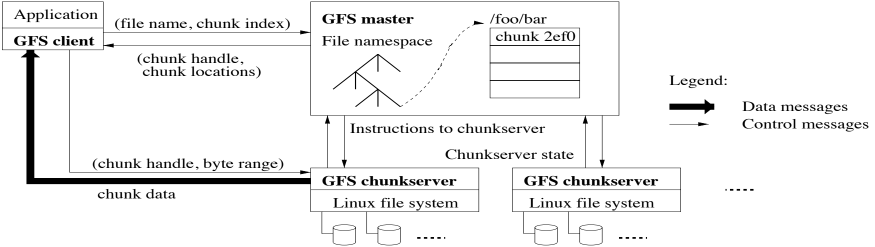
# Dependable Distributed Systems – 5880V/UE

Part 8. Distributed File Systems – 2021-10-14

Prof. Dr. Hans P. Reiser | WS 2021/22

UNIVERSITÄT PASSAU




(source: Ghemawat et al.: The Google File System, SOSP 2013)

# Today's class

**What concepts allow you to store large amount of data in distributed systems?**

# Today's class

**What concepts allow you to store large amount of data in distributed systems?**

Requirements

- Reliability
- Security
- Consistency
- ...

# Today's class

**What concepts allow you to store large amount of data in distributed systems?**

Requirements

- Reliability
- Security
- Consistency
- ...

Context

- in closely coupled, large-scale data centers

# Overview

1. Large-scale file systems
   - Example: Google File System (GFS)

# High volume distributed file systems

Examples:

- Google File System – GFS
  - Slides mostly based on the GFS SOSP'03 presentation
- HDFS (Yahoo!/Apache)
  - Similar, but things have different names, some simplifications
  - Open source
  - Used by: Amazon/A9, Facebook, Google, Joost, Last.fm, New York Times, PowerSet, Veoh, Yahoo!, . . .

# Introduction

Design constraints:

- Component failures are the norm
  - 1000s of components
  - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
  - Mechanisms: monitoring, error detection, fault masking, automatic recovery
- Files are huge by traditional standards
  - Multi-GB files are common
  - *Important design option: modest number of huge files*

# Introduction

Design constraints:

- Most modifications are appends
  - Random writes are practically nonexistent
  - Many files are written once, and read sequentially
- Two types of reads
  - Large streaming reads
  - Small random reads
- Sustained throughput more important than latency
- File system APIs are open to changes

# Interface Design

- Not POSIX compliant
- Standard operations
  - create, delete, open, close, read, write
- Additional operations
  - Snapshot (cheap copy of a file or a directory)
  - Record append (allows concurrent atomic appends from multiple clients)

# Architectural Design

A GFS cluster:

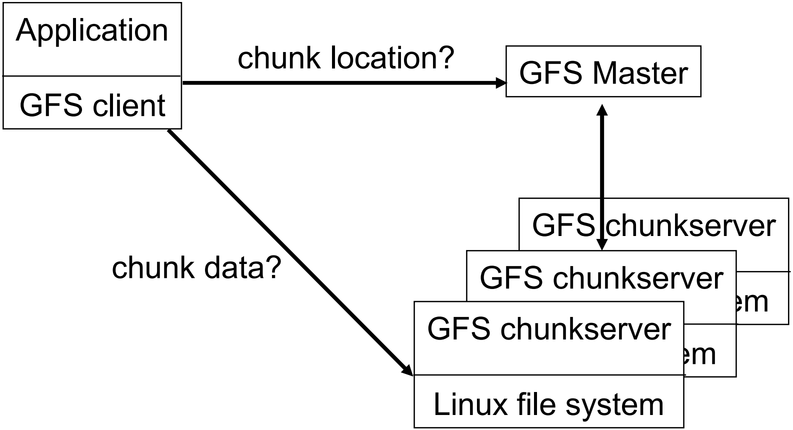
- A single **master** $\longrightarrow$ contains *metadata* only
- Multiple **chunkservers** per master $\longrightarrow$ contain *data*
  - Accessed by multiple clients
- Running on commodity Linux machines

A file:

- Represented as fixed-sized **chunks**
  - Labeled with 64-bit unique global IDs
  - Stored at chunkservers
  - Replicated in several chunkservers (3 by default)

| **File x** |
|---|
| chunk 1 |
| chunk 2 |
| chunk 3 |
| … |
| chunk n |

# Architectural Design (2)



# Architectural Design (3)

Master server:

- Maintains all metadata
  - Name space, access control, file-to-chunk mappings, garbage collection, chunk migration
- Bottleneck, so they try to minimize its workload / functionality

GPS clients:

- Consult master for metadata
- Access data from chunkservers
- No caching at clients and chunkservers due to the frequent case of streaming (well, almost)

# Single-Master Design

- Simple
- Master answers only chunk locations
- A client typically asks for multiple chunk locations in a single request
- The master also predictively provides chunk locations immediately following those requested

# Chunk Size

- 64 MB
- Fewer chunk location requests to the master
- Reduced overhead to access a chunk
- Persistent TCP connection to the chunkserver over an extended period of time
- Fewer metadata entries
  - Kept in memory
- Potential problem: Hot spot development

# Metadata

Three major types

a) File and chunk namespaces — persistent

b) File-to-chunk mappings — persistent

c) Locations of a chunk’s replicas — non persistent

# Metadata

All kept in memory

- Fast!
- Quick global scans
  - Garbage collections
  - Reorganizations
- $<$ 64 bytes per 64 MB of data
- Prefix compression
- But some data is also logged

# Master fault tolerance (1)

Locations of a chunk's replicas is not persistent — (c)

- Master polls chunkservers at startup
  - "What chunks do you have?"
- Use heartbeat messages to monitor chunkservers
- Simplicity
- On-demand approach vs. coordination
  - On-demand wins when changes (failures) are frequent

# Master fault tolerance (2)

- File and chunk namespaces and File-to-chunk mappings — (a),(b)
- Variation of **passive replication**
- Metadata updates are **logged**
  - Log replicated on remote machines
- Take global **snapshots (checkpoints)** to truncate logs
  - Memory mapped (no serialization/deserialization)
  - Checkpoints can be created while updates arrive
- Recovery
  - Latest checkpoint + subsequent log files

# Consistency Model

Namespace mutations (e.g., file creations / deletions) are atomic

- Total order, handled exclusively by the master

Relaxed consistency

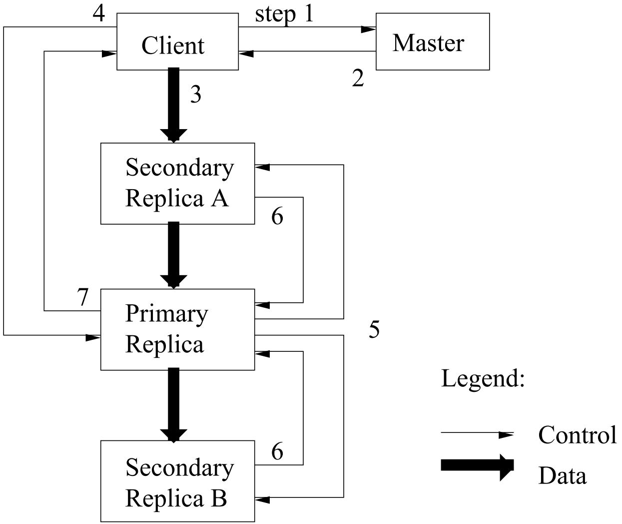
- Concurrent changes to a chunk are:
  - Consistent — all clients see the same data
  - Undefined — the chunk may end with a mix of the writes done
- An append is atomically committed at least once
  - But GFS defines the offset (i.e., position of the append at the file)
- All changes to a chunk are applied in the same order to all replicas
- Use version numbers to detect missed updates

# System Interactions

(objective: spare the master)

- The master grants a **chunk lease** to a replica
  - Replica = chunkserver that has the chunk
- The replica holding the lease determines the order of updates to all replicas
  - The replica becomes the **primary** for that chunk
- Lease
  - 60 second timeouts
  - Can be extended indefinitely
  - Extension request piggybacked on heartbeat messages
  - After timeout expires, master can grant new leases

# Write control and data flow



# Data Flow

- Separation of control and data flows
  - Avoid network bottleneck
- Updates are pushed linearly among replicas
- Pipelined transfers

- Instead of sending data around as needed, GFS “thinks” about the paths it should follow in order to minimize bottlenecks

# Snapshot of a file

- Copy-on-write approach
- When the Master receives a file snapshot request:
  - (1) Revoke all leases on the file chunks
    $\Rightarrow$ all subsequent updates to the file must pass through the Master
    $\Rightarrow$ Master can do a copy of the file first (any updates are logged while taking the snapshot)
  - (2) Replicate the metadata but leave it pointing to the same chunks
  - (3) Apply the log to a copy of the metadata
  - A chunk is not copied until the next update (copy-on-write)

# Master Operation

Master:

- Executes all namespace operations
- Manages chunk replicas throughout the system
  - Makes placement decisions
  - Creates new chunks and replicas
  - Coordinates activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage

Some characteristics:

- Does not have a data structure per-directory
- No hard links and symbolic links
- Full path name to metadata mapping
  - With prefix compression

# Locking Operations

Locking a file

- To lock /d1/d2/leaf
- Need to lock /d1, /d1/d2, and /d1/d2/leaf
- Totally ordered locking to prevent deadlocks

# Replica Placement

- *Replicas of chunks*
- Goals:
  - Maximize data reliability and availability
  - Maximize network bandwidth
- Need to spread chunk replicas across machines and racks
- Higher priority to replica chunks with lower replication factors
- Limited resources spent on replication

# Garbage Collection of files

- Garbage collection is simpler than eager deletion due to
  - Unfinished replica creation
  - Lost deletion messages
- Deleted files are hidden for three days
- Then they are garbage collected
- Combined with other background operations (taking snapshots)
- Safety net against accidents

# Fault Tolerance and Diagnosis

- Fast recovery
  - Master and chunkserver are designed to restore their states and start in seconds regardless of how they terminated (normal, abnormal, killed, . . .)
- Chunk replication
- Master replication
  - *Shadow masters* provide read-only access when the primary master is down
  - If master fails it is restarted immediately
  - If it cannot be restarted (due to hardware failure), a new one is elected from outside GFS (Chubby?)

# Fault Tolerance and Diagnosis

Data integrity

- A chunk is divided into 64-KB blocks
- Each with its checksum
- Verified at read and write times
- Also background scans for rarely used data
  - Locally at each chunkserver

# Measurements

- Chunkserver workload
  - Bimodal distribution of small and large files
  - Ratio of write to append operations: 3:1 to 8:1
  - Virtually no overwrites
- Master workload
  - Most request for chunk locations and open files
- Reads achieve 75% of the network limit
- Writes achieve 50% of the network limit

# Major Innovations

- File system API tailored to specific workload
- Single-master design to simplify coordination
- Metadata fit in memory
- Flat namespace