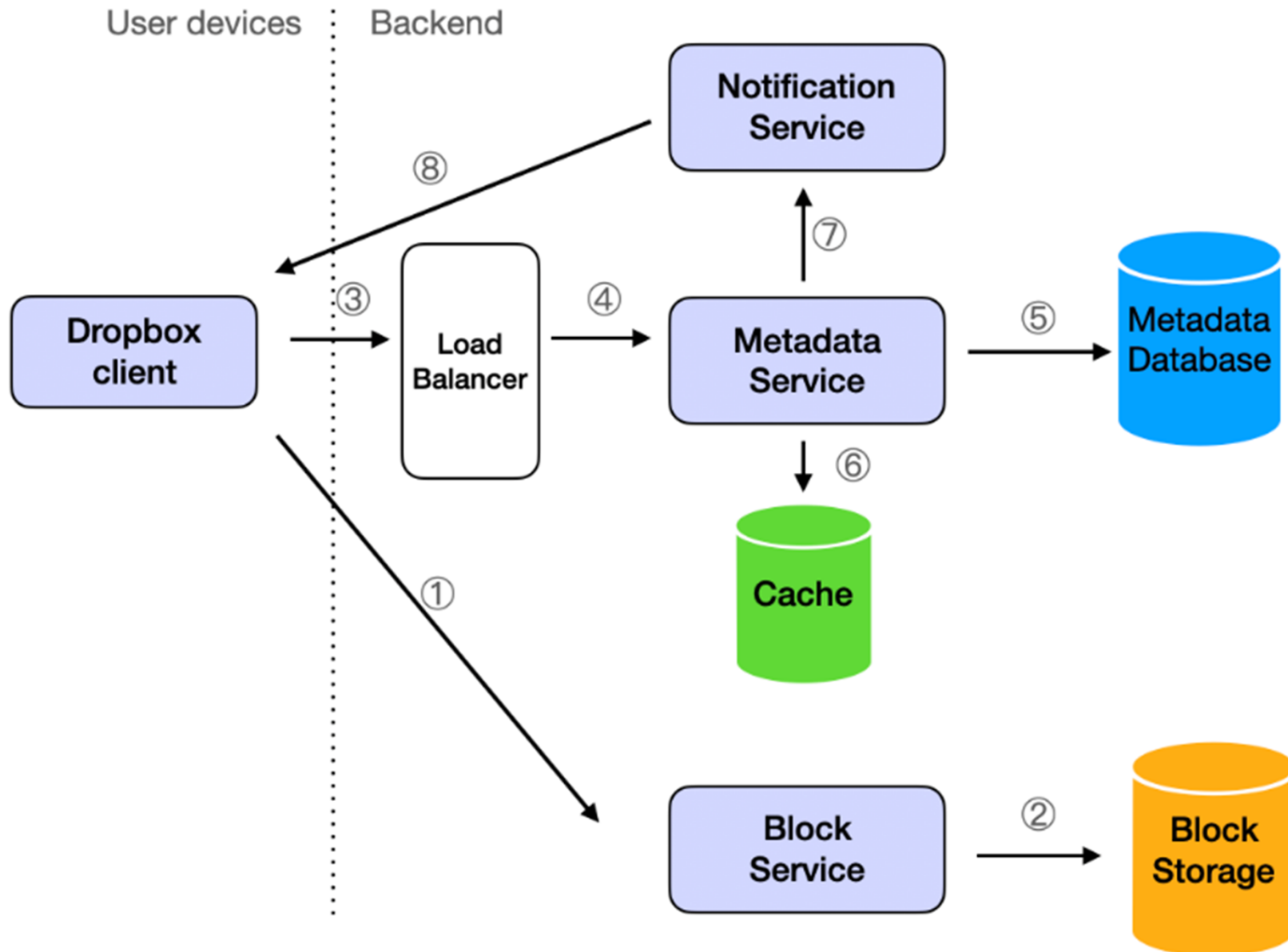
8. Secondaries respond to the primary

9. Primary respond back to the client

Note: If write fails at one of chunkservers, client is informed and retries the write




