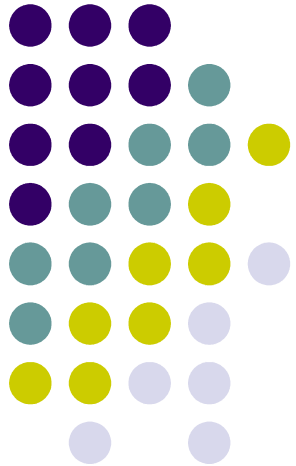
Distributed File Systems

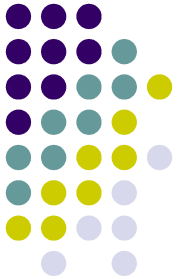
(Chapter 14, M. Satyanarayanan)

CS 249

Kamal Singh



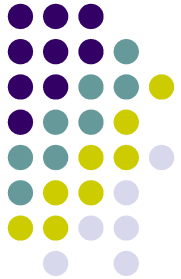
Topics



- Introduction to Distributed File Systems
- Coda File System overview
- Communication, Processes, Naming, Synchronization, Caching & Replication, Fault Tolerance and Security
- Summary

- Brief overview of Distributed Google File System (GFS)

Introduction



Distributed File Systems is a File System that aims to support file sharing, resources in the form of secure and persistent storage over a network.

Distributed File Systems (DFS)

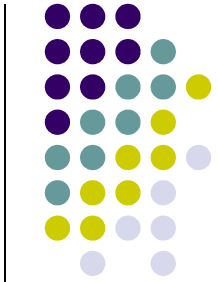
- DFS stores files on one or more computers and make these files accessible to clients, where they appear as normal files
- Files are widely available
- Sharing the files is easier than distributing individual copies
- Backups and security easier to manage

Distributed File Systems (DFS)

Issues in designing a good DFS

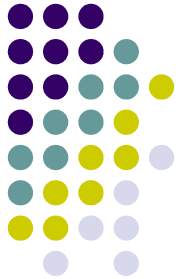
- File transfer can create
  - Sluggish performance
  - Latency
- Network bottlenecks and server overload can occur
- Security of data is important
- Failures have to be dealt without affecting clients

Coda File System (CFS)



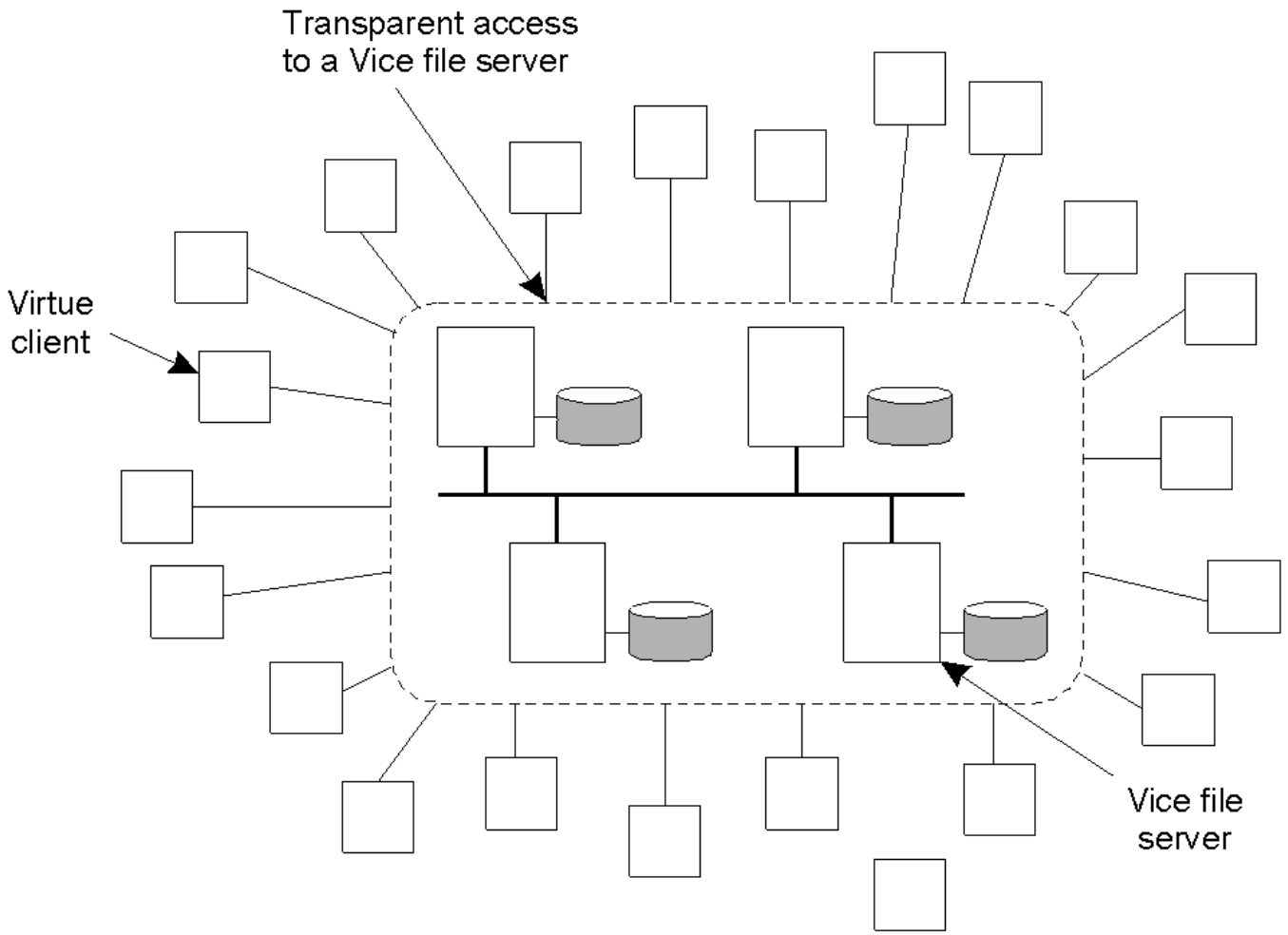
- Coda has been developed in the group of M. Satyanarayanan at Carnegie Mellon University in 1990's
- Integrated with popular UNIX operating systems
- CFS main goal is to achieve high availability
- Advanced caching schemes
- Provide transparency

