Cloud Storage 1

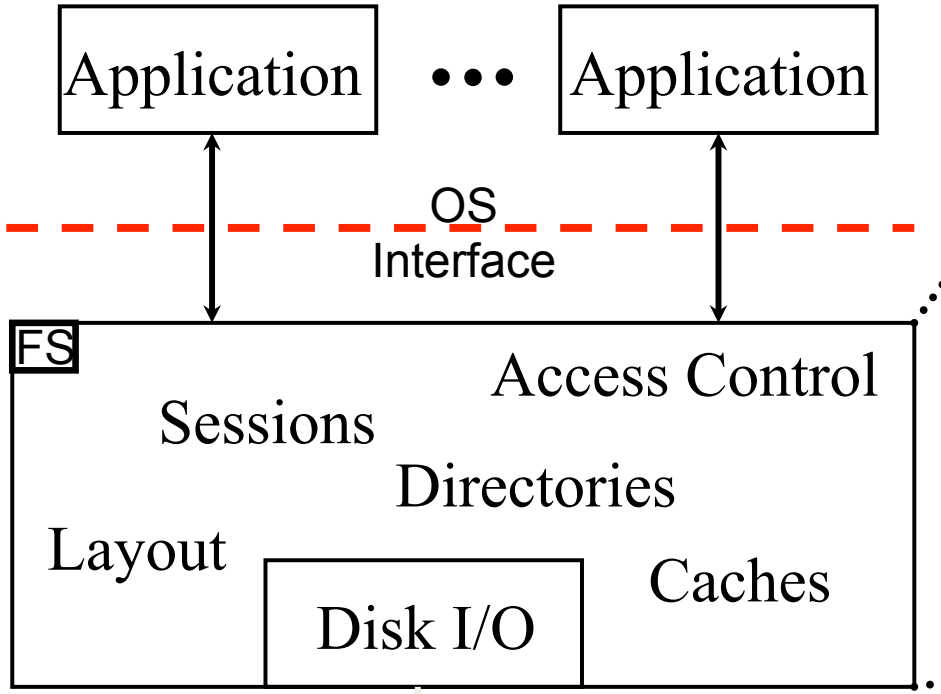
15-719 Advanced Cloud Computing

Garth Gibson
Greg Ganger
Majd Sakr

Cloud Scale Storage

- An essential service for scalable cloud systems is scalable storage
- One family of solutions extends the familiar distributed file system
  - Can NFS and/or AFS scale to cloud sizes?
  - NFS & AFS → NASD → GFS → HDFS
  - Shared file systems have always had a version of isolation (access control), and a version of sharing (global pathnames)
  - Today we review the technology leading to and in GFS-class cloud storage
- Another family abandons it (e.g S3) -- more on this next day

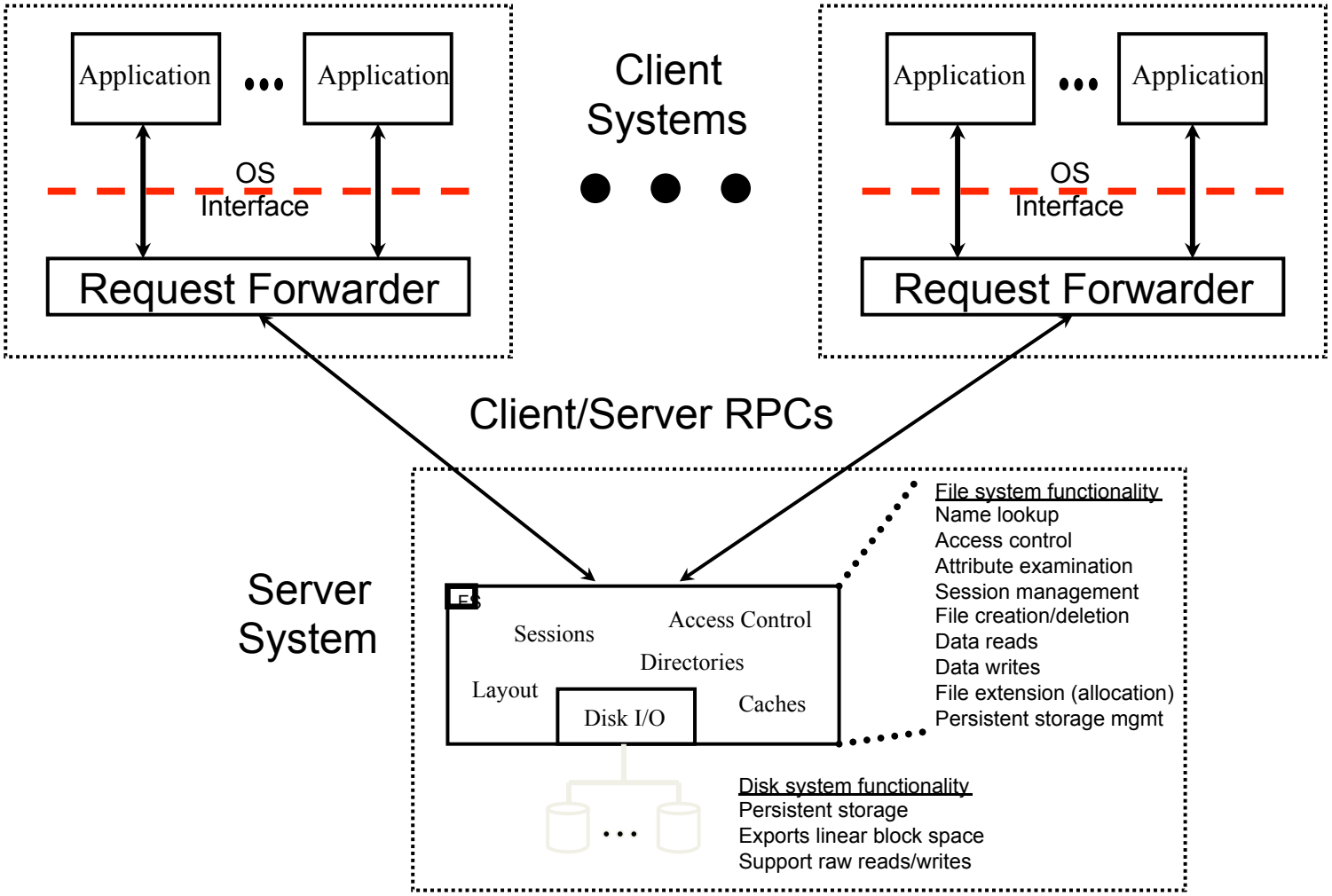
File system functionality



File system functionality
Name lookup
Access control
Attribute examination
Session management
File creation/deletion
Data reads
Data writes
File extension (allocation)
Persistent storage mgmt

Disk system functionality
Persistent storage
Exports linear block space
Support raw reads/writes