# Dropbox write path



# Record Append Algorithm

Important operation for Google

- Merging results from multiple machines in one file
- Using file as producer - consumer queue

- 
1. Application originates record append request
  2. GFS client translates requests and sends it to master
  3. Master responds with chunk handle and (primary + secondary) replica locations
  4. Client pushes write data to all locations

# Record Append Algorithm

5. Primary checks if record fits in specified chunk
6. If record doesn't fit, then the primary:
  - Pads the chunk
  - Tell secondaries to do the same
  - And informs the client
  - Client then retries the append with the next chunk
7. If record fits, then the primary:
  - Appends the record
  - Tells secondaries to do the same
  - Receives responses from secondaries
  - And sends final response to the client

# Fault Tolerance

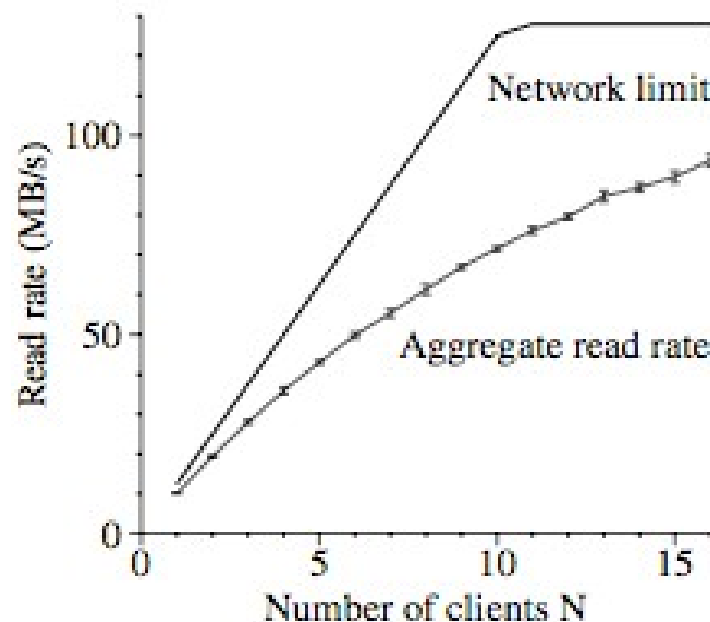
- Fast Recovery: master and chunkservers are designed to restart and restore state in few seconds
- Chunk Replication: across multiple machines, across multiple racks
- Master Mechanisms:
  - Keep log of all changes made to metadata
  - Periodic checkpoints of the log
  - Log and checkpoints replicated on multiple machines
  - Master state is replicated on multiple machines
  - *Shadow* master for reading data if *real master* is down

# Performance (Test Cluster)

- Performance measured on clusters with:
  - 1 master
  - 16 chunkservers
  - 16 clients
- Server machines connected to central switch by 100 Mbps Ethernet
- Same for client machines
- Switches connected with 1 Gbps link