Architecture

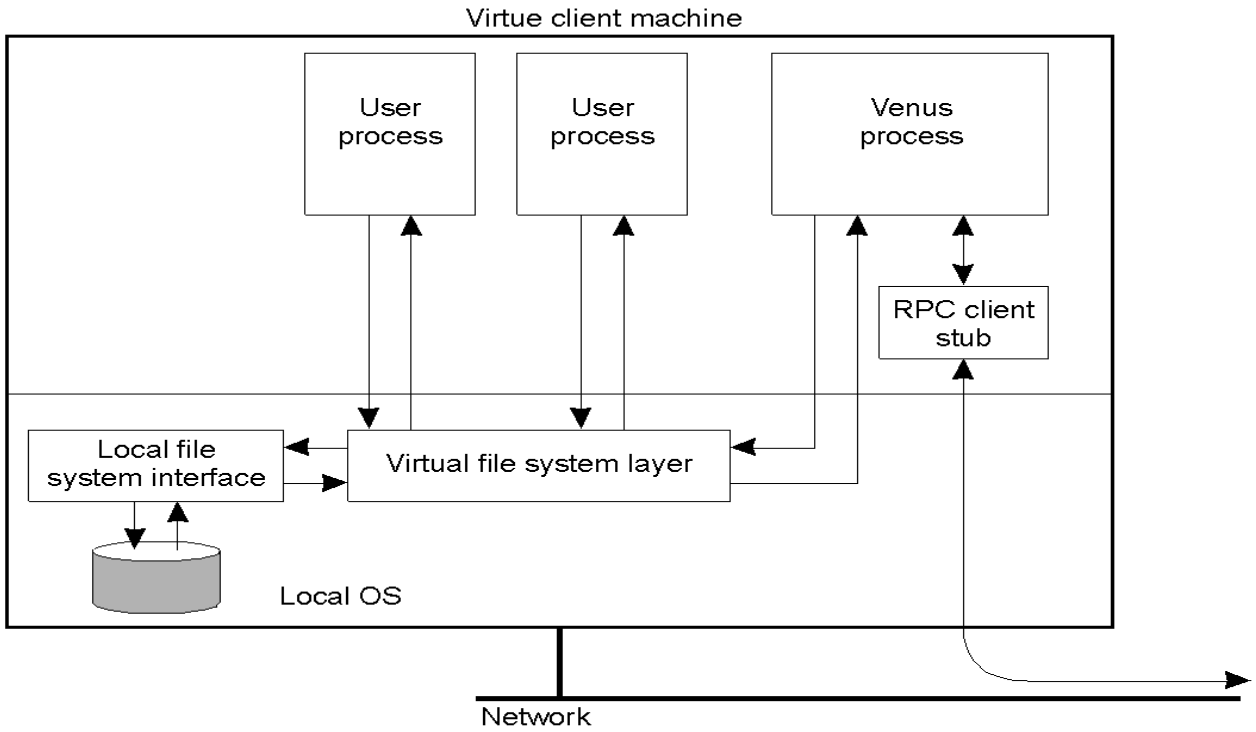


- Clients cache entire files locally
- Cache coherence is maintained by the use of callbacks (inherit from AFS)
- Clients dynamically find files on server and cache location information
- Token-based authentication and end-to-end encryption is used

Overall organization of Coda



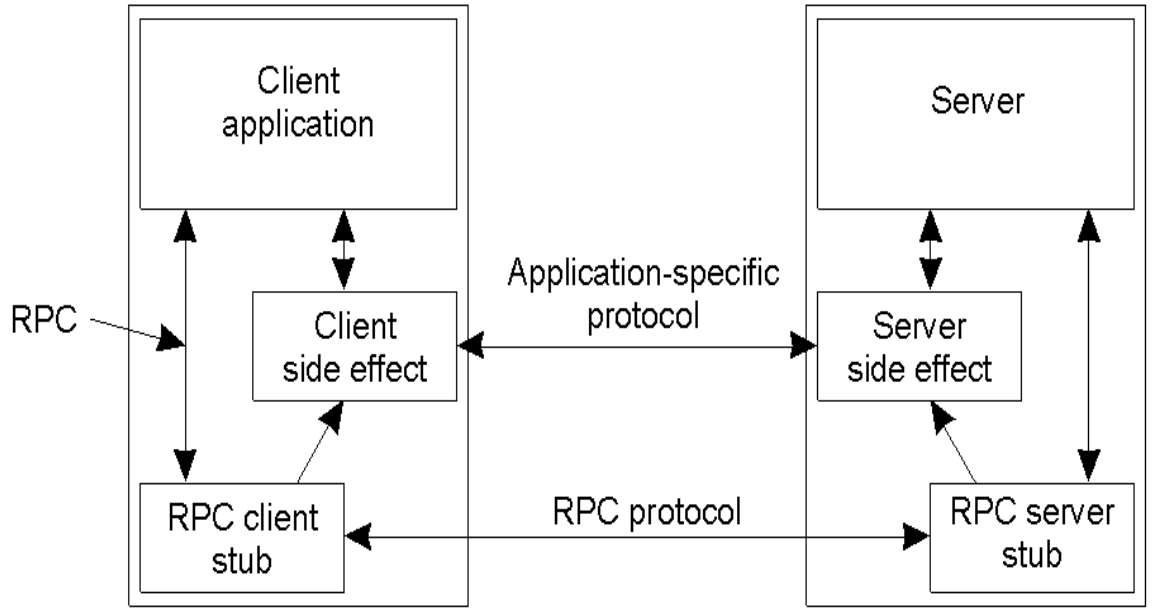
Virtue client machine



The internal organization of a Virtue workstation

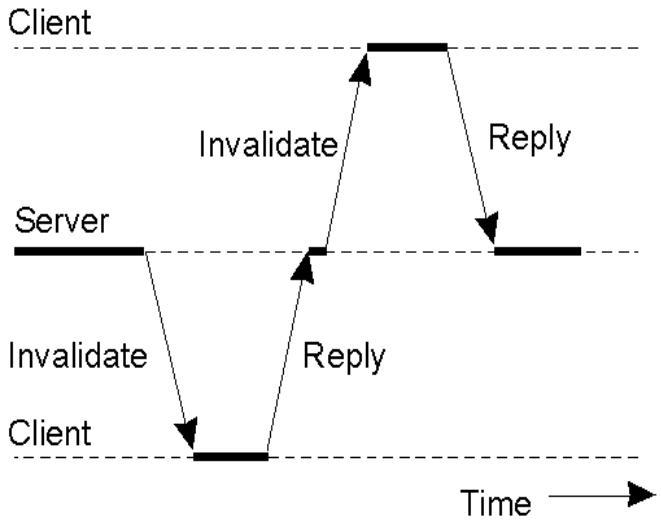
- Designed to allow access to files even if server is unavailable
- Uses VFS to intercepts calls from client application