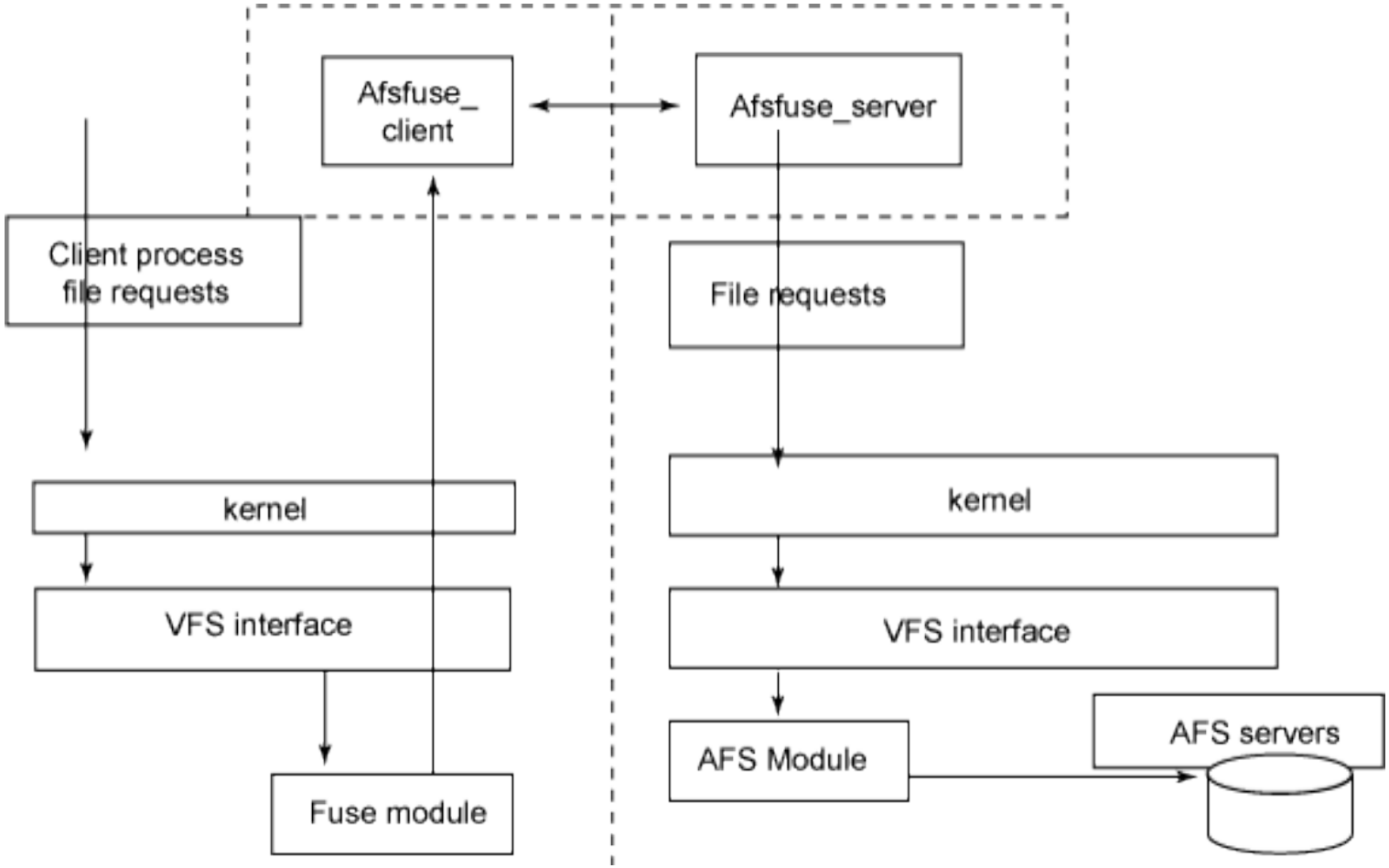
Approach #1: Server does everything



Pros/Cons of approach #1

- Pros: Simplicity
  - Server just looks same as kernel-based FS
    - application requests simply forwarded from clients
  - FS simplicity, security, etc… equivalent to standard case
    - with clients viewed as applications
    - Even system call atomicity can be preserved
- Cons: Performance
  - Performance of a server can bottleneck clients
    - this becomes a problem quickly as number of clients grows
  - Performance of network can bottleneck clients
    - even for a single client
  - Memory state must be maintained for each client app session

Example: AFS distributed filesystem w/ FUSE



Performance tuning: client-side caches

- Pretty much mandatory **in small-scale environments** (absent in GFS)
  - Spark (Tachyon -> Alluxio) adds (client) caching to a MR cluster
  - Special case: read-only …..
- Big new problem: cache coherence
  - having a consistent view across set of client caches
  - Preserving atomicity of unix system call layer (err, but they don't?)
- Different models of consistency
  - Unix (Sprite): all reads see most recent write
  - Original HTTP: all reads see a recent read
    - i.e., no consistency -- if you care, hit "reload"
  - NFS: other clients' writes visible in 30 seconds (open/close)
  - AFS (v2): file session semantics
    - reads see version closed most recently before calling open

Approach #2: Sprite (Unix)

- All reads see most recent write
- How?
  - Clients tell server when they open a file for read or write
    - also asks for latest timestamp if previously cached
  - Server tells client whether caching of the file is allowed
    - any # of readers or a single writer; any other combo denied to all
  - Server also calls back to clients to disable caching as needed
  - Client tells server whenever file is closed
- Who does what?
  - Client maintains most session info, like current offset
  - Server does almost all of FS activity
  - Client has cache, but so does server

Approach #3: NFS v3

- Other clients' writes visible within 30 seconds
  - Or the local open() after the other client's close()
- How?
  - Clients cache blocks and remember when last verified
  - Server promises nothing and remembers nothing
  - Clients check block freshness whenever older than 30 secs
  - No write-back caching is allowed on client or server (sort of)
  - Most implementations assume no concurrent write sharing
    - Test timestamps from last close on open to allow cache hits
- Who does what?
  - Client provides session and caching
  - Server does everything an FS would do on a local system
    - note that it replicates everything, including caching & access control
    - also note that other servers do this too, but also do other stuff

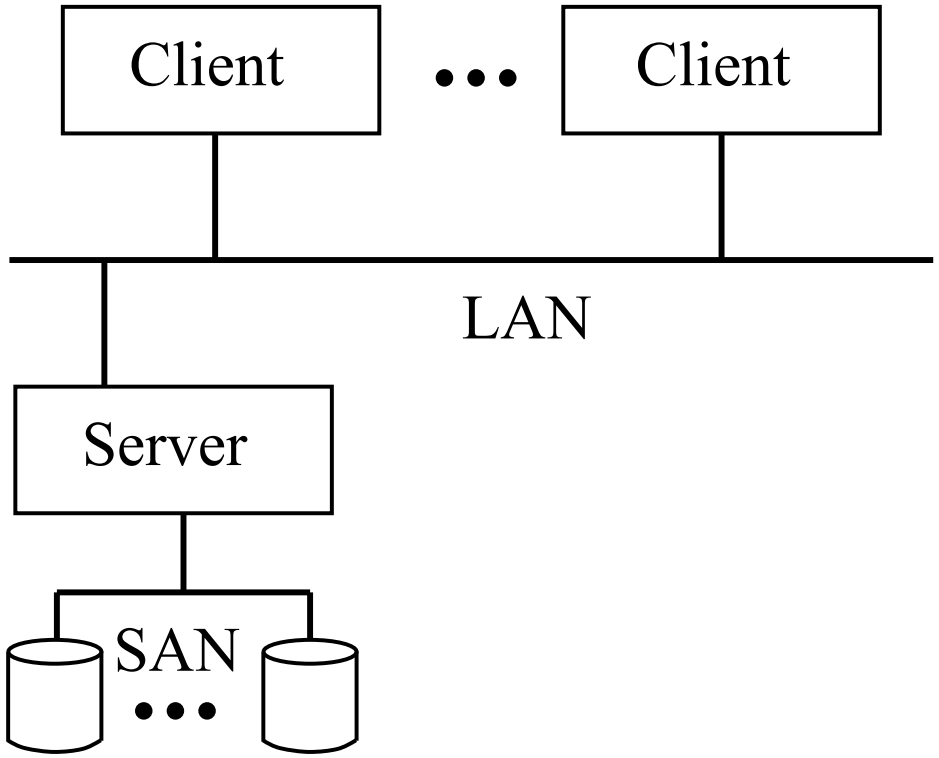
Approach #4: AFS (v2)