# Performance (Test Cluster)

## Reads

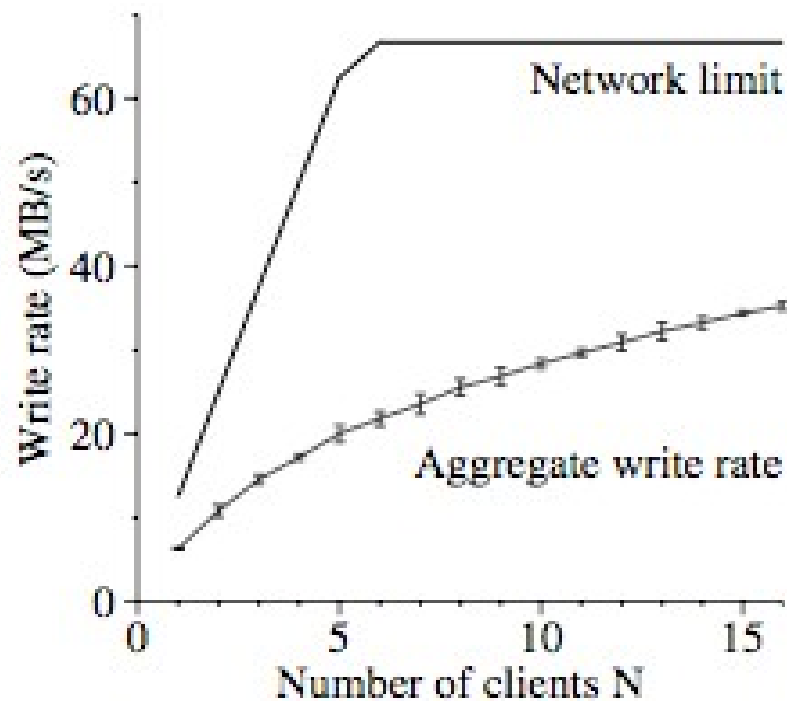


One client:  
10 MB/s, 80% of limit  
16 clients:  
6 MB/s per client, 75%  
of the limit



# Performance (Test Cluster)

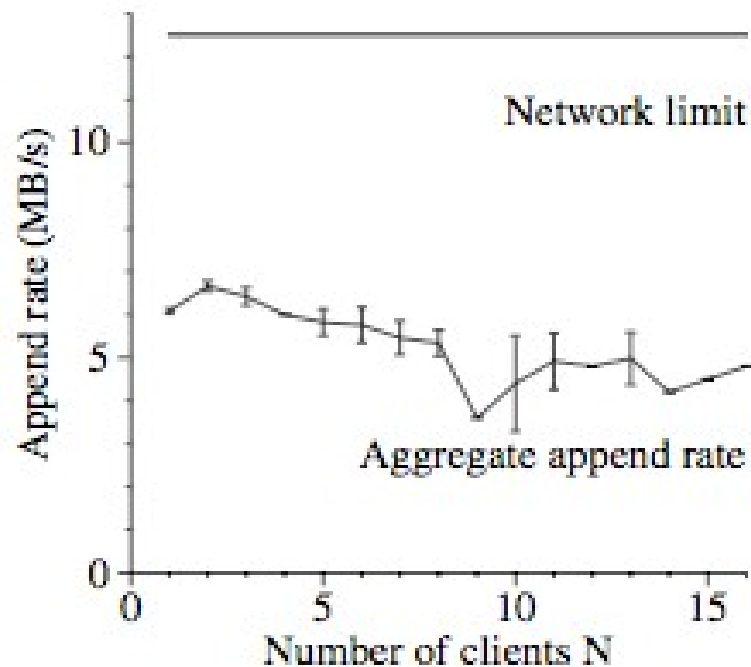
## Write



One client:  
6.3 MB/s,  
max limit: 12.5 MB/s  
16 clients:  
35 MB/s,  
2.2 MB/s per client

