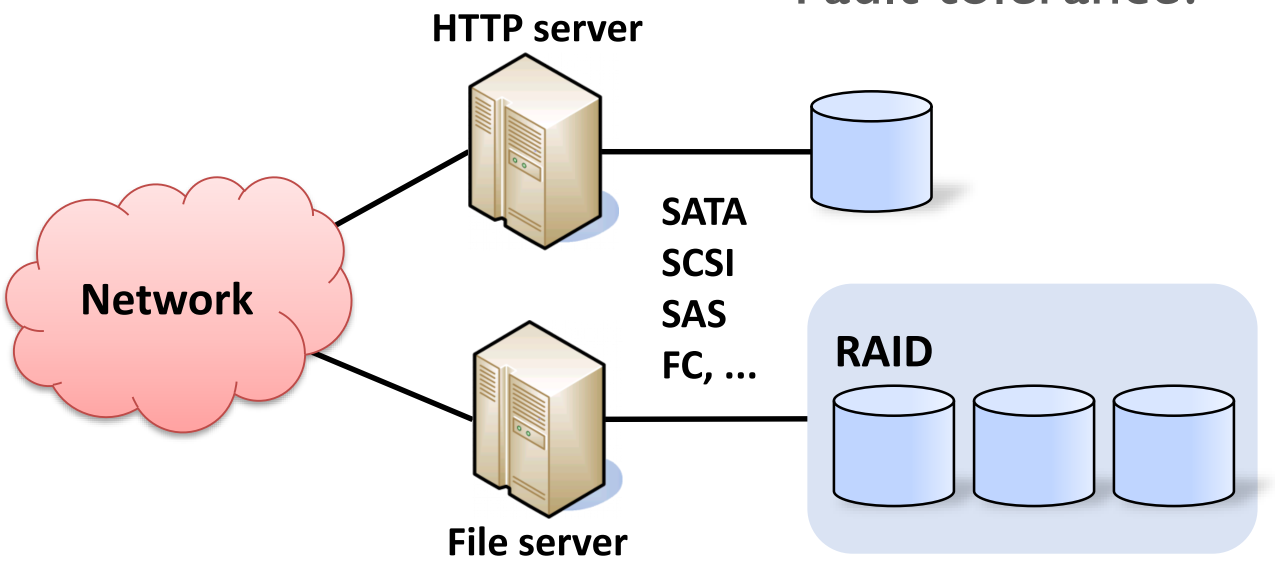
Distributed File Systems

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)
Systems Software &
Architecture Lab.
Seoul National University

Spring 2019

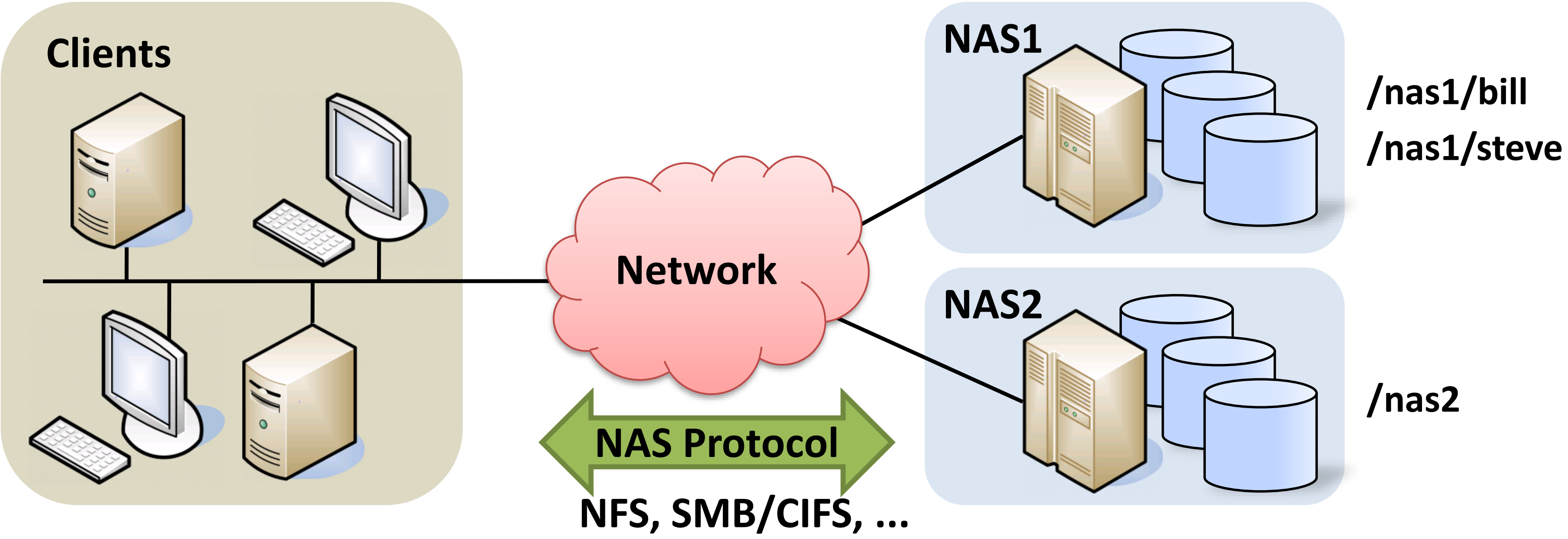
DAS

- Direct-Attached Storage
  - Simple to deploy
  - Lower initial cost
  - Sharing data?
  - Load balancing?
  - Scalability?
  - Fault tolerance?