- All reads see opened version, which is most recent close
- How?
  - Clients obtain full copy of file and callback from server
    - only when opening a file that is not currently cached with callback
    - V3: no whole file semantics; use 64 KB chunks (huge file support)
  - Server promises to callback if the file changes
    - so client doesn't need to check again!
  - Client writes back entire file (if changed) upon close
    - also keeps cached copy, of course
  - Server revokes callbacks for the file from other clients
    - Simple race on close for concurrent access semantics
- Who does what?
  - Client provides session and complete caching
  - Server does callbacks, directory mgmt, and add'l caching

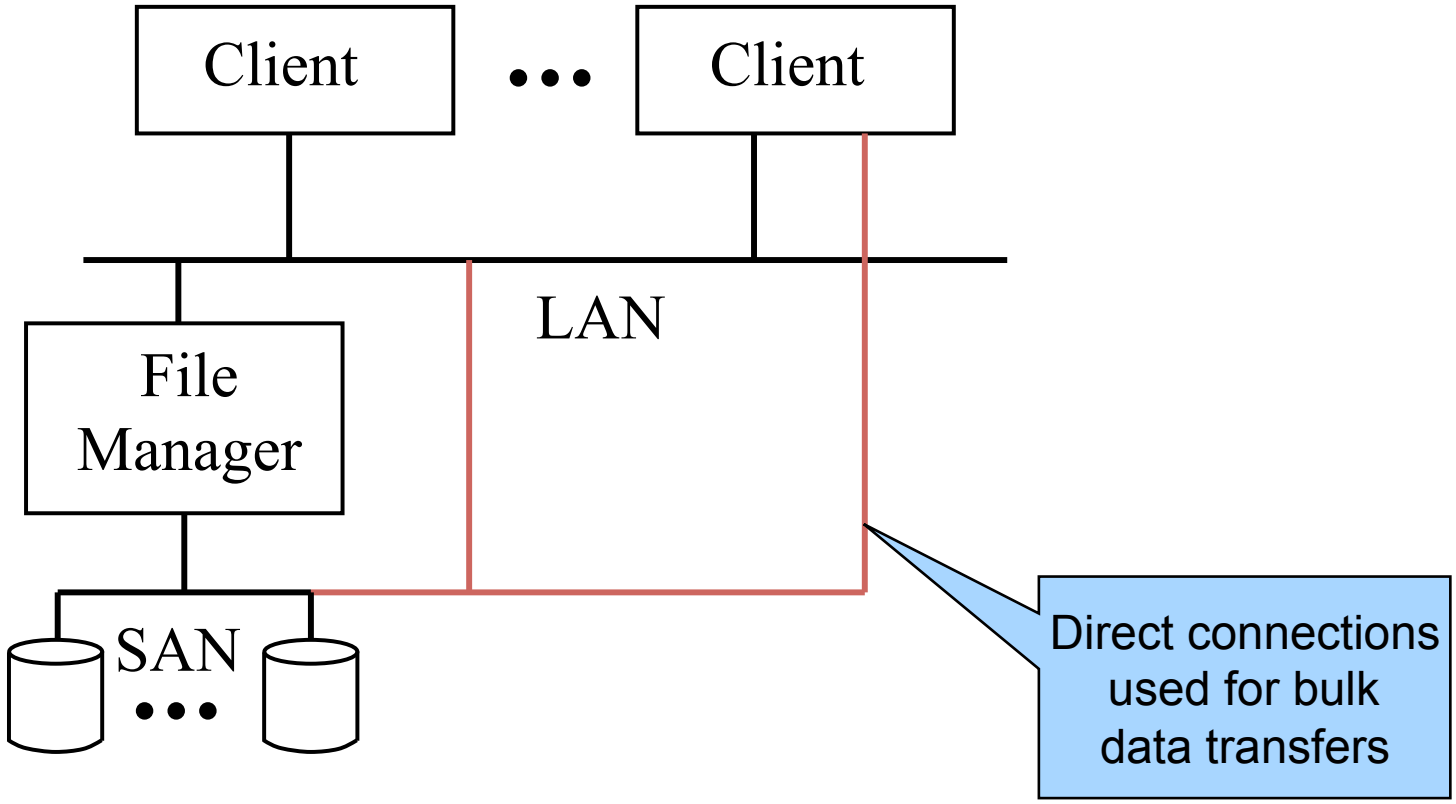
Should servers keep state about clients?

- Stateless servers
  - simple
  - may be faster (because of simplicity)
    - though usually not
  - quick/easy to recover from crashes
    - don't need to reconstruct client-related state
  - no problem with running out of state-tracking resources
- Stateful servers
  - may be faster (because of long-lived connections)
    - can transform workload to something "better" for server
    - can keep clients from repeatedly asking "still up-to-date?"
  - can provide better semantics for users/programmers
  - Spark driver is stateful for a while ….