# Performance (Test Cluster)

## Record Append



One client: 6 MB/s  
16 clients:  
4.8 MB/s per client

# Symmetric File Systems

Peer-to-Peer

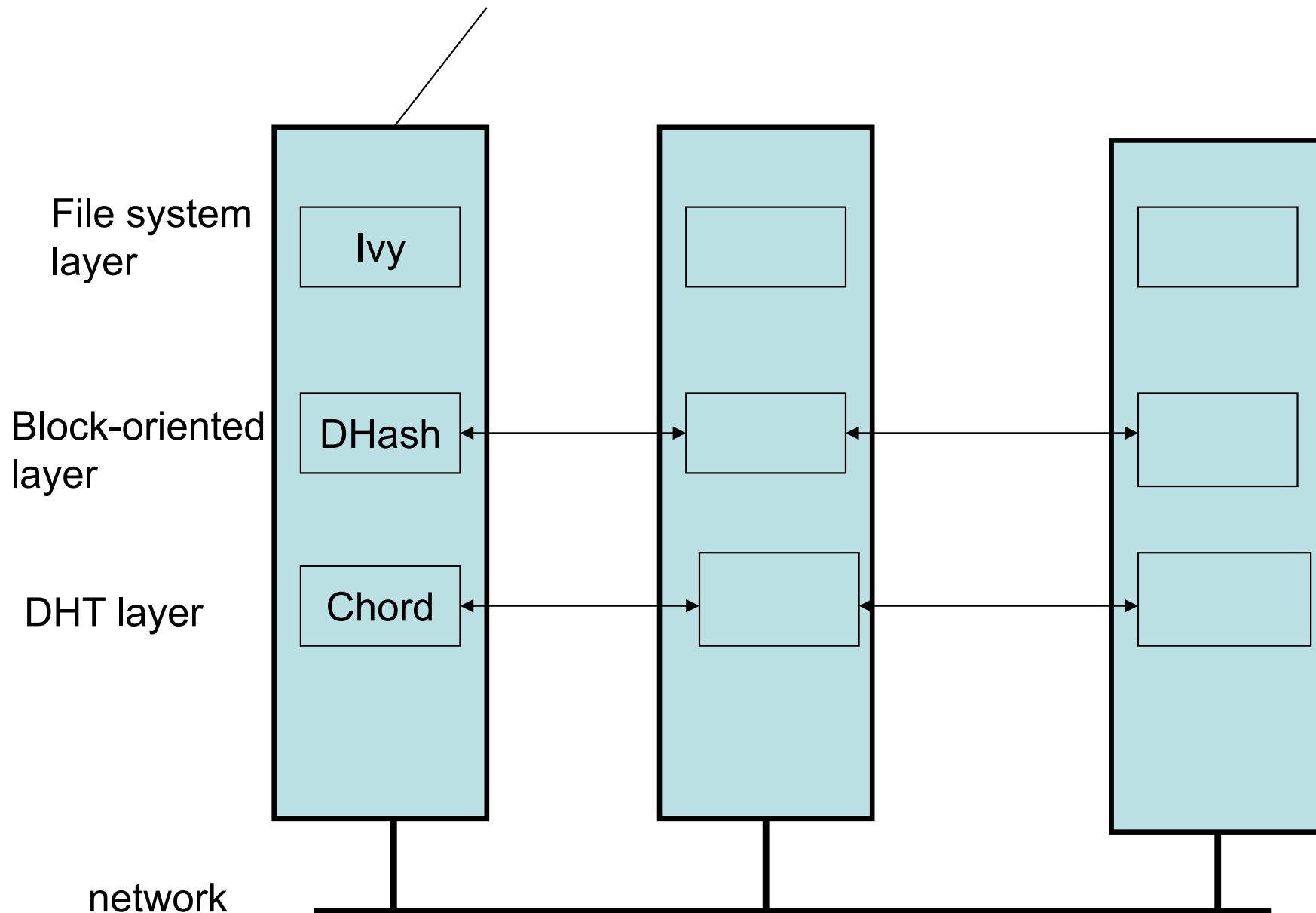
# Symmetric Architectures

- Fully distributed (decentralized) file systems do not distinguish between client machines and servers.
- Most proposed systems are based on a distributed hash table (DHT) approach for data distribution across nodes.
- The Ivy system is typical. It has a 3-layer structure.

# Ivy System Structure

- The DHT layer implements a Chord scheme for mapping keys (which represent objects to be stored) to nodes.
- The DHash layer is a block-storage layer
  - Blocks = logical file blocks
  - Different blocks are stored in different locations
- The top, or file system layer, implements an NFS-like file system.

Node where a file system is rooted



# Characteristics

- File data and meta-data stored as blocks in a DHash P2P system
- Blocks are distributed and replicated across multiple sites – increased availability.
- Ivy is a read-write file system. Writing introduces consistency issues (of data and meta-data)

# Characteristics

- Presents an NFS-like interface and semantics
- Performance: 2X-3X slower than NFS
- Potentially more available because of distributed nature



# DHash Layer

- Manages data blocks (of a file)
- Stored as content-hash block or public-key block
- Content-hash blocks
  - Compute the secure hash of this block to get the key
  - Clients must know the key to look up a block
  - When the block is returned to a client, compute its hash to verify that this is the correct (uncorrupted) block.

