Today: Coda, xFS

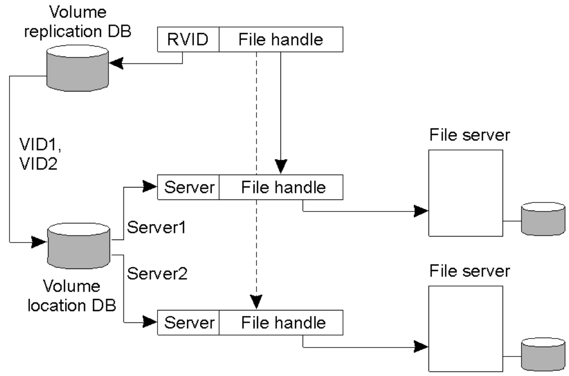
- Distributed File Systems
- Case Study: Coda File System

- Brief overview of other file systems
  - xFS
  - Log structured file systems
  - HDFS
  - Object Storage Systems

Coda Overview

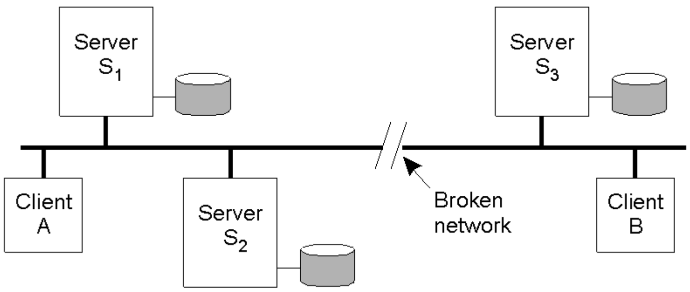
- DFS designed for mobile clients
  - Nice model for mobile clients who are often disconnected
    - Use file cache to make ***disconnection*** transparent
    - At home, on the road, away from network connection
- Coda supplements file cache with user preferences
  - E.g., always keep this file in the cache
  - Supplement with system learning user behavior
- How to keep cached copies on disjoint hosts consistent?
  - In mobile environment, "simultaneous" writes can be separated by hours/days/weeks

File Identifiers



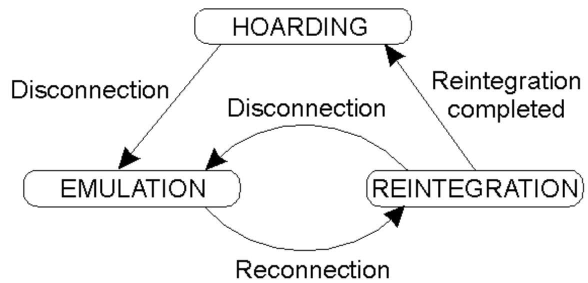
- Each file in Coda belongs to exactly one volume
  - Volume may be replicated across several servers
  - Multiple logical (replicated) volumes map to the same physical volume
  - 96 bit file identifier = 32 bit RVID + 64 bit file handle

Server Replication



- Use replicated writes: read-once write-all
  - Writes are sent to all AVSG (all accessible replicas)
- How to handle network partitions?
  - Use optimistic strategy for replication
  - Detect conflicts using a Coda version vector
  - Example: [2,2,1] and [1,1,2] is a conflict => manual reconciliation

Disconnected Operation



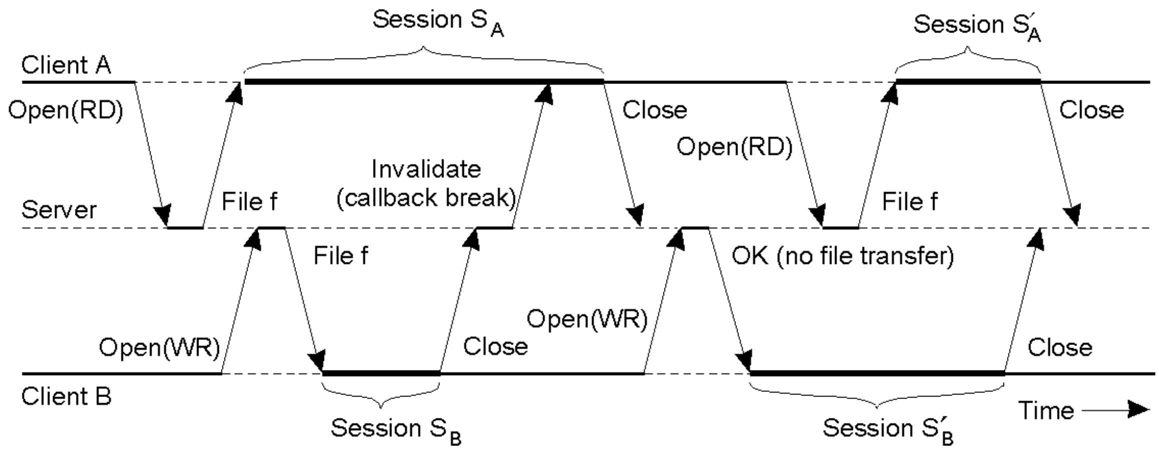
- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection
  - Prefetch all files that may be accessed and cache (hoard) locally
  - If AVSG=0, go to emulation mode and reintegrate upon reconnection

Transactional Semantics

- Network partition: part of network isolated from rest
  - Allow conflicting operations on replicas across file partitions
  - Reconcile upon reconnection
  - Transactional semantics => operations must be serializable
    - Ensure that operations were serializable after thay have executed
  - Conflict => force manual reconciliation

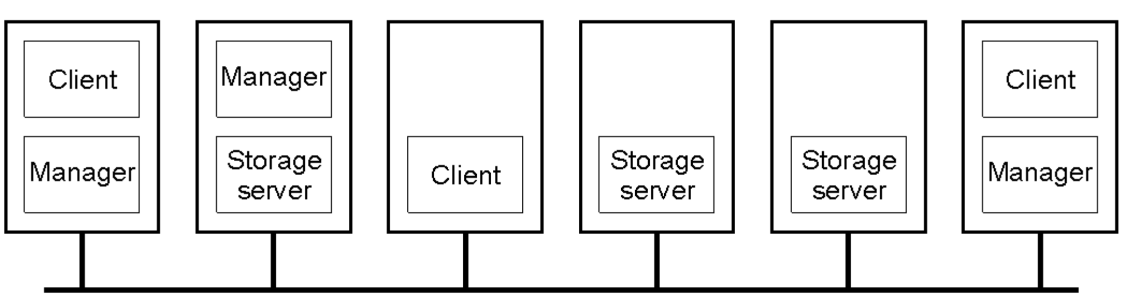
Client Caching

- Cache consistency maintained using callbacks



Overview of xFS.

- Key Idea: fully distributed file system [serverless file system]
  – Remove the bottleneck of a centralized system
- xFS: x in "xFS" => no server
- Designed for high-speed LAN environments



xFS Summary

- Distributes data storage across disks using software RAID and log-based network striping
  - RAID == Redundant Array of Independent Disks
- Dynamically distribute control processing across all servers on a per-file granularity
  - Utilizes serverless management scheme
- Eliminates central server caching using cooperative caching
  - Harvest portions of client memory as a large, global file cache.

Array Reliability

- Reliability of N disks = Reliability of 1 Disk ÷ N

  50,000 Hours ÷ 70 disks = 700 hours

  Disk system MTTF: Drops from 6 years to 1 month!

- Arrays (without redundancy) too unreliable to be useful!

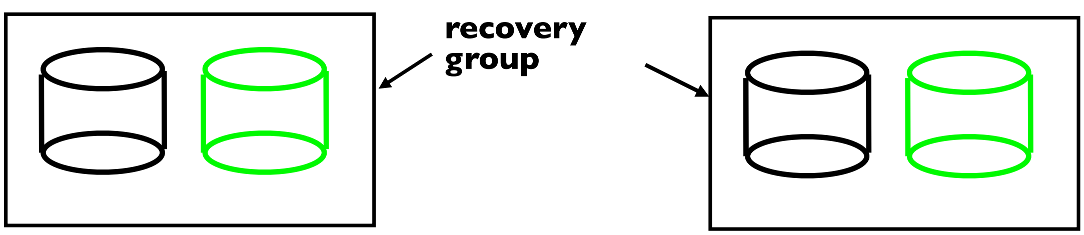
Hot spares support reconstruction in parallel with access: very high media availability can be achieved

RAID Overview

- Basic idea: files are "striped" across multiple disks
- Redundancy yields high data availability
  - Availability: service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array
  - Capacity penalty to store redundant info
  - Bandwidth penalty to update redundant info

Slides courtesy David Patterson

Redundant Arrays of Inexpensive Disks RAID 1: Disk Mirroring/Shadowing

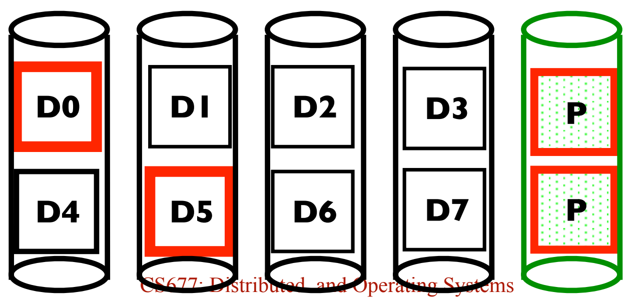


- Each disk is fully duplicated onto its "mirror"
  - Very high availability can be achieved
- Bandwidth sacrifice on write:
  - Logical write = two physical writes
  - Reads may be optimized
- Most expensive solution: 100% capacity overhead

- (RAID 2 not interesting, so skip…involves Hamming codes)

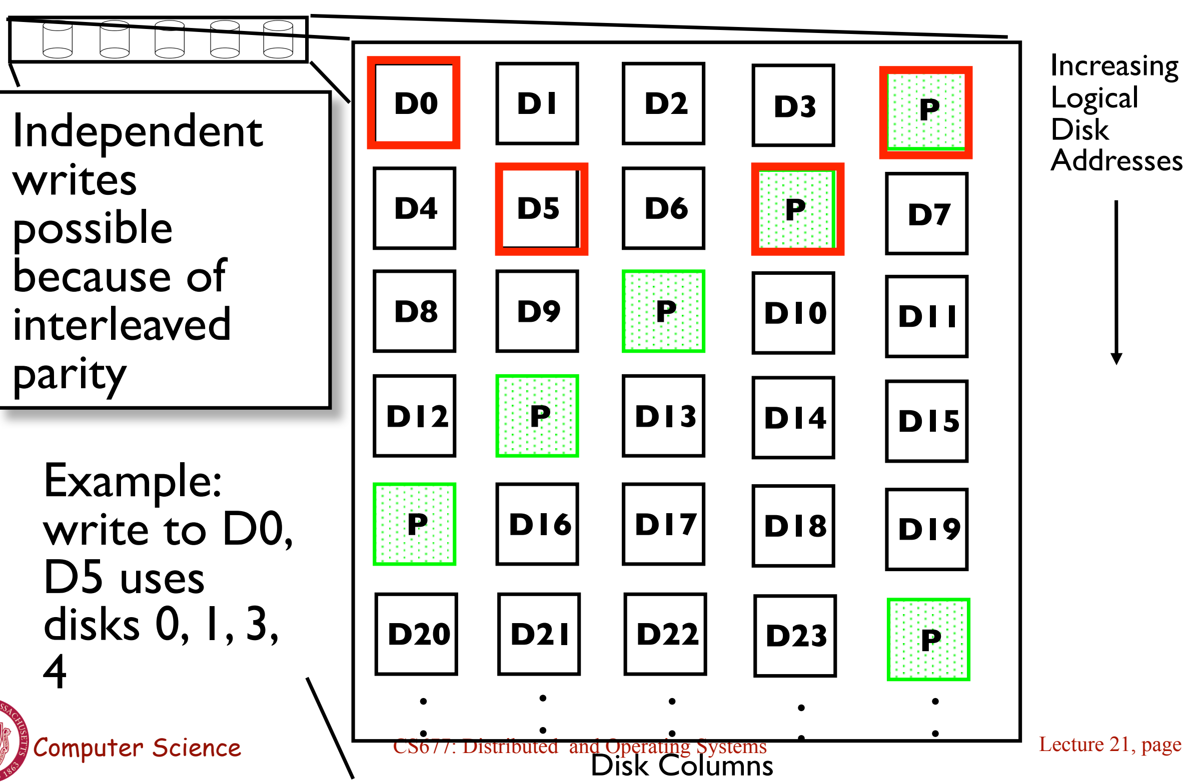
Inspiration for RAID 5

- Use parity for redundancy
  - $D0 \otimes D1 \otimes D2 \otimes D3 = P$
  - If any disk fails, then reconstruct block using parity:
    - e.g., $D0 = D1 \otimes D2 \otimes D3 \otimes P$
- RAID 4: all parity blocks stored on the same disk
  - Small writes are still limited by Parity Disk: Write to D0, D5, both also write to P disk
  - Parity disk becomes bottleneck



Redundant Arrays of Inexpensive Disks RAID 5: High I/O Rate Interleaved Parity

Independent writes possible because of interleaved parity



Example: write to D0, D5 uses disks 0, 1, 3, 4

xFS uses software RAID

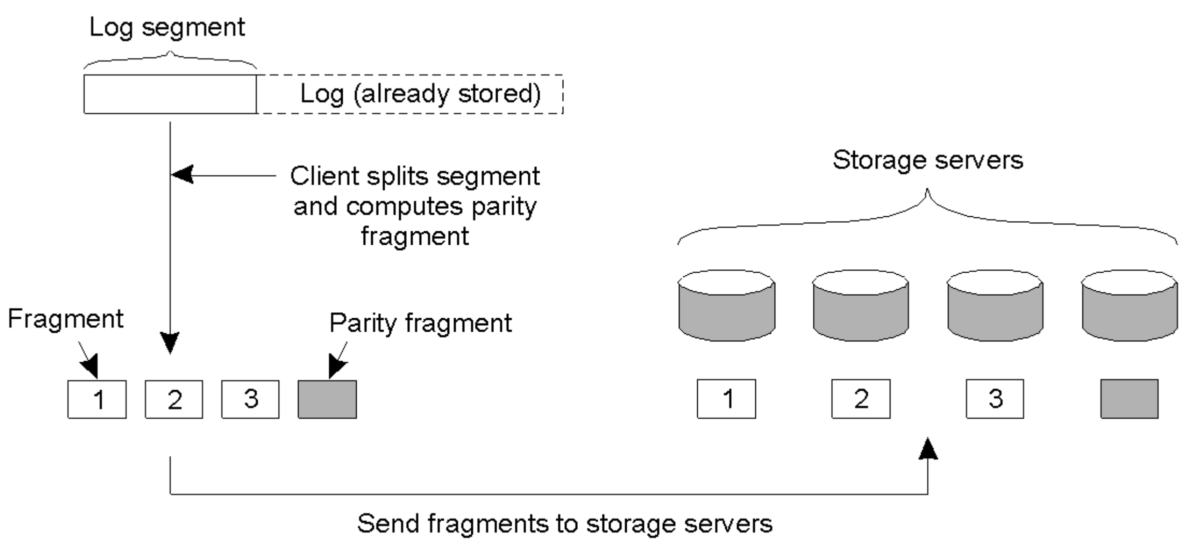
- Two limitations
  - Overhead of parity management hurts performance for small writes
    - Ok, if overwriting all N-1 data blocks
    - Otherwise, must read old parity+data blocks to calculate new parity
    - Small writes are common in UNIX-like systems
  - Very expensive since hardware RAIDS add special hardware to compute parity