Client/Server Architecture

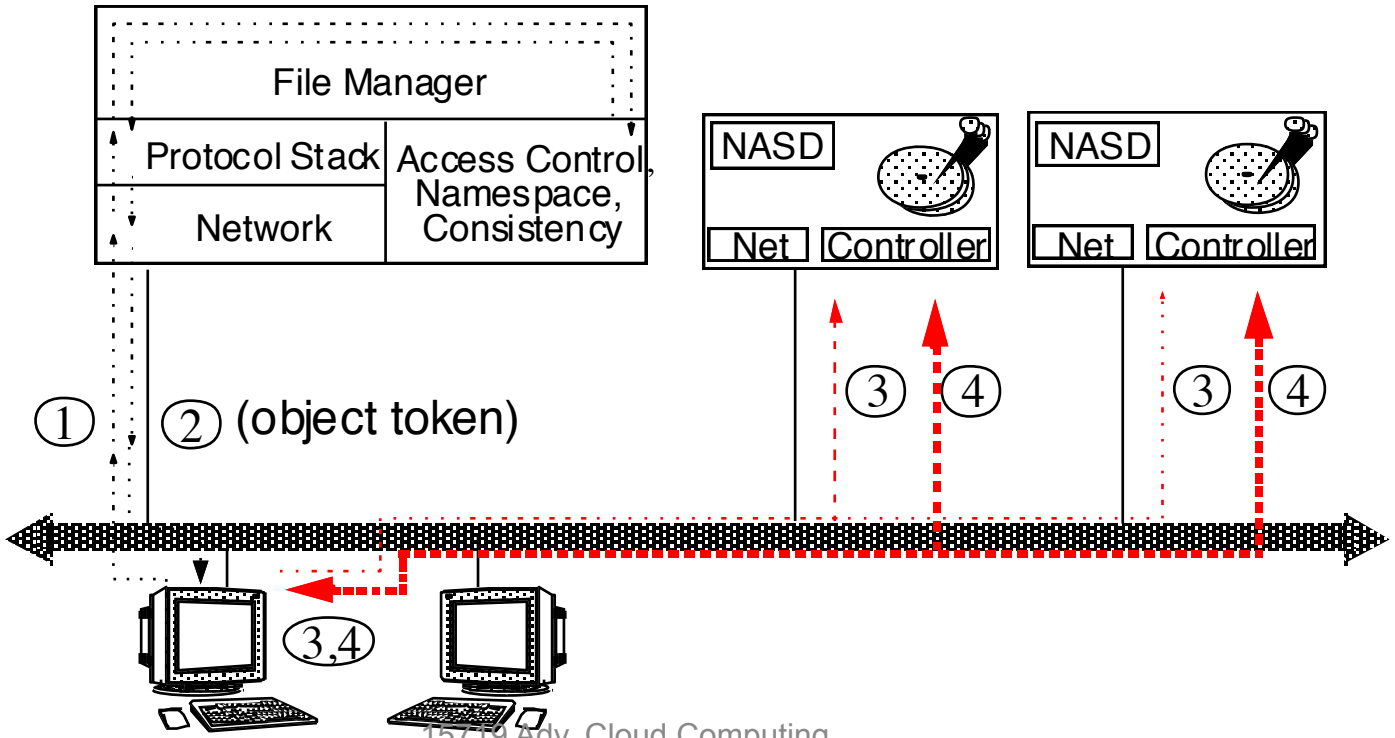


Example: NASD, EMC HighRoad, pNFS



Network Attached Secure Disk (NASD)

- File/object storage management in storage device (inode-like)
  - Request (1, 2) & cache rights to read/write/extend objects on disks (3, 4)
    E.g. CMU NASD, Pansas, Lustre, Google File System, HDFS
- But, changes in storage device standards blocked “in the disk” solutions

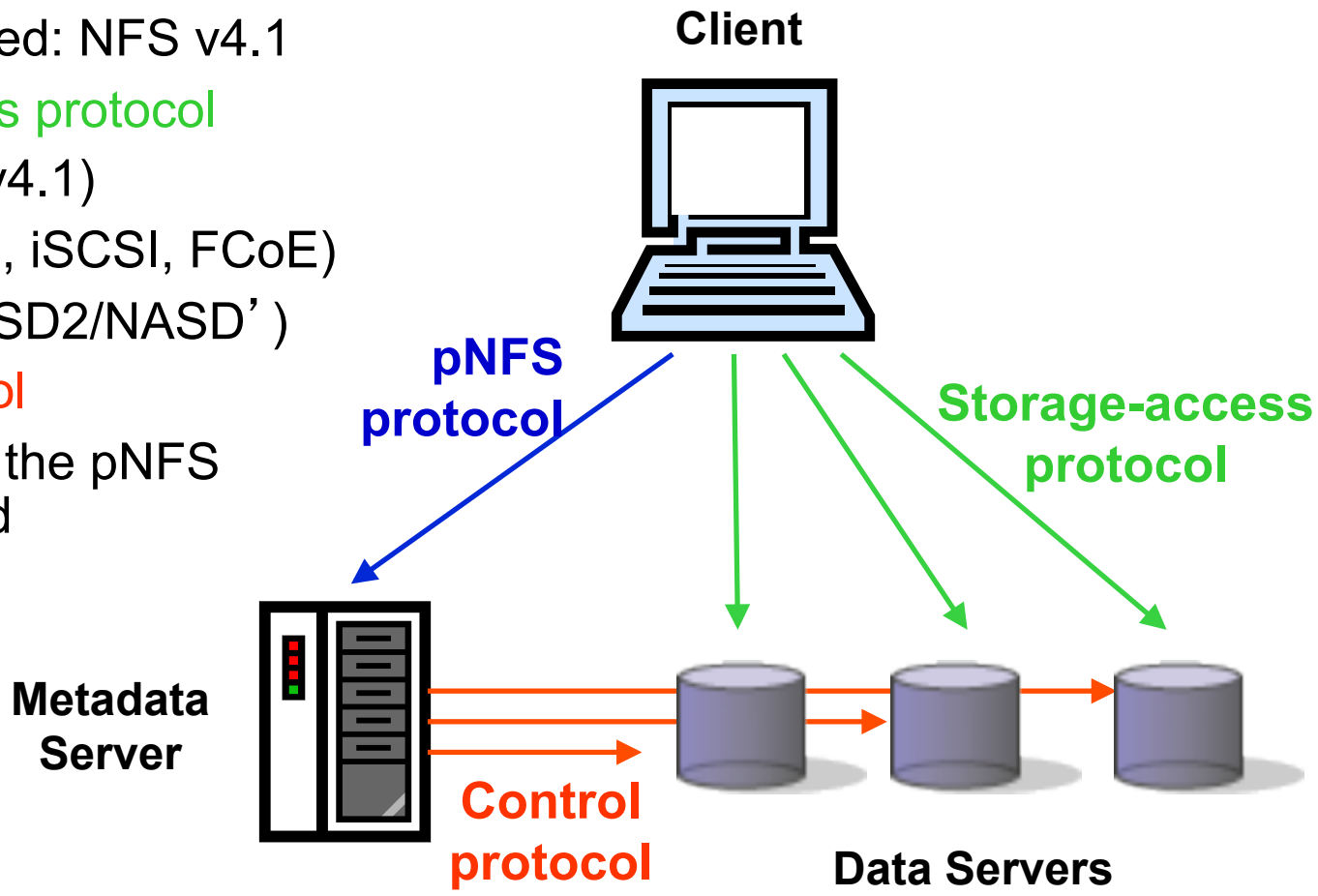


Example: NASD standardized as pNFS

pNFS protocol
- standardized: NFS v4.1

Storage-access protocol
- files (NFSv4.1)
- blocks (FC, iSCSI, FCoE)
- objects (OSD2/NASD')

Control protocol
- Outside of the pNFS standard



The Google File System.

Ghemawat03: S. Ghemawat, H. Gobioff, S-T. Leung, Symp. on Operating Systems Principles (SOSP'03), Oct 2003, Bolton Landing, NY.

Scaling Distributed Storage

- Failures become the norm (many cheap nodes)
  - Spend disk capacity for simple solution
  - Design schemes to be less impacted by failure
  - Weaken promises to applications
- Having a captive application set a big plus
  - Limit number of files
  - Rule out small files performance as a goal
  - Allow some access patterns to perform poorly
  - Change file system semantics (weaken traditional guarantees, add new special ops)

Google FS Arch

- Should look a lot like NASD cuz it is
  - Applications more homogeneous (then at least)