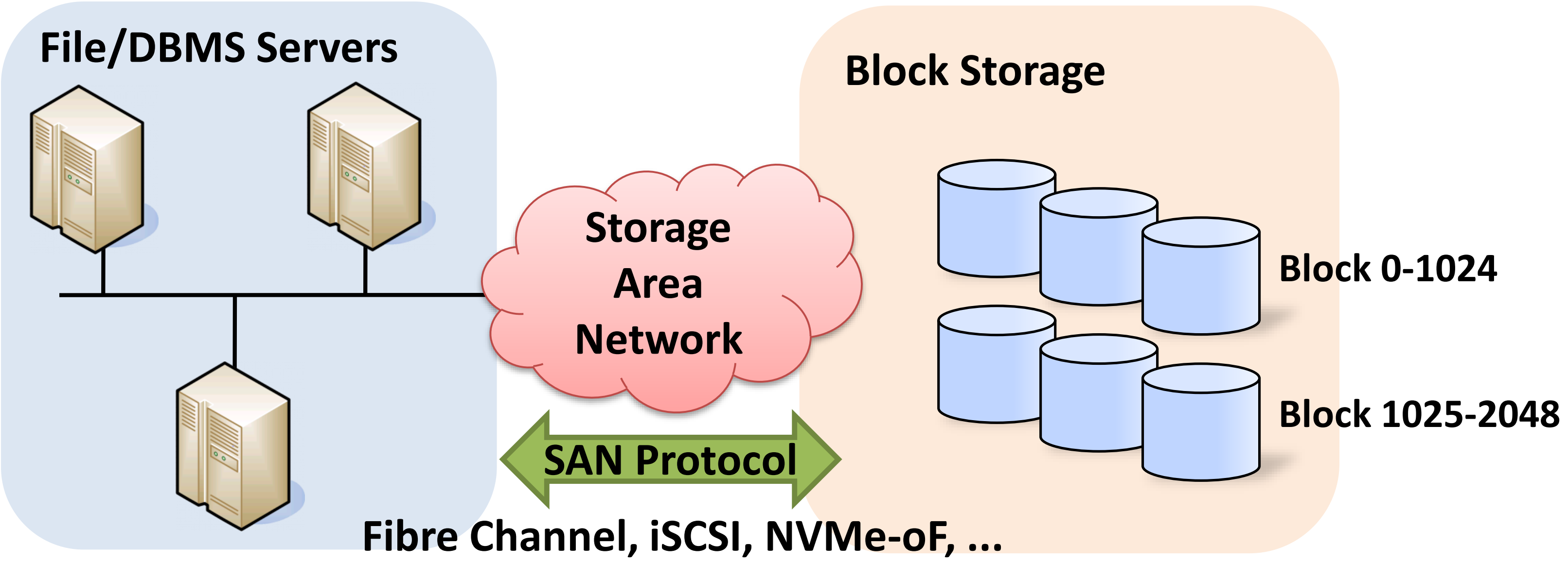
NAS

- Network-Attached Storage
  - File-level data sharing
  - Easy to install & deploy
  - Heterogeneous systems support
  - Static data partitioning
  - Scalability?
  - Automatic load balancing?
  - Transparent migration?