# DHash Layer – Public Key Blocks

- “A public key block requires the block’s key to be a public key, and the value to be signed using the private key.”
- Users can look up a block without the private key, but cannot change data unless they have the private key.
- Ivy layer verifies all the data DHash returns and is able to protect against malicious or corrupted data.

# DHash Layer

- The DHash layer replicates each file block B to the next k successors of the server that stores B.
  - (remember how Chord maps keys to nodes)
- This layer has no concept of files or file systems. It merely knows about blocks

# Ivy – the File System Layer

- A file is represented by a log of operations
- The log is a linked list of immutable (can't be changed) records.
  - Contains all of the additions made by a single user (to data and metadata)
- Each record records a file system operation (open, write, etc.) as a DHash content-hash block.
- A *log-head* node is a pointer to the most recent log entry

# Using Logs

- A user must consult all logs to read file data, (find records that represent writes) but makes changes only by adding records to its own log.
- Logs contain data and metadata
- Start scan with most recent entry
- Keep local snapshot of file to avoid having to scan entire logs

- Update: Each participant maintains a log of its changes to the file system
- Lookup: Each participant scans all logs
- The view-block has pointers to all log-heads

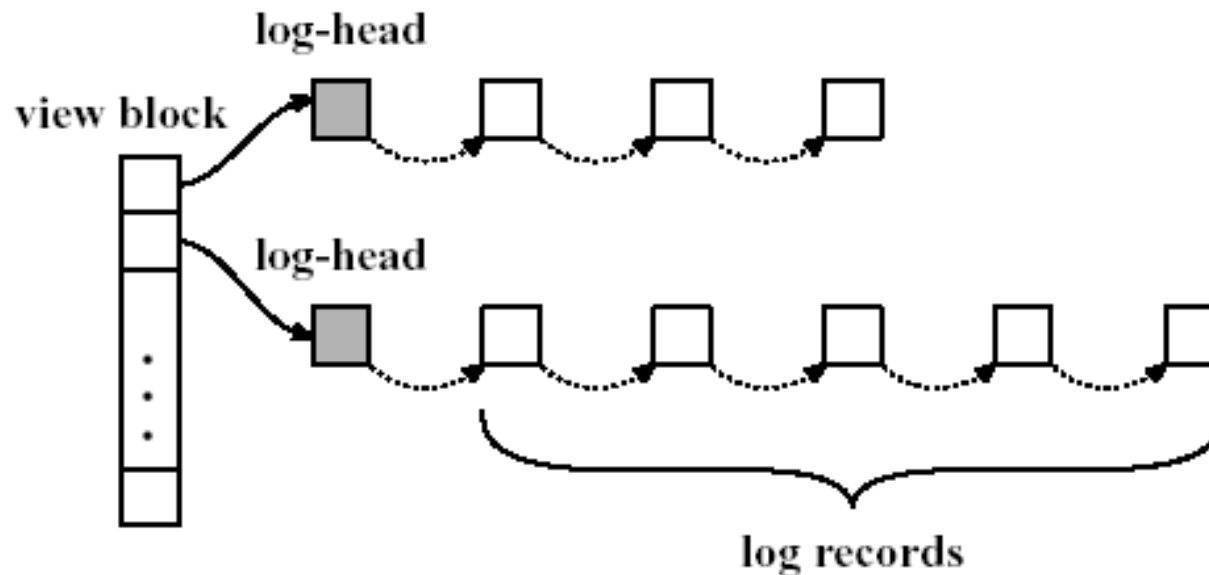


Figure 1: Example Ivy view and logs. White boxes are DHash content-hash blocks; gray boxes are public-key blocks.

# Combining Logs

- Block order should reflect causality
- All users should see same order
- For each new log record assign
  - A sequence # (orders blocks in a single log)
  - A tuple with an entry for each log showing the most recent info about that log (from current user's viewpoint)
    - Tuples are compared somewhat like vector timestamps; either  $u < v$  or  $v < u$  or  $v = u$  or no relation ( $v$  and  $u$  are concurrent)
    - Concurrency is the result of simultaneous updates