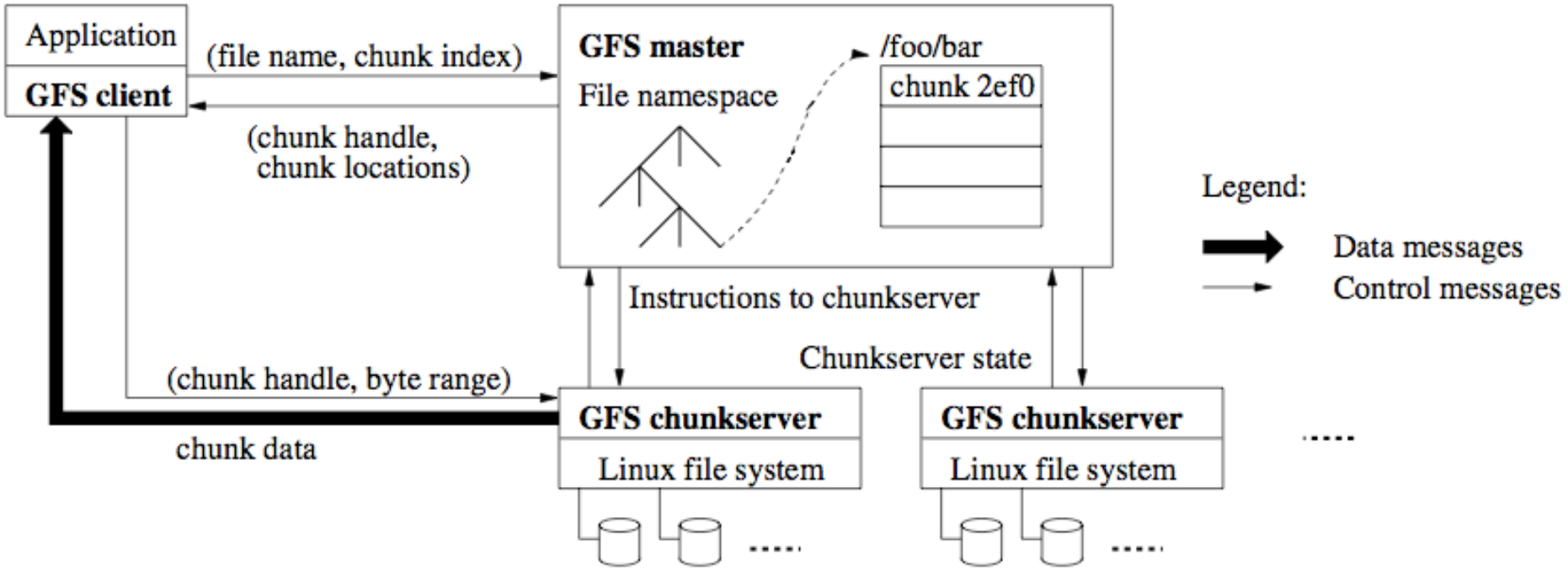


Figure 1: GFS Architecture

Simplify !

- Single master w/ all metadata in memory
  - all files much be large; one block size (64MB)
  - rebuild state often, to allow non-durable state
    - locations of chunks not durable; poll on boot
- Concentrate all synchronization in new op
  - Don’t hide inconsistency from application
- User-level library (no legacy support)
  - UNIX-like API (but no links, special fast copy)
- Repair/recovery is async, so over-replicate
- No data caching anywhere but under object server (in ext3)

Dual Ordering

- Chunk transfers

  & ordered different
  - one copy of data

    elected as leader

    (output ordering)
  - orders writes AFTER

    data has arrived at all
    - Transfer pipeline by distance in IP space
    - Design for poor cluster network bisection BW

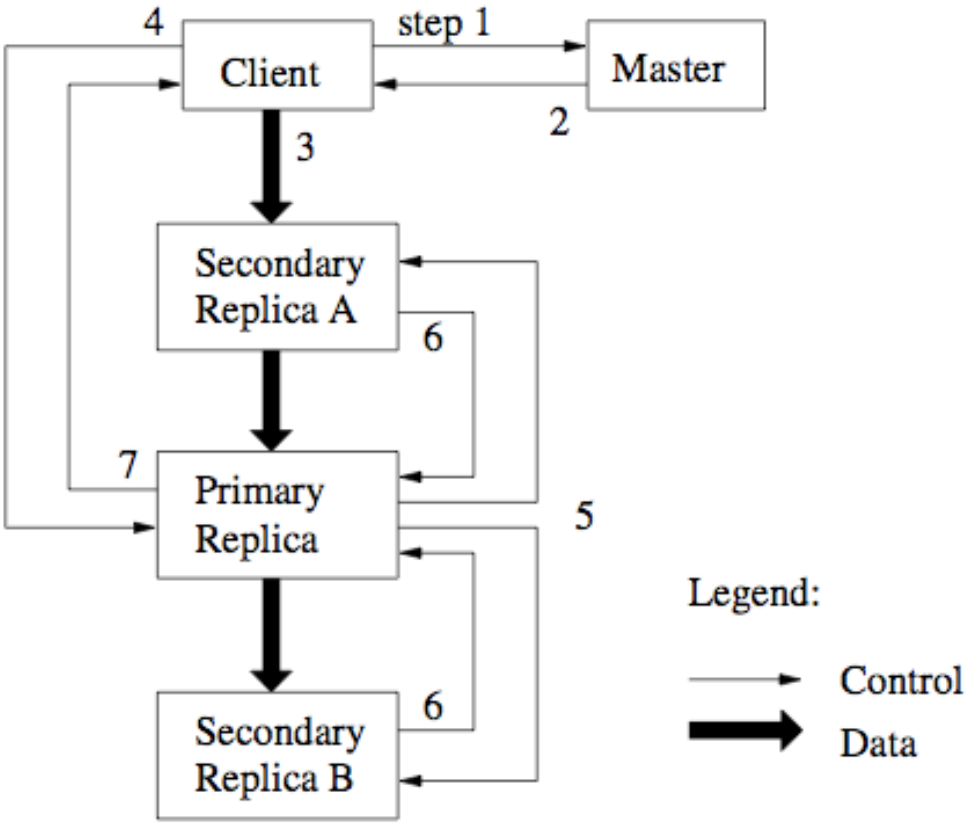


Figure 2: Write Control and Data Flow

Concurrent sharing

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

- Concurrent updates can be "undefined"
  - Large transfer split into separate races (like NFS)
- Append as special "thin" synch'g solution
  - GFS picks and returns append offset (<1/4 chunk)
  - record appended atomically AT LEAST ONCE
    - Racing failures can leave differences in copies
    - GFS may pad or duplicate records
- Apps SHOULD validate (don't trust GFS)
  - formating, checkpointing, checksums, sequence numbers (nonces)

Recovery

- Background visit to each chunk
  - if parts are missing, re-replicate
    - Reconstructed 600 GB in 23 mins (DDN HW RAID 72+ hrs)
  - throttled to increase “availability”
  - gradual slow rebalancing
  - remove “stale” chunks w/ old version numbers
- Delete is just unlink
  - chunk is garbage collected
  - until collected, it is still available (time travel)