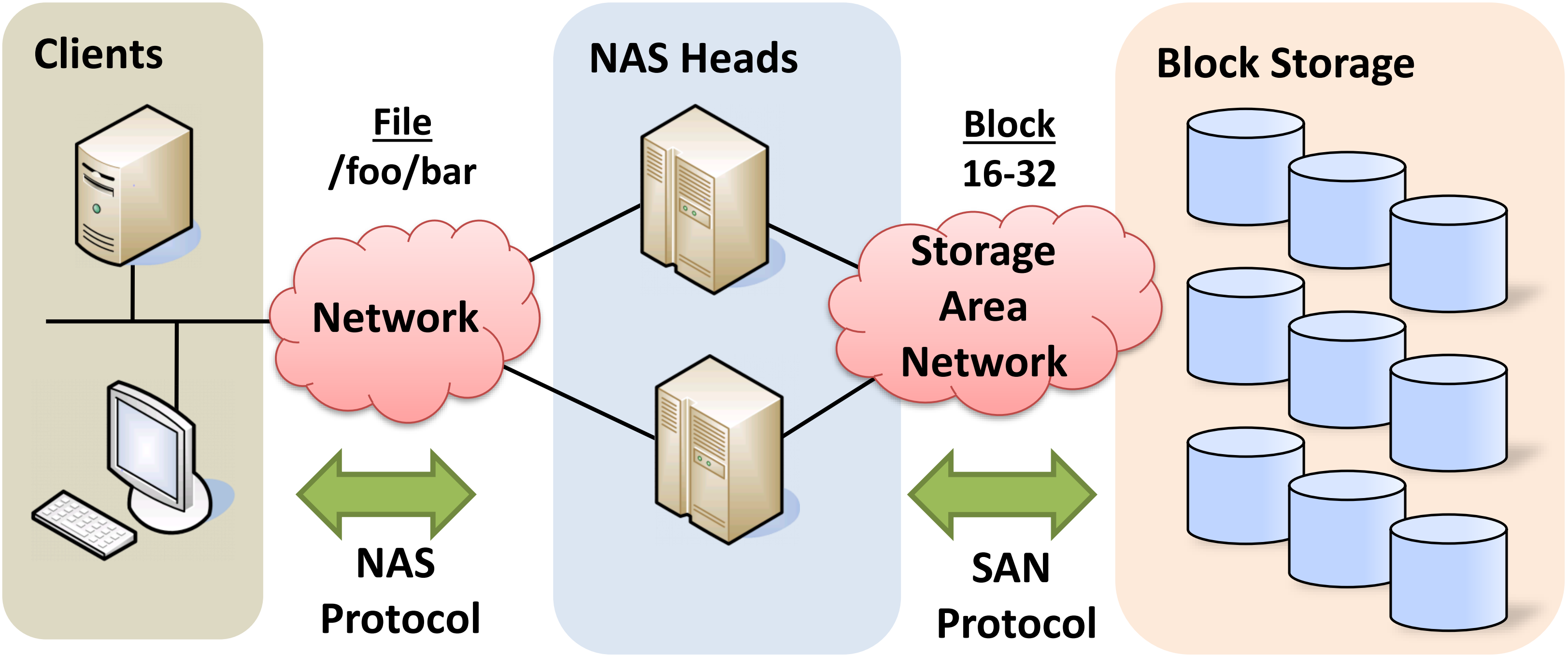
SAN

- Storage Area Network
  - Block-level data sharing
  - High performance
  - High availability
  - Sharing files?
  - Cost?
  - Management complexity?
  - Interoperability?



NAS/SAN Convergence

- NAS Head
  - A NAS with no on-board storage (connected to a SAN)
  - File system operations → Block device operations
  - Cache file contents



Andrew File System (AFS)

AFS

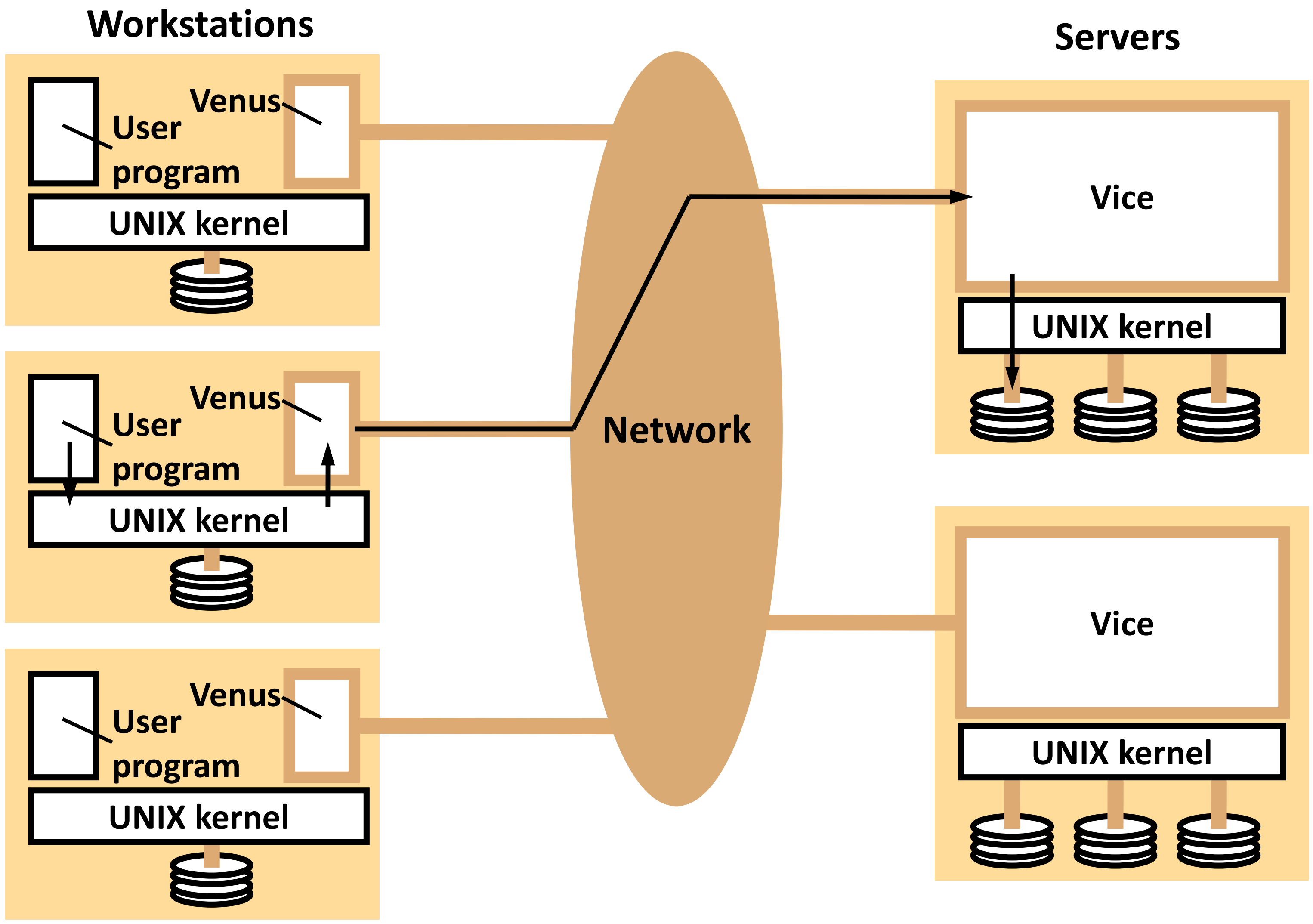
- A distributed filesystem for Andrew, a distributed computing environment developed at CMU (1983~)
- Transarc Corp. founded to commercialize AFS (1989)
- Transarc becomes subsidiary of IBM (1998)
- IBM releases OpenAFS as open source (http://www.openafs.org)