Log-structured FS

- Provide fast writes, simple recovery, flexible file location method
- Key Idea: buffer writes in memory and commit to disk in large, contiguous, fixed-size log segments
  - Complicates reads, since data can be anywhere
  - Use per-file inodes that move to the end of the log to handle reads
  - Uses in-memory imap to track mobile inodes
    - Periodically checkpoints imap to disk
    - Enables "roll forward" failure recovery
- Drawback: must clean "holes" created by new writes

Combine LFS with Software RAID

- The principle of log-based striping in xFS
  - Combines striping and logging

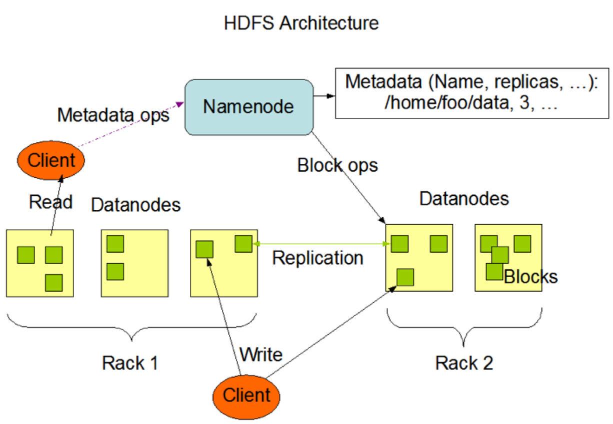


HDFS

- Hadoop Distributed File System
  - High throughput access to application data
  - Optimized for large data sets (accessed by Hadoop)
- Goals
  - Fault-tolerant
  - Streaming data access: batch processing rather than interactive
  - Large data sets: scale to hundreds of nodes
  - Simple coherency model: WORM (files don't change, append)
  - Move computation to the data when possible

HDFS Architecture

- Principle: meta data nodes separate from data nodes
- Data replication: blocks size and replication factor configurable



Google File System

- Master-slave; file divided into chunks (replicated thrice)
- Atomic writes

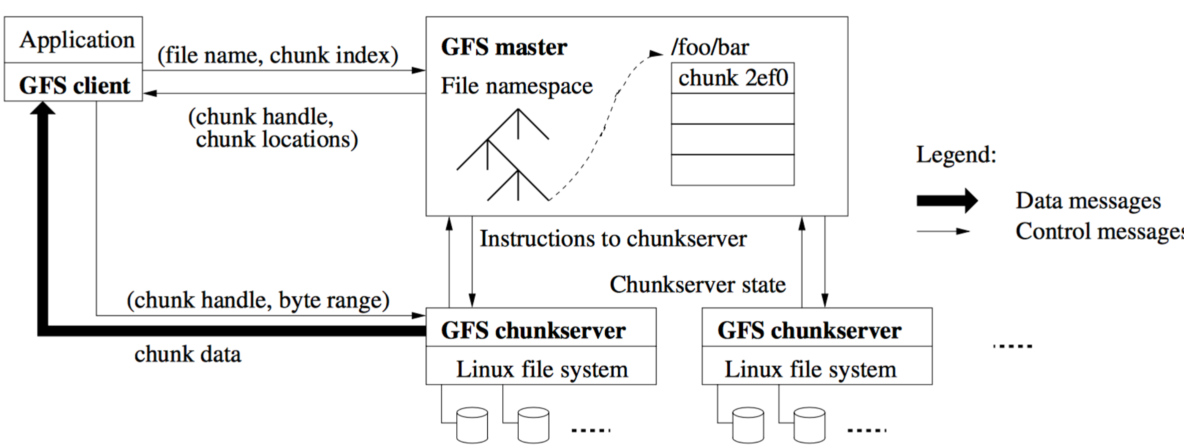


Figure 1: GFS Architecture

Object Storage Systems

- Use handles (e.g., HTTP) rather than files names
  - Location transparent and location independence
  - Separation of data from metadata
- No block storage: objects of varying sizes
- Uses
  - Archival storage
    - can use internal data de-duplication
  - Cloud Storage : Amazon S3 service
    - uses HTTP to put and get objects and delete
    - Bucket: objects belong to bucket/ partitions name space