Single Metadata Server

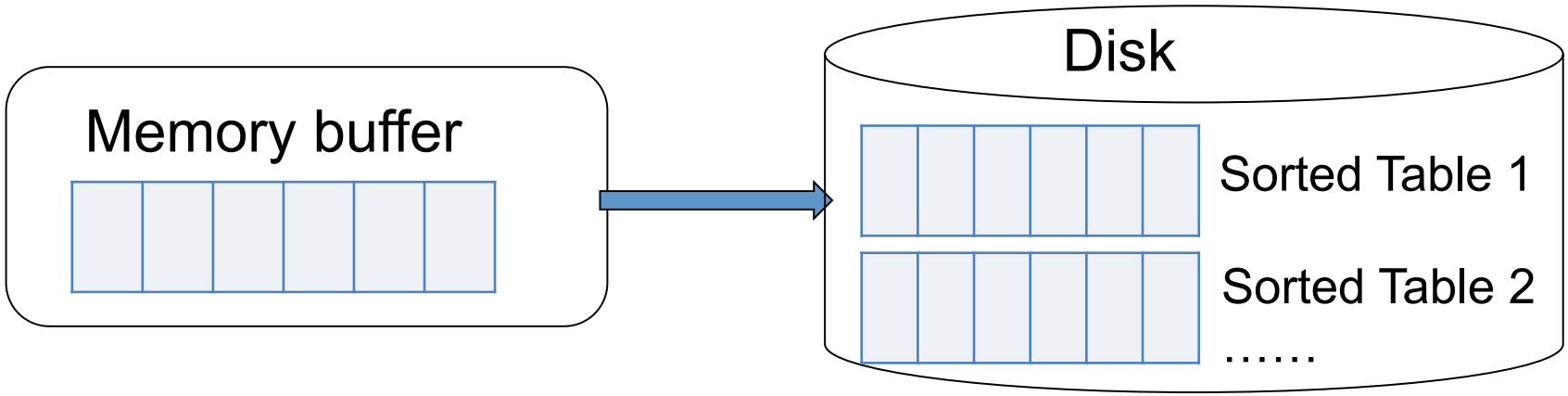
- 1:1,000,000 metadata to data size ratio
  - force all large files (highly restrictive as FS)
  - 4 GB metadata (in memory) maps 1 PB data, millions of files
- prefix table of pathnames, not directory tree
  - faster, but data management tools all break
- not single threaded (not that simple)
- replicate log & checkpoints
  - shadows can support readonly access

Hints of GFS2: Colossus

- GFS was limited to 50M files, 10 PB
  - Users generate too many small files
  - Large file bias induced extra complexity on apps
  - Low latency apps poorly supported
- No longer a single metadata server
  - “shards” metadata over many servers
  - Uses BigTable to store this metadata
- No longer simply replication
  - Uses Reed-Solomon (RAID 6 like) encoding
  - Computes encoding at client (like Panasas)
    - Client sends all “copies” rather than chaining copying

Log Structured Merge (LSM) Trees

- Insert / Updates
  - Buffer and sort recent inserts/updates in memory
  - Write-out sorted buffers into local file system sequentially
  - Less random disk writes than traditional B-Tree
- Lookup / Scan
  - Search sorted tables one by one from the disk
    - Compaction is merge sort into new files, deleting old (cleaning)
  - Bloom-filter and in-memory index to reduce lookups

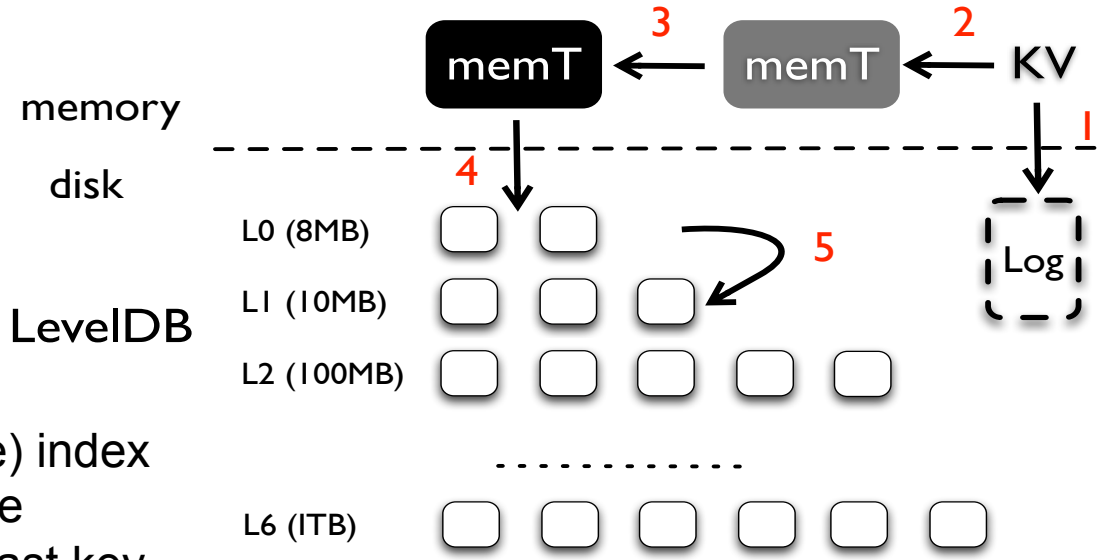


Write Optimized like LFS (cleaning = compaction)

LSM-trees: Insertion

Clean so there is no overlap in SSTables in each level after 0

1. Write sequentially 2. Sort data for quick lookups
3. Sorting and garbage collection are coupled



- (Cacheable) index per SSTable
- Lists 1st & last key per SSTable

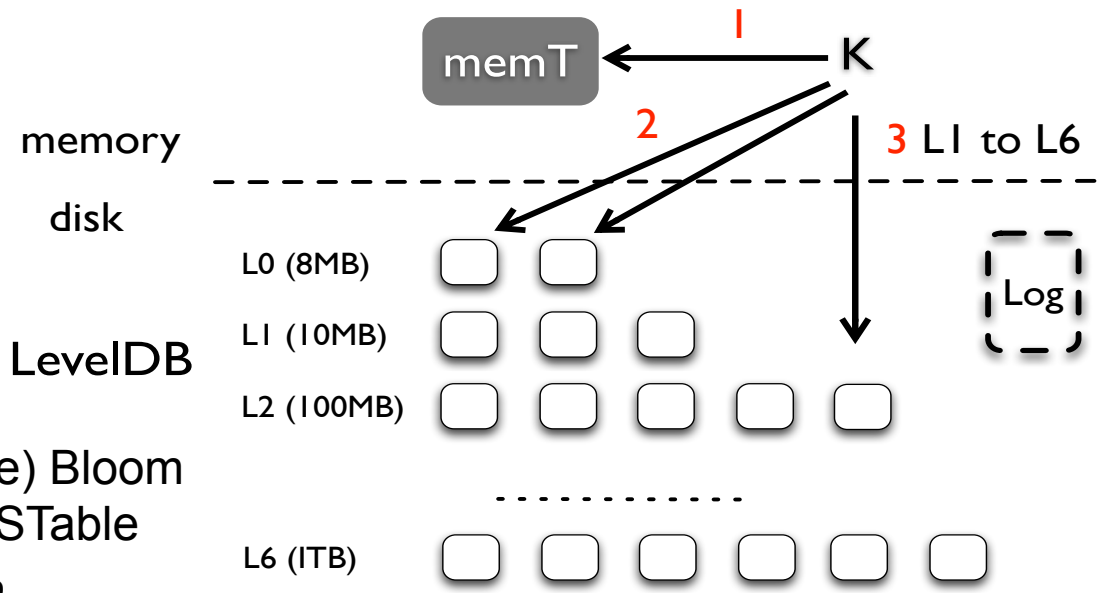
[Lanyue Lu, FAST16]