- Originally for campus computing network with up to at least 7000 workstations
- In 1991, approximately 800 workstations are serviced by ~ 40 AFS servers at CMU

AFS Design Goals

- Transparent access to remote shared files for UNIX programs
  - Compatibility with UNIX at the system call level
  - No modification or recompilation of UNIX programs
- Common namespace from all workstations
- Client-server model
- Scalability
- User-level implementation

AFS Architecture



Source: G. Coulouris et al., Distributed Systems Concepts and Design, 5th Ed., 2012

AFS Characteristics

- Volume-based for easy of location and movement
  - A partial subtree of the shared namespace
  - Volume Location Database on each server
- Whole-file serving
- Whole-file caching
- AFS session semantics (or close-to-open semantics)
  - Once a file is closed, the changes made to it are visible to new opens anywhere on the network
  - No communication during reads/writes
- Callbacks to maintain cache coherency
- Replication of read-only volumes

Distributed File Systems: Design Issues

- POSIX compliance
- Metadata management (File metadata, namespace, location)
- Consistency models
- Maintaining consistency against component failures (disk, server, network)
- Presence of SPOF (single point of failure)
- Scalability
- Automatic load balancing
- Compression, Deduplication, Encryption, …
- Online node addition/removal
- Ease-of-maintenance

The Google File System

(S. Ghemawat et al., SOSP, 2003)

Some of slides are borrowed from the authors' presentation.

The Joys of Real Hardware

- Typical first year for a new cluster:
  - ~0.5 overheating (power down most machines in < 5 mins, ~1-2 days to recover)
  - ~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
  - ~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
  - ~1 network rewiring (rolling ~5% of machines down over 2-day span)
  - ~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
  - ~5 racks go wonky (40-80 machines see 50% packetloss)
  - ~8 network maintenances (4 might cause ~30-minute random connectivity losses)
  - ~12 router reloads (takes out DNS and external vips for a couple minutes)
  - ~3 router failures (have to immediately pull traffic for an hour)
  - ~dozens of minor 30-second blips for dns
  - ~1000 individual machine failures
  - ~thousands of hard drive failures
  - slow disks, bad memory, misconfigured machines, flaky machines, etc.
  - Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Source: J. Dean, "Designs, Lessons and Advice from Building Large Distributed Systems," LADIS Keynote, 2009.

Introduction

- Motivation
  - Manipulate large (TBs) sets of data
  - Large numbers of machines with a modest amount (~ 1TB) of storage
  - Component failures are the norm

- Goal
  - Scalable, high performance, fault tolerant distributed filesystem
  - RAIS (Redundant Array of Inexpensive Servers) ☺

Why Build Rather Than Buy/Steal?

- Many existing filesystems: AFS, xFS, InterMezzo, Lustre, GPFS, NFS, Swift, etc.
- We build on the lessons of the past

- None designed for our failure model
- Few scale as highly or dynamically
- Lack special primitives for large distributed computation

Observations

- Component failures are the norm rather than the exception
  - The system is built from many inexpensive commodity components
  - Requires constant monitoring, error detection, fault tolerance, and autonomic recovery
- Files are huge by traditional standards: Multi-GB files are common
- Applications use a few specific access patterns
  - Large streaming reads / small random reads
  - Append to large files
- Co-designing the applications and the filesystem API increase the flexibility
  - APIs similar to POSIX + atomic append, snapshot
- High sustained bandwidth is more important than low latency