Communication in Coda

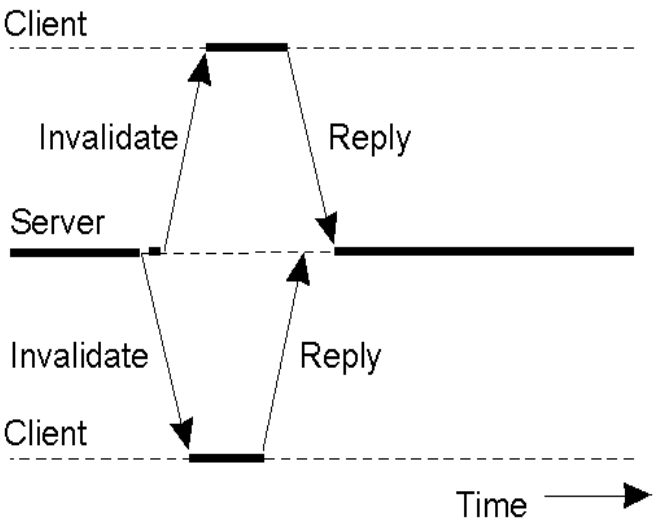


- Coda uses RPC2: a sophisticated *reliable* RPC system
  - Start a new thread for each request, server periodically informs client it is still working on the request
- RPC2 supports ***side-effects****:* application-specific protocols
  - Useful for video streaming
- RPC2 also has multicast support

Communication in Coda



(a)



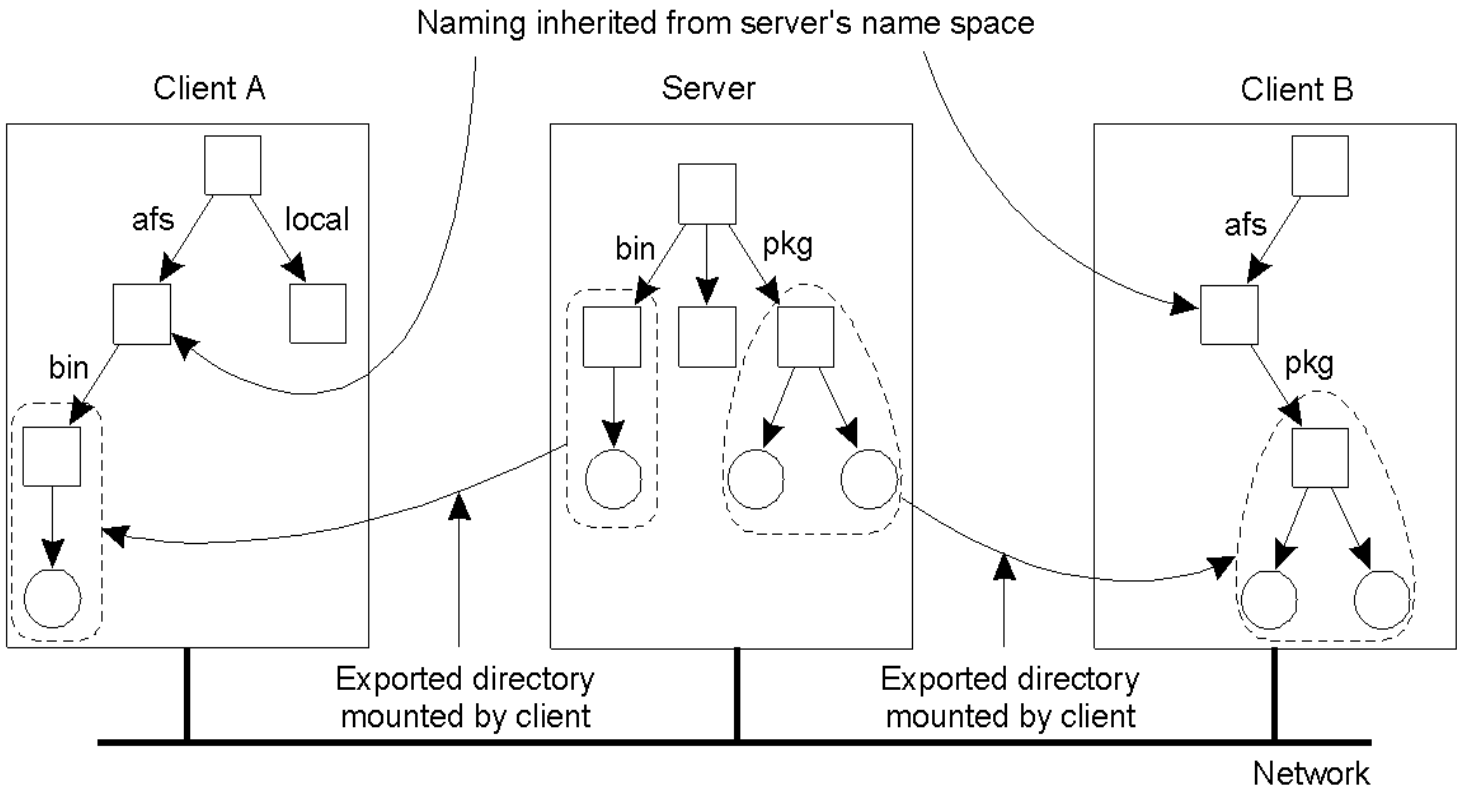
(b)

- Coda servers allow clients to cache whole files
- Modifications by other clients are notified through invalidation messages require **multicast RPC**
  a) Sending an invalidation message one at a time
  b) Sending invalidation messages in parallel

Processes in Coda

- Coda maintains distinction between client and server processes
- Client – Venus processes
- Server – Vice processes
- Threads are nonpreemptive and operate entirely in user space
- Low-level thread handles I/O operations