O(log size) lookup like B-tree

Clean so there is no overlap in SSTables in each level after 0

LSM-trees: Lookup

1. Random reads
2. Travel many levels for a large LSM-tree

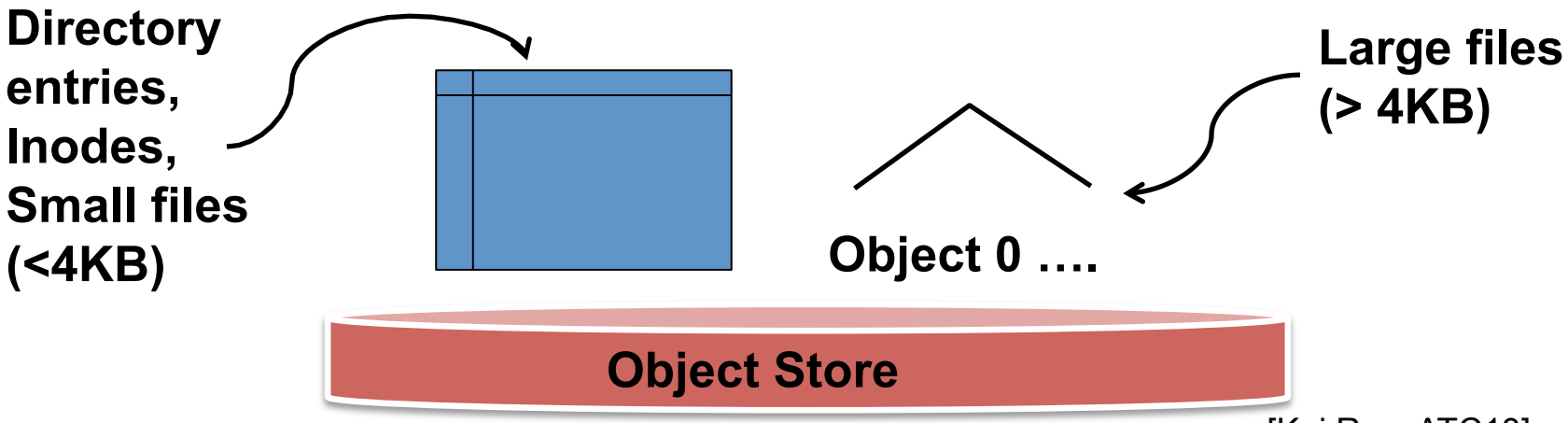


- (Cacheable) Bloom filter per SSTable
- Skip ~99% unneeded lookups

[Lanyue Lu, FAST16]

TableFS packs metadata into LSM Trees

- Small objects (<= 4KB) are embedded into LSM (Log-structure Merge) tree (a tabular structure)
  - E.g. directory entries, inodes, small files
  - Turn many small files into one large object (~ 2MB)
- Larger files (> 4KB) are stored directly in object store indexed by TableFS assigned ID number

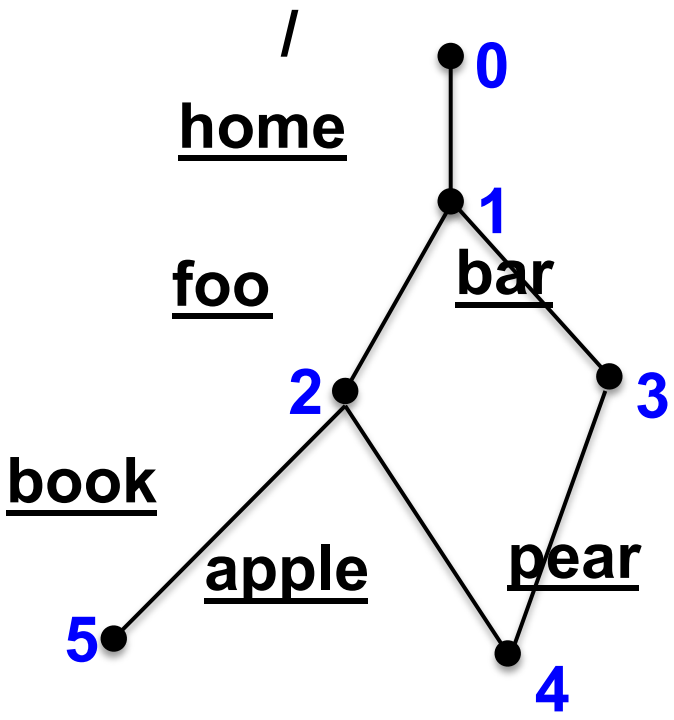


[Kai Ren, ATC13]

Table Schema

- Key: <Parent inode number, hash(filename)>
  - Inodes with multiple hard links: <inode number, null>
- Value: filename, inode attributes, inlined file data (or symbolic link to large object).

[Kai Ren, ATC13]



Lexicographic order

Key	Value
<0,hash(*home*)>	1, “home”, struct *stat*
<1,hash(*foo*)>	2, “foo”, struct *stat*
<1,hash(*bar*)>	3, “bar”, struct *stat*
<2,hash(*apple*)>	4, “apple”, hard link
<2,hash(*book*)>	5, “book”, struct *stat,* inline small file
<3,hash(*pear*)>	4, “pear”, hard link
<4,null>	4, struct *stat*, large file pointer

Table Schema (cont')

- Advantages:
  - Reduce random lookups by co-locating directory entries with inode attributes, and small files
  - "*readdir*" performs sequential scan on the table

Entries in the same directory

Key	Value
<0,hash(*home*)>	1, "home", struct *stat*
<1,hash(*foo*)>	2, "foo", struct *stat*
<1,hash(*bar*)>	3, "bar", struct *stat*
<2,hash(*apple*)>	4, "apple", hard link
<2,hash(*book*)>	5, "book", struct *stat,* inline small file
<3,hash(*pear*)>	4, "pear", hard link
<4,null>	4, struct *stat*, large file pointer

[Kai Ren, ATC13]

Next day plan

- More cloud storage

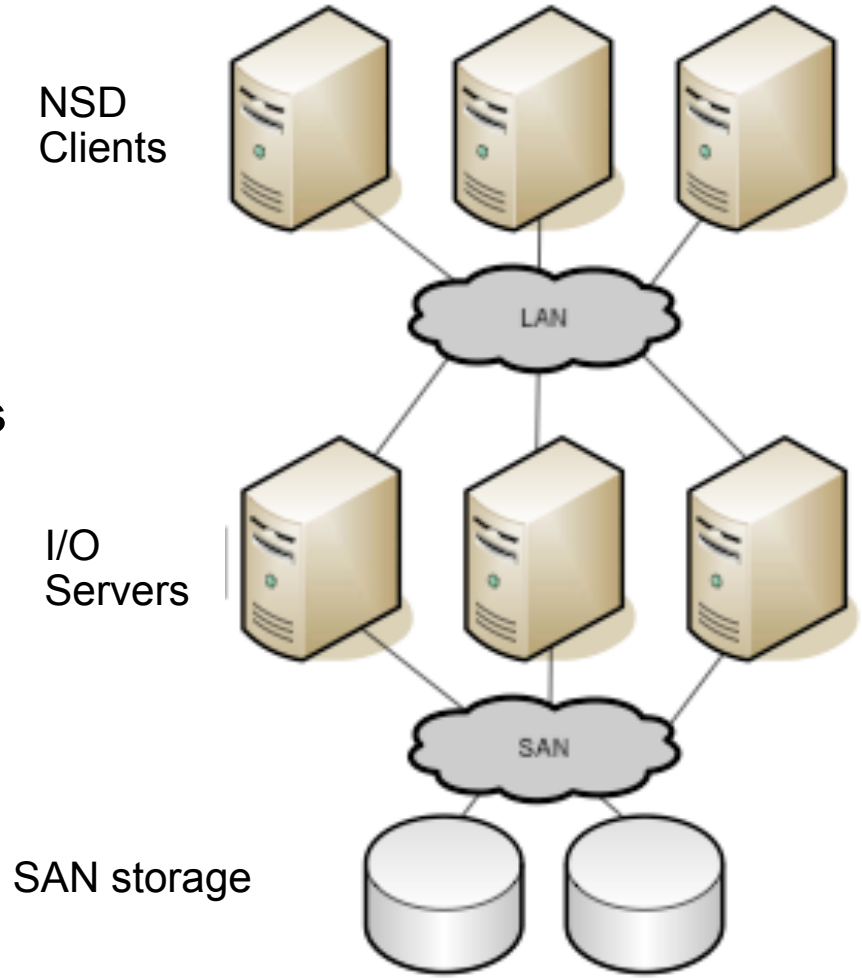
Production Parallel File Systems

- All four systems scale to support very large compute clusters
  - LLNL Purple, LANL RoadRunner, Oakridge Jaguar/Titan, etc.
- All but GPFS delegate block management to “object-like” data servers or OSDs
- Approaches to metadata vary
- Approaches to fault tolerance vary
- Emphasis on features, “turn-key” deployment, vary

GPFS

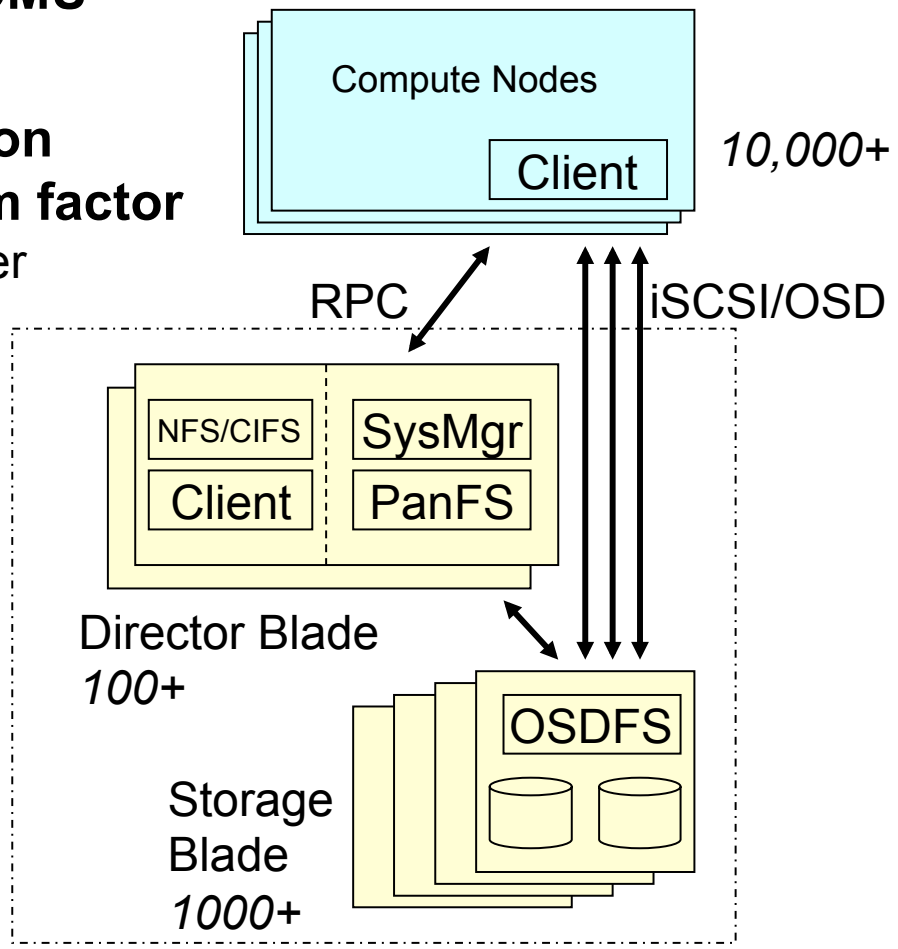
GPFS

- **IBM product**
- **General Parallel File System**
- **Legacy: multimedia filesystem**
- **Block interface to storage nodes**
  - Shared memory abstraction
  - Symmetric I/O servers can all "master" all/any data
- **Distributed locking (DLM)**
  - Needed to access metadata describing block storage



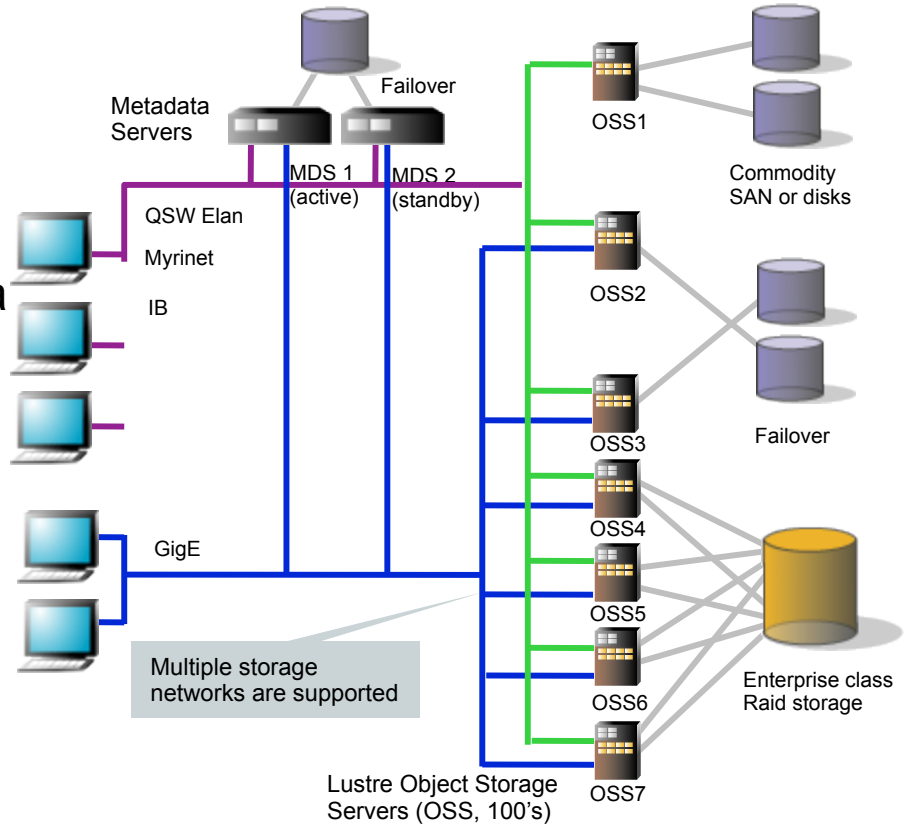
PANFS

- **Panasas product based on CMU NASD research**
- **Complete "appliance" solution (HW + SW), blade server form factor**
  - DirectorBlade = metadata server
  - StorageBlade = OSD
- **Coarse grained metadata clustering**
  - DirectorBlades manipulate metadata distributed in objects
- **Linux native client for parallel I/O**
  - Fast & scalable but complex



LUSTRE

- **Dominant Supercomputer FS today**
- **Open source object-based parallel file system**
  - Based on CMU NASD architecture
  - Lots of file system ideas from Coda
  - ClusterFS acquired by Sun,
  - Sun acquired by Oracle
  - Intel acquired Whamcloud team
  - Today: OpenSFS Foundation
- **Asymmetric design with separate metadata server**
  - Metadata is bottleneck (see GFS)
- **Distributed locking with client-driven lock recovery**
  - For write-back caching of metadata changes



Lustre material from www.lustre.org and various talks