Components

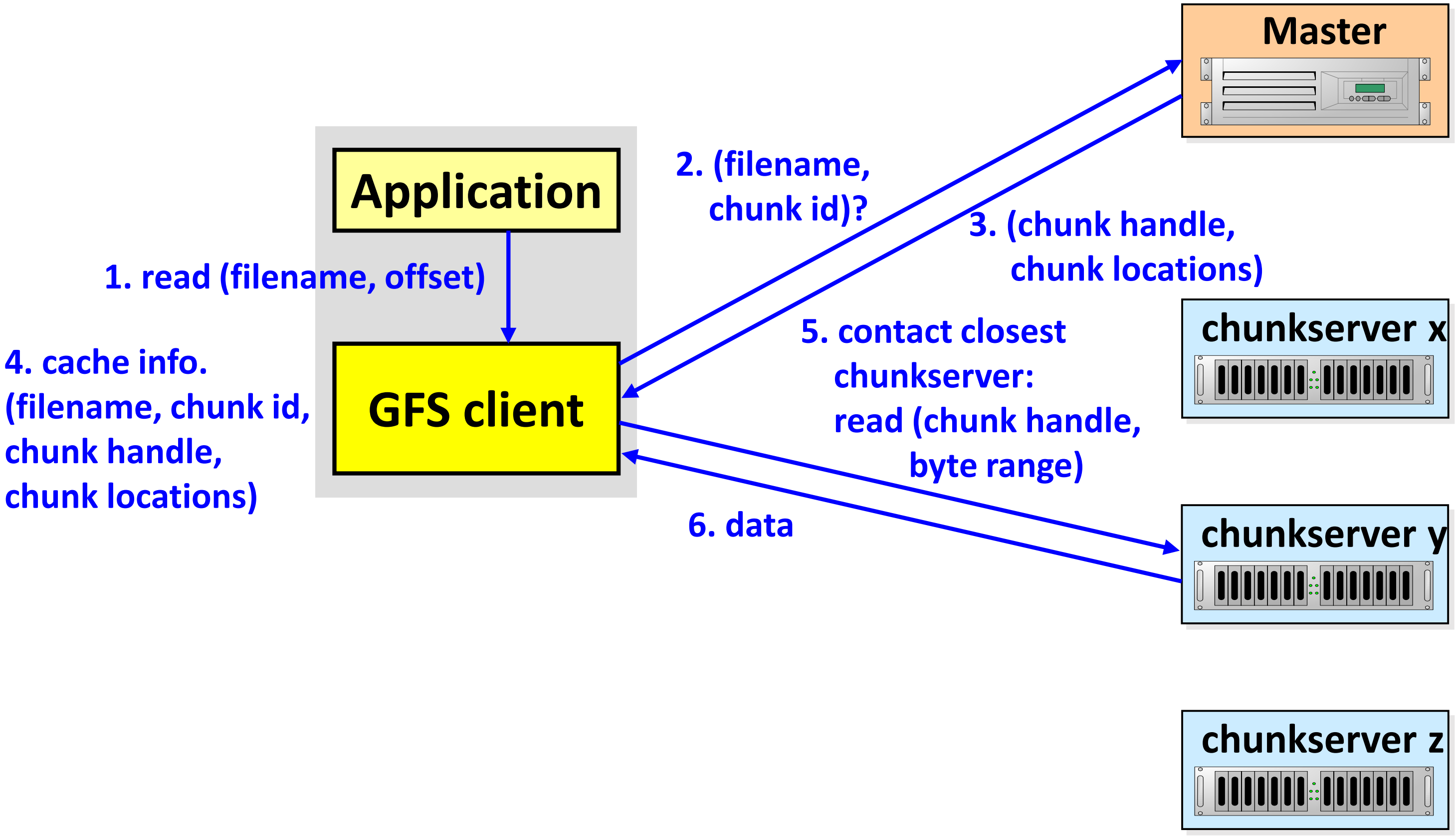
- Master
  - Manages metadata: Namespace, access control information, the mapping from files to chunks, replica locations, etc.
  - All the metadata is kept in memory
  - Controls leases, placement, replication, migration, etc.
  - Not involved in data transfer
- Chunkservers
  - Store "chunks" of data
  - Built on local Linux filesystem (no knowledge of GFS filesystem structure)
  - Periodically communicate with the master using HeartBeats
- Clients

Chunks

- Files are divided into fixed-size chunks
  - The chunk size is 64MB
  - 64-bit chunk handle is assigned by the master
  - Lazy space allocation to avoid internal fragmentation
  - Chunkservers store chunks on local disks as Linux files
  - By default, three replicas are maintained for reliability