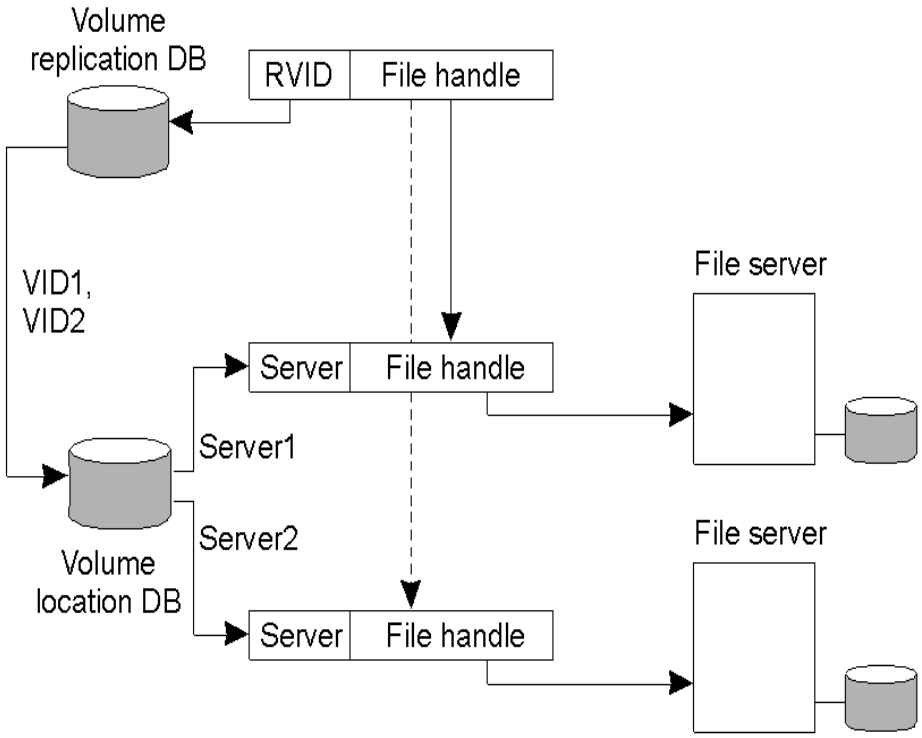
Naming in Coda



Clients have access to a single shared name space.

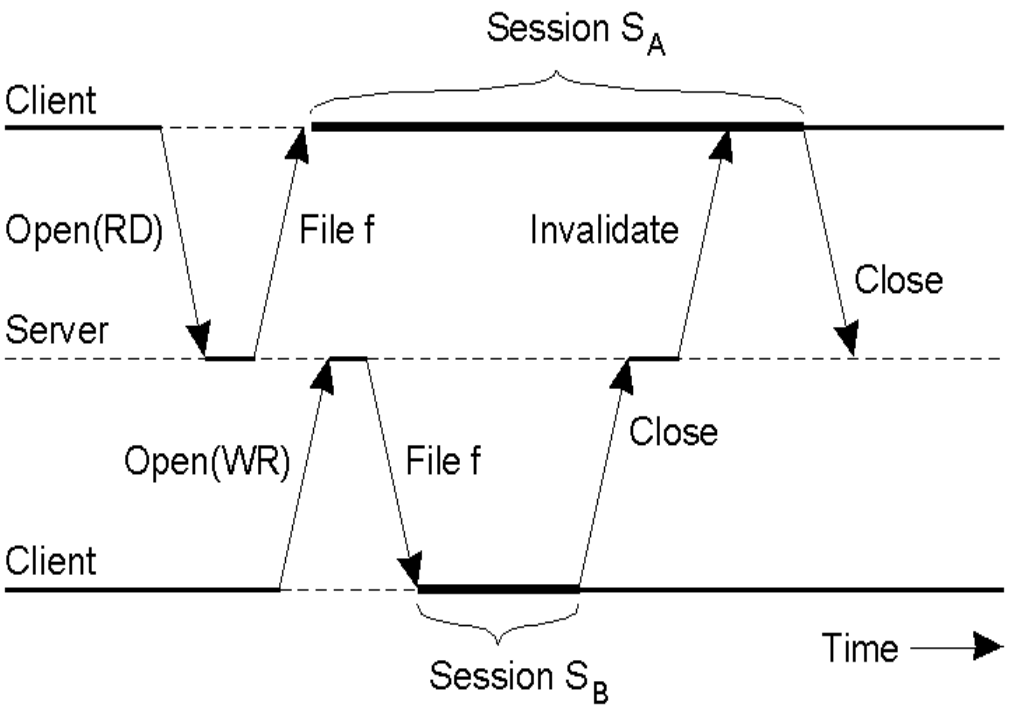
Notice Client A and Client B!

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

Synchronization in Coda



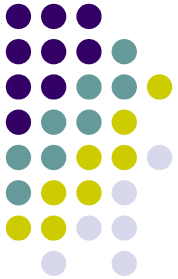
- File open: transfer entire file to client machine
- Uses session semantics: each session is like a transaction
  - Updates are sent back to the server only when the file is closed

Transactional Semantics

File-associated data	Read?	Modified?
File identifier	Yes	No
Access rights	Yes	No
Last modification time	Yes	Yes
File length	Yes	Yes
File contents	Yes	Yes

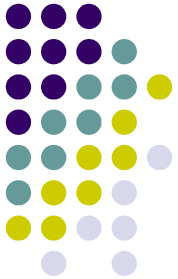
- Partition is a part of network that is isolated from rest (consist of both clients and servers)
  - Allow conflicting operations on replicas across file partitions
  - Resolve modification upon reconnection
  - Transactional semantics: operations must be serializable
    - Ensure that operations were serializable *after they have executed*
  - Conflict force manual reconciliation