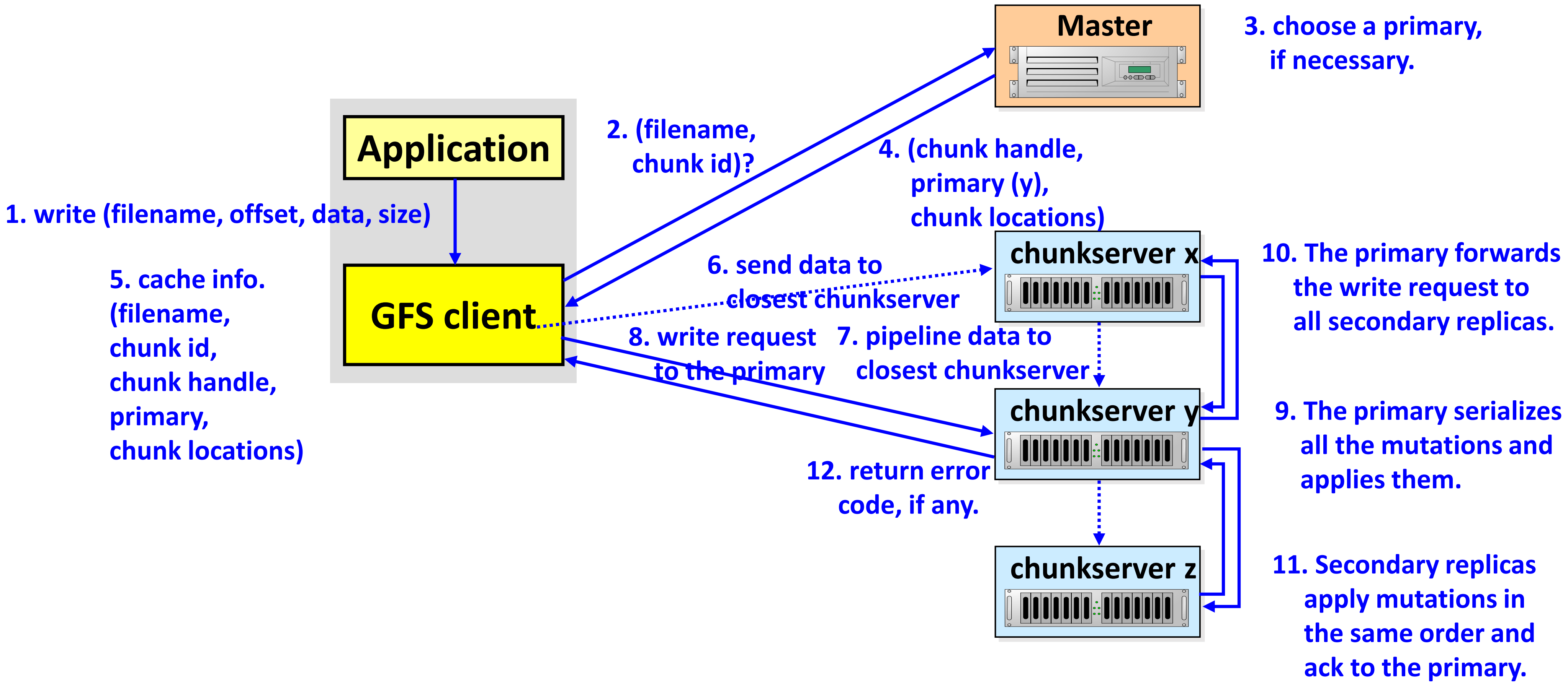
- Why a large chunk size?
  - Reduces the communication with the master
  - Reduces network overhead by keeping a persistent TCP connection to chunkserver
  - Reduces the metadata size

Reads

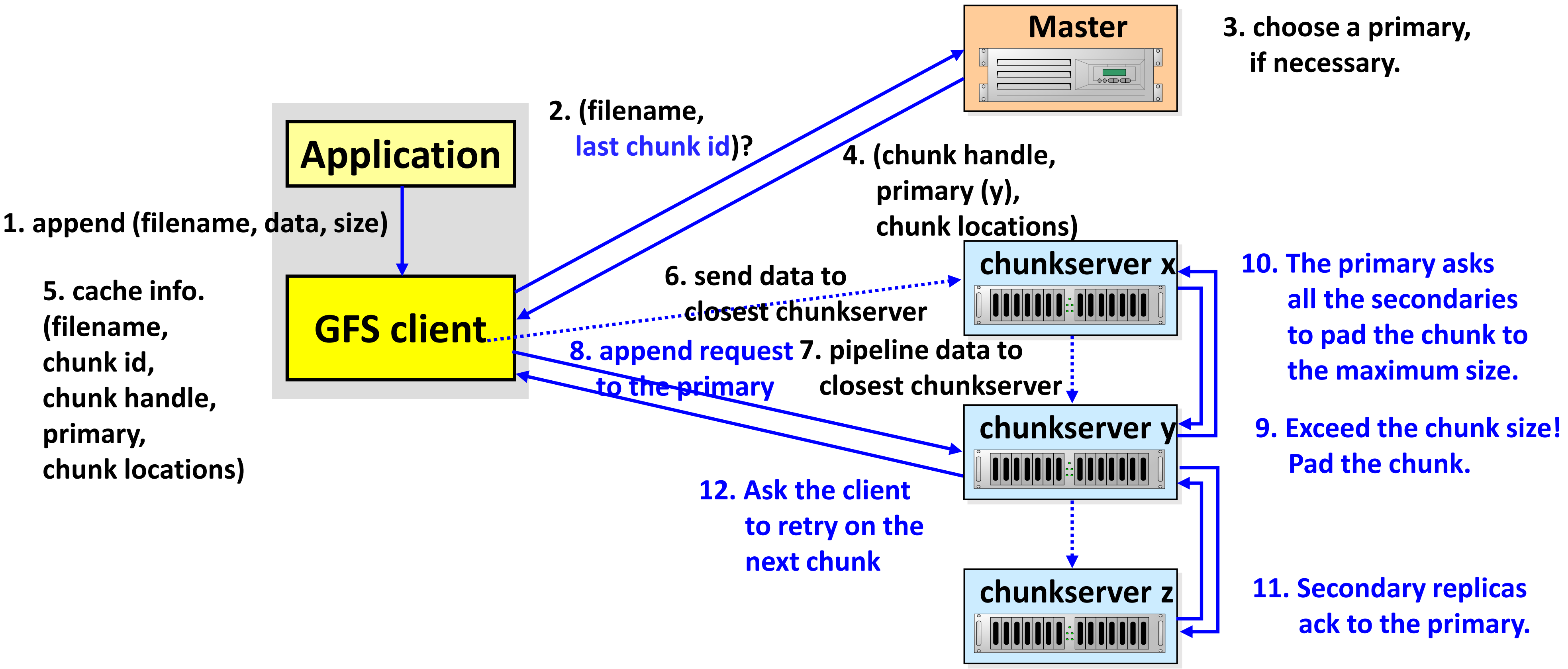


Master
Application
2. (filename,
chunk id)?
3. (chunk handle,
chunk locations)
1. read (filename, offset)
4. cache info.
(filename, chunk id,
chunk handle,
chunk locations)
GFS client
5. contact closest
chunkserver:
read (chunk handle,
byte range)
chunkserver x
chunkserver y
6. data
chunkserver z

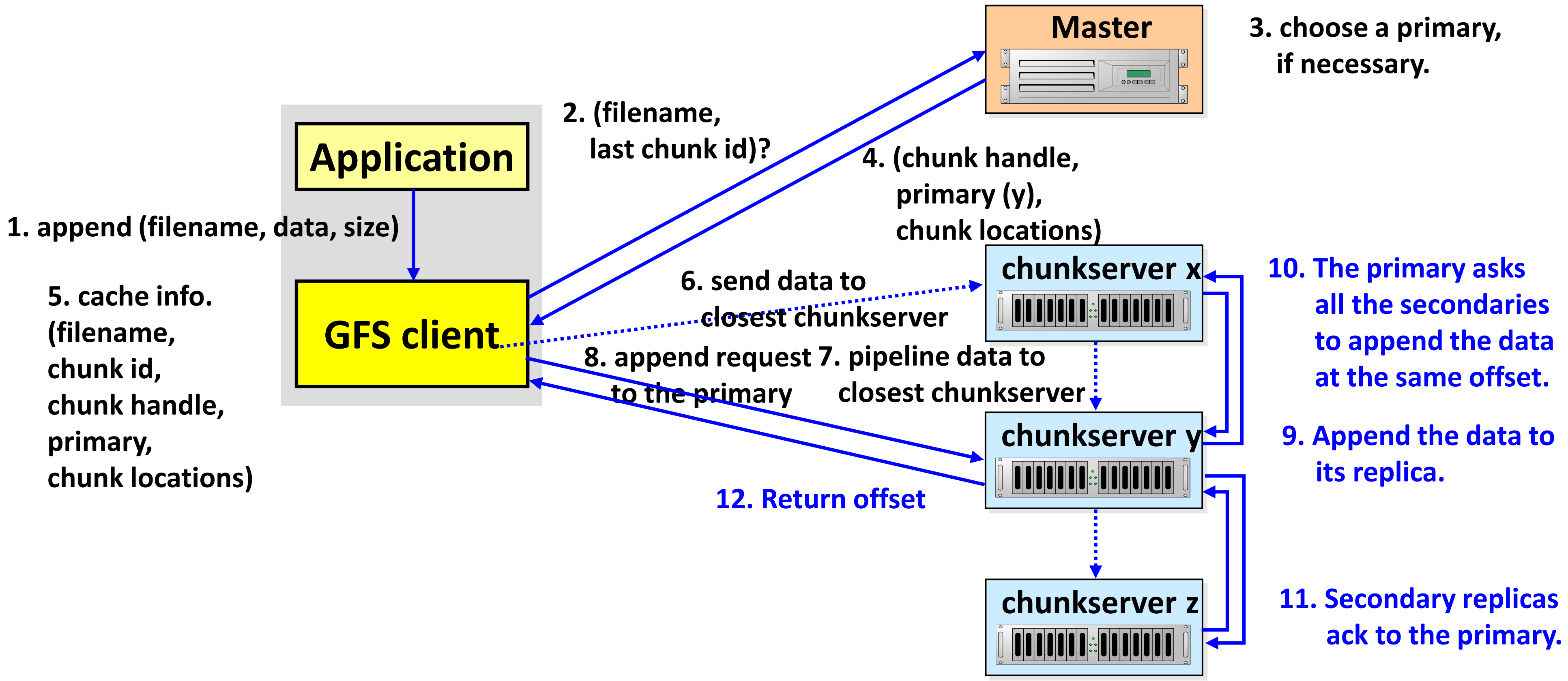
Writes


Master
3. choose a primary,
if necessary.
Application
2. (filename,
chunk id)?
4. (chunk handle,
primary (y),
chunk locations)
1. write (filename, offset, data, size)
5. cache info.
(filename,
chunk id,
chunk handle,
primary,
chunk locations)
GFS client
6. send data to
closest chunkserver
chunkserver x
10. The primary forwards
the write request to
all secondary replicas.
8. write request
to the primary
7. pipeline data to
closest chunkserver
chunkserver y
9. The primary serializes
all the mutations and
applies them.
12. return error
code, if any.
chunkserver z
11. Secondary replicas
apply mutations in
the same order and
ack to the primary.

RecordAppends



RecordAppends (Normal Case)



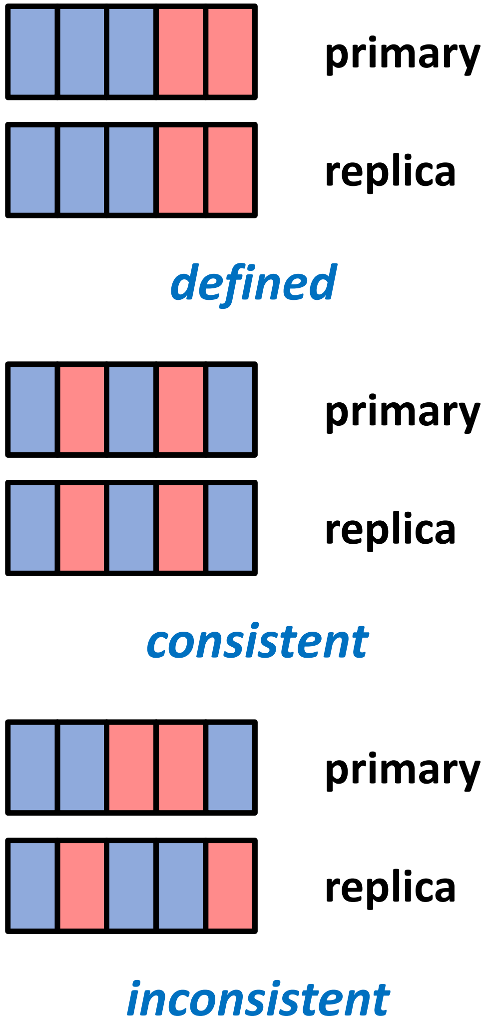
Consistency Model

- Concurrent file namespace mutations
  - Each mutation is atomic and handled exclusively by the master
  - Operation log defines a global total order
- Concurrent writes
  - The data may be broken into multiple write operations
  - The region may end up containing data fragments from multiple clients
- Concurrent RecordAppends
  - GFS appends data at least once atomically at an offset of GFS's choosing
  - Replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part

Data Consistency

- A region is consistent if all clients will always see the same data, regardless of which replicas they read from
- A region is defined if it is consistent and clients will see what the mutation writes in its entirety

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



Replica Placement

- Choose chunkservers with below-average disk space utilization
  - Equalize disk utilization
- Limit the number of "recent" creations on each chunkserver
  - Heavy write traffic will follow soon
- Spread replicas across racks
  - Maximize data reliability and availability
  - Maximize network bandwidth utilization

Re-replication

- The master re-replicate a chunk when:
  - A chunkserver becomes unavailable
  - A chunkserver reports its replica may be corrupted
  - One of disks is disabled because of errors
  - The replication goal is increased
- Minimize application disruption and data loss
  - More replicas missing → priority boost
  - Recently deleted files → priority decrease
  - Client blocking on a write → large priority boost
- Master directs copying of data from an existing replica
- Keep cloning traffic from overwhelming client traffic

Rebalancing (Chunk Migration)

- The master periodically examines the current replica distribution
- Replicas are moved for better disk space and load balancing
- A new chunkserver is gradually filled up

- Prefer to remove chunks on chunkservers with below-average free space

Garbage Collection

- Garbages: any replica not known to the master
  - Deleted files are temporarily kept for 3 days
  - Orphaned chunks
- Why not eager deletion?
  - Simple and reliable in a large-scale distributed system where component failures are common
  - Storage reclamation can be merged into the regular background activities for the master
  - The delay provides a safety net against accidental, irreversible deletion
- Users may control the policy especially when the storage is tight

Stale Replica Detection

- What if a chunkserver misses mutations?
- Chunk version number
  - Maintained by the master for each chunk
  - Increased whenever the master grants a new lease
  - Recorded in the persistent state by the master and all replicas
- Stale replicas can be detected if the chunk version number is less than the master's
- The master removes stale replicas in its regular garbage collection
- The master also informs clients of the chunk version number

High Availability

- Fast recovery
  - The master and chunkservers restart in seconds
- Chunk replication
  - Different replication levels for different parts of the namespace
  - More complicated redundancy schemes may be possible
    - Mostly appends and reads instead of small random writes
- Master replication
  - The operation log and checkpoints are replicated
  - The master can be instantly restarted or replaced
  - Shadow master provide read-only access even when the primary master is down

Data Integrity: Checksumming

- Disk failures cause data corruption
- 32-bit checksum for each 64KB block
- On checksum failure:
  - The chunkserver reports the mismatch to the master
  - The client retries other replicas
  - The master clones the chunk from another replica
  - The corrupted chunk is deleted
- Chunkservers can verify the checksum during idle periods

GFS Usage @ Google (As of 2003)

- 10+ clusters
- Filesystem clusters up to 1000+ machines
- Pools of 1000+ clients
- 350+ TB filesystems
- 500+ MB/s read/write load
- Operational in the presence of frequent hardware failures

GFS Usage @ Google (As of 2009)

- 200+ clusters
- Many clusters of 1000s of machines
- Pools of 1000s of clients
- 4+ PB filesystems
- 40 GB/s read/write load
- Operational in the presence of frequent hardware failures

Source: J. Dean, "Designs, Lessons and Advice from Building Large Distributed Systems," LADIS Keynote, 2009.

(Their) Lessons

- Inexpensive commodity components can be the basis of a large-scale reliable system
- Adjusting the API, e.g., RecordAppend, can enable large distributed applications
- It solves the problem for our initial target applications, but …
- Build something fault tolerant and people will find more uses than you expect

From GFS to Colossus

- Workload changes
  - Hundreds of TBs → Tens of PBs (100x increase)
  - Batch-oriented workload (crawling and indexing) → Interactive applications (Gmail, …)
  - Many files < 64MB

- Colossus: Google's next-generation cluster-level file system
  - Distributed master architecture
  - Use BigTable for metadata storage
  - 100M files per master, hundreds of masters
  - 1MB chunk size
  - Data typically written using Reed-Solomon (1.5x)