Caching in Coda



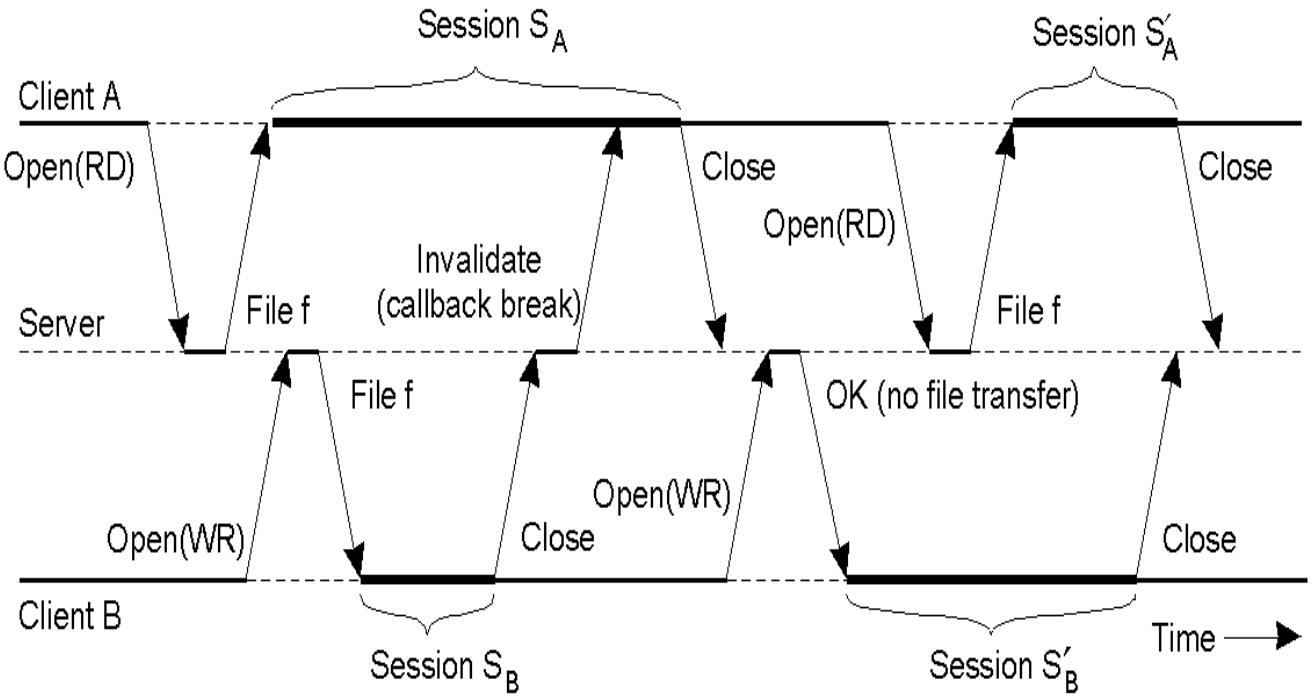
- Caching:
  - Achieve scalability
  - Increases fault tolerance
- How to maintain data consistency in a distributed system?
- Use **callbacks** to notify clients when a file changes
- If a client modifies a copy, server sends a **callback break** to all clients maintaining copies of same file

Caching in Coda



- Cache consistency maintained using **callbacks**
- Vice server tracks all clients that have a copy of the file and provide **callback promise**
  - Token from Vice server
  - Guarantee that Venus will be notified if file is modified
- Upon modification Vice server send invalidate to clients

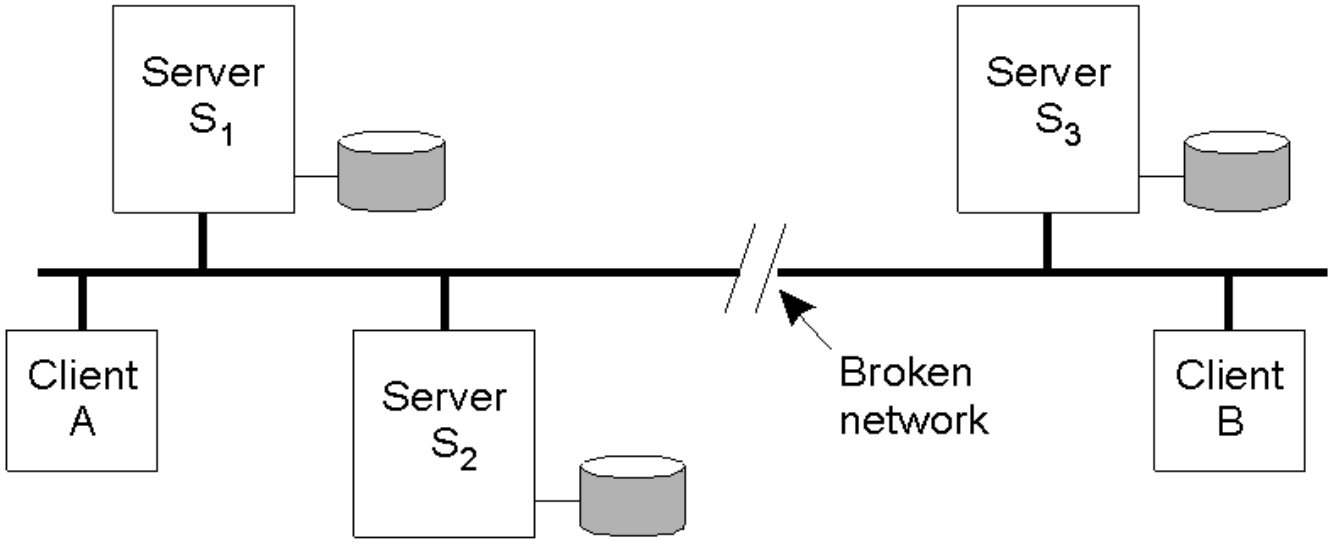
Example: Caching in Coda



Server Replication in Coda

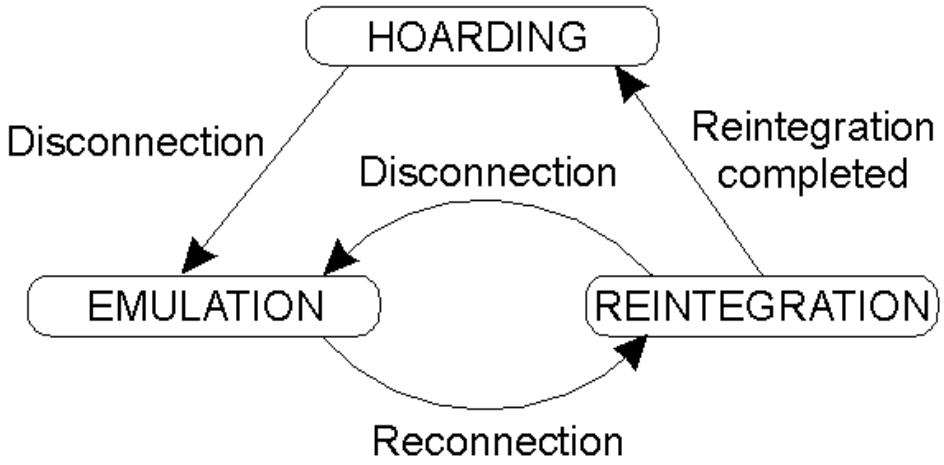
- Unit of replication: **volume**
- **Volume Storage Group** (VSG): set of servers that have a copy of a volume
- **Accessible Volume Storage Group** (AVSG): set of servers in VSG that the client can contact
- Use vector versioning
  - One entry for each server in VSG
  - When file updated, corresponding version in AVSG is updated

Server Replication in Coda



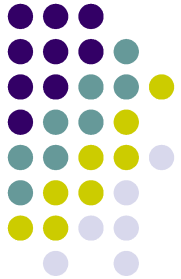
- Versioning vector when partition happens: [1,1,1]
- Client A updates file → versioning vector in its partition: [2,2,1]
- Client B updates file → versioning vector in its partition: [1,1,2]
- Partition repaired → compare versioning vectors: **conflict**!

Fault Tolerance in Coda



- HOARDING: File cache in advance with all files that will be accessed when disconnected
- EMULATION: when disconnected, behavior of server emulated at client
- REINTEGRATION: transfer updates to server; resolves conflicts

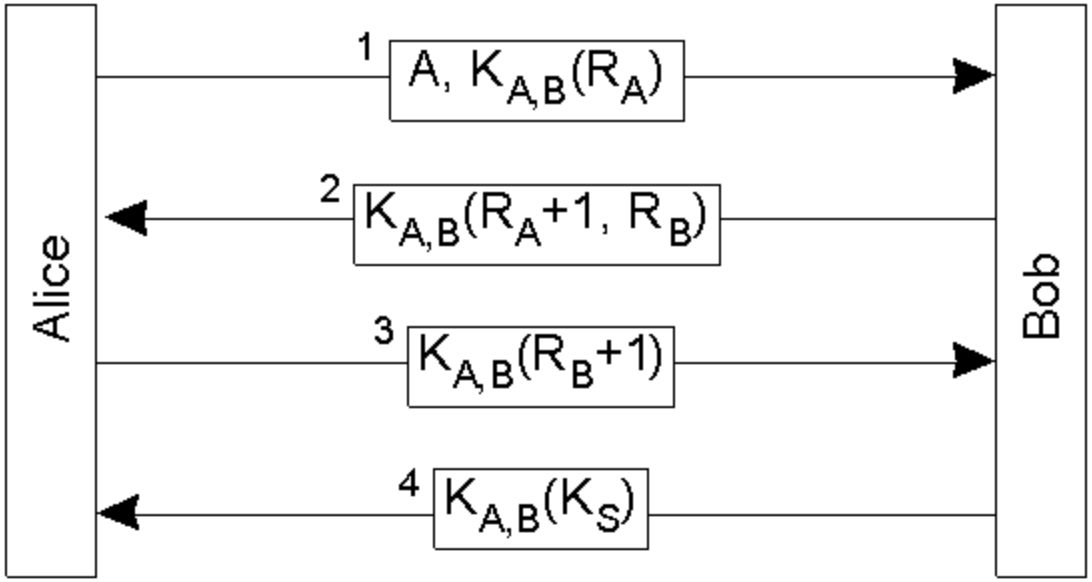
Security in Coda



- Set-up a secure channel between client and server
  - Use secure RPC
- System-level authentication

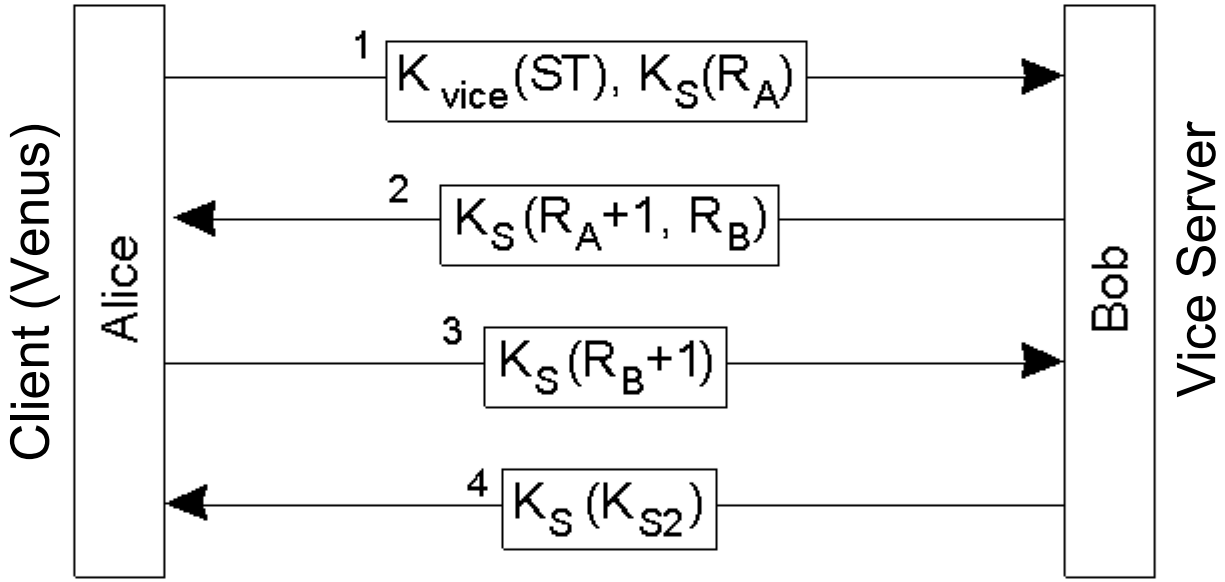
Security in Coda

- Mutual Authentication in RPC2
- Based on Needham-Schroeder protocol

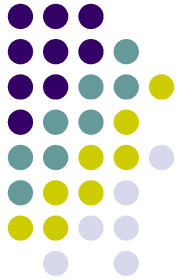


Establishing a Secure Channel

- Upon authentication AS (authentication server) returns:
  - Clear token: CT = [Alice, TID, K_S, T_{start}, T_{end}]
  - Secret token: ST = $K_{vice}([CT]^*_{Kvice})$
  - K_S: secret key obtained by client during login procedure
  - K_{vice}: secret key shared by vice servers
- Token is similar to the ticket in Kerberos

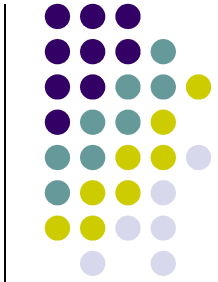


Summary of Coda File System



- High availability
- RPC communication
- Write back cache consistency
- Replication and caching
- Needham-Schroeder secure channels

Google File System



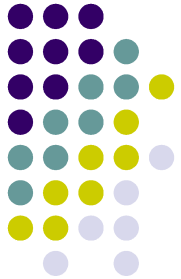
The Google File System

http://labs.google.com/papers/gfs.html

By: Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung

Appeared in: 19th ACM Symposium on Operating Systems Principles,
Lake George, NY, October, 2003.

Key Topics

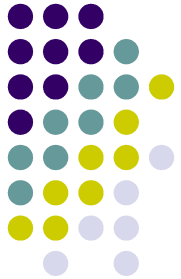


- Search Engine Basics
- Motivation
- Assumptions
- Architecture
- Implementation
- Measurements
- Conclusion

Google Search Engine

- Search engine performs many tasks including
  - Crawling
  - Indexing
  - Ranking
    - Maintain Web Graph, Page Rank
  - Deployment
    - Adding new data, update
    - Processing queries

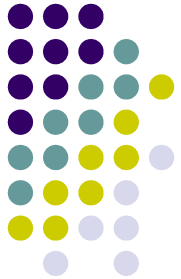
Google Search Engine



- Size of the web > 1 billion textual pages (2000)
- Google index has over 8 billion pages (2003)
- Google is indexing 40-80TB (2003)
- Index update frequently (~every 10 days) (2000)
- Google handles 250 million searches/day (2003)

How to manage this huge task, without going down????

Motivation



- Need for a scalable DFS
- Large distributed data-intensive applications
- High data processing needs
- Performance, Reliability, Scalability, Consistency and Availability
- More than traditional DFS

Assumptions – Environment

- System is build from inexpensive hardware
- Hardware failure is a norm rather than the exception
- Terabytes of storage space

- 15000+ commodity machines (2001)
- ~100 machines die each day (2001)

Assumptions – Applications

- Multi-GB files rather than billion of KB-sized files
- Workloads
  - Large streaming reads
  - Small random reads
  - Large, sequential writes that append data to file
  - Multiple clients concurrently append to one file
- High sustained bandwidth preferred over latency

Architecture

- Files are divided into fixed-size *chunks*
- Globally unique 64-bit *chunk handles*
- Fixed-size chunks (**64MB**)
- Chunks stored on local disks as Linux files
- For reliability each chuck replicated over *chunkservers,* called *replicas*

Why 64 MB chunk size?

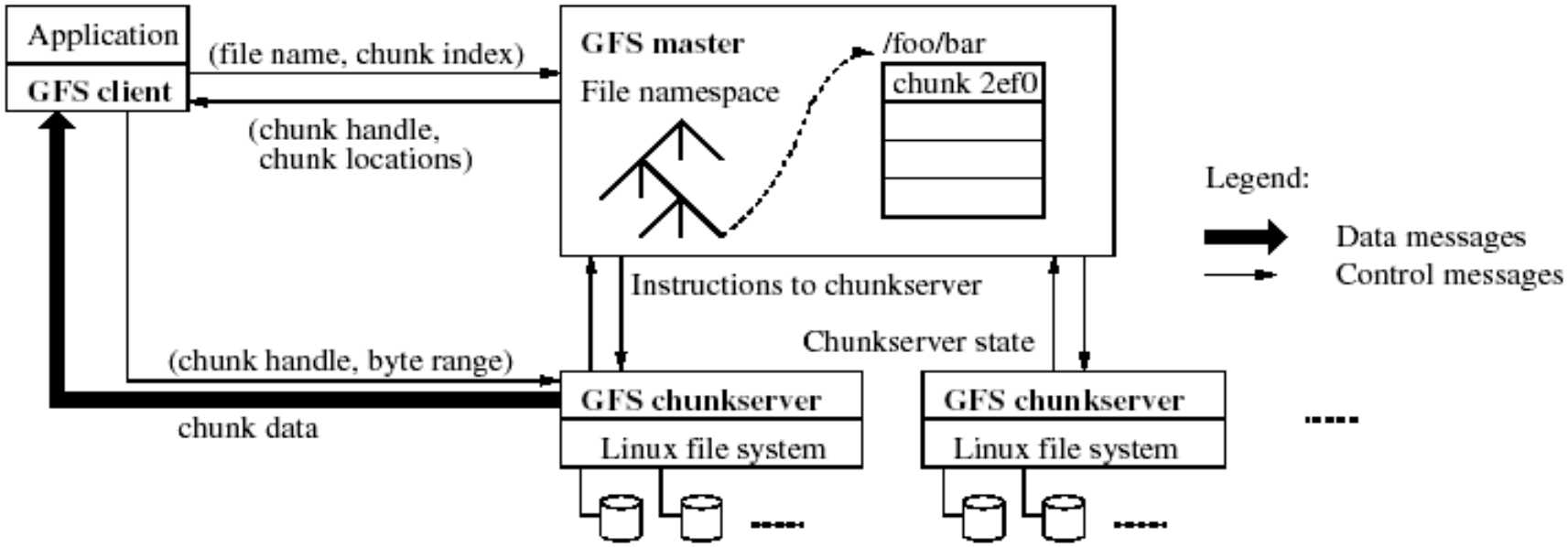
- Reduces need to interact with master server
- Target apps read/write large chunks of data at once, can maintain persistent TCP connection
- Larger chunk size implies less metadata

- Disadvantages:
  - Possible internal fragmentation
  - Small file may be one chunk, could cause chunkserver hotspots

Architecture

- Master server (*simplifies design)*:
  - Maintains all file system metadata
    - Namespace
    - access control info
    - file→chunk mappings
    - current location of chunks (which chunkserver)
  - Controls system-wide activities
    - Chunk lease management
    - Garbage collection of orphaned chunks
    - Chunk migration between servers
  - Communicates with chunkservers via “eartbeat” messages
    - Give slaves instructions & collect state info

Architecture



- Contact single *master*
- Obtain chunk locations
- Contact one of chunkservers
- Obtain data

Metadata

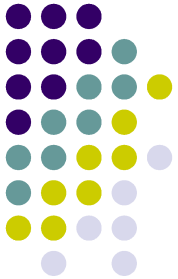
- Master stores 3 types of metadata:
  - File and chunk namespaces
  - Mapping from files to chunks
  - Location of chunk replicas

  - Metadata kept in memory
    - It's all about **speed**
    - 64 bytes of metadata per 64MB chunk
    - Namespaces compacted with prefix compression
  - First two types logged to disk: operation log
    - In case of failure & also keeps chunk versions (timestamps)
  - Last type probed at startup, from each chunkserver

Consistency Model

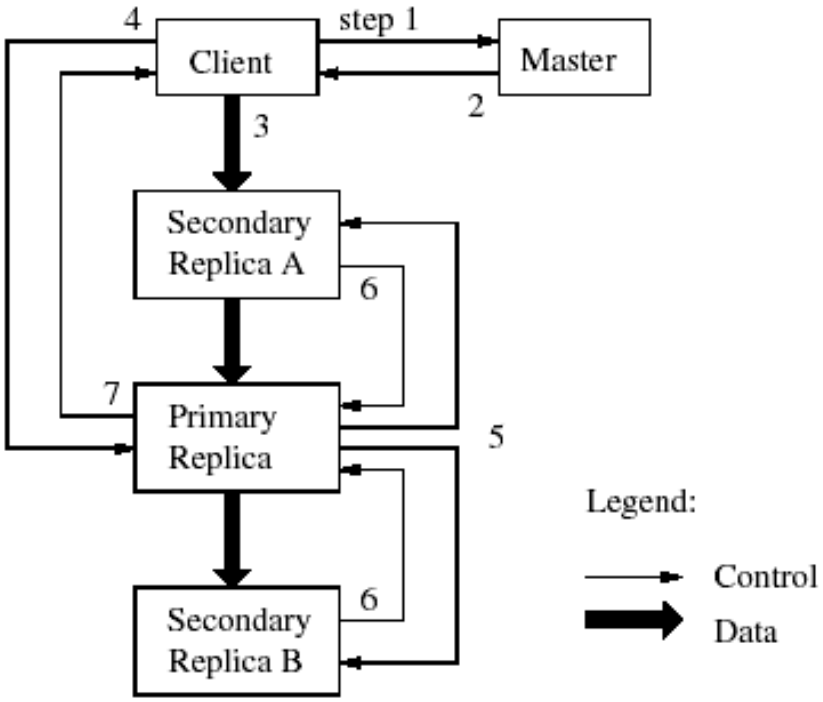
- Relaxed consistency model
- Two types of mutations
  - **Writes**
    - Cause data to be written at an application-specified file offset
  - **Record appends**
    - Operations that append data to a file
    - Cause data to be appended atomically at least once
    - Offset chosen by GFS, not by the client
- States of a file region after a mutation
  - **Consistent**
    - If all clients see the same data, regardless which replicas they read from
  - **Defined**
    - *Consistent* & all clients see what the mutation writes in its entirety
  - **Undefined**
    - *Consistent* but it may not reflect what any one mutation has written
  - **Inconsistent**
    - Clients see different data at different times

Leases and Mutation Order



- Master uses *leases* to maintain a consistent mutation order among replicas
- *Primary* is the chunkserver who is granted a chunk lease
- All others containing replicas are *secondaries*
- Primary defines a mutation order between mutations
- All *secondaries* follows this order

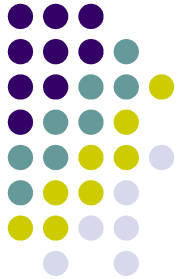
Implementation – Writes



Mutation Order

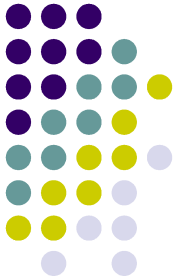
- *identical replicas*
- File region may end up containing *mingled fragments* from different clients (*consistent* but *undefined*)

Data flow



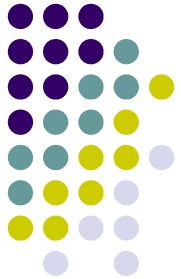
- Decoupled from control flow
  - to use the network efficiently
- Pipelined fashion
  - Data transfer is pipelined over TCP connections
  - Each machine forwards the data to the “closest” machine
- Benefits
  - Avoid bottle necks and minimize latency

Atomic Record Appends



- **The client specifies only the data**
- **Similar to writes**
  - Mutation order is determined by the *primary*
  - All *secondaries* use the same mutation order
- **GFS appends data to the file <u>at least once atomically</u>**
  - The chunk is padded if appending the record exceeds the maximum size → *padding*
  - If a record append fails at any replica, the client retries the operation → *record duplicates*
  - File region may be *defined* but interspersed with *inconsistent*

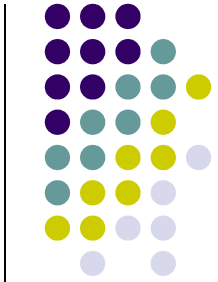
Snapshot



- **Goals**
  - To quickly create branch copies of huge data sets
  - To easily checkpoint the current state

- **Copy-on-write technique**
  - Metadata for the source file or directory tree is duplicated
  - Reference count for chunks are incremented
  - Chunks are copied later at the first write

Namespace Management and Locking



- Namespaces are represented as a lookup table mapping full pathnames to metadata
- Use locks over regions of the namespace to ensure proper serialization
- Each master operation acquires a set of locks before it runs

Example of Locking Mechanism

- Preventing /home/user/foo from being created while /home/user is being snapshotted to /save/user
  - Snapshot operation
    - Read locks on /home and /save
    - Write locks on /home/user and /save/user
  - File creation
    - Read locks on /home and /home/user
    - Write locks on /home/user/foo
  - Conflict locks on /home/user

  Note: Read lock is sufficient to protect the parent directory from deletion

Replica Operations

- Chunk Creation
  - New replicas on chunkservers with low disk space utilization
  - Limit number of recent creations on each chunkserver
  - Spread across many racks
- Re-replication
  - Prioritized: How far it is from its replication goal
  - The highest priority chunk is cloned first by copying the chunk data directly from an existing replica
- Rebalancing
  - Master rebalances replicas periodically

Garbage Collection

- Deleted files
  - Deletion operation is logged
  - File is renamed to a hidden name, then may be removed later or get recovered
- Orphaned chunks (unreachable chunks)
  - Identified and removed during a regular scan of the chunk namespace
- Stale replicas
    - Chunk version numbering

Fault Tolerance and Diagnosis

- High availability
  - Fast recovery
    - Master, chunk servers designed to restore state quickly
    - No distinction between normal/abnormal termination
  - Chunk replication
  - Master replication
    - State of master server is replicated (i.e. operation log)
    - External watchdog can change DNS over to replica if master fails
    - Additional “shadow” masters provide RO access during outage
      - Shadows may lag the primary master by fractions of 1s
      - Only thing that could lag is metadata, not a big deal
      - Depends on primary master for replica location updates

Fault Tolerance and Diagnosis

- Data Integrity
  - Chunkservers checksum to detect corruption
    - Corruption caused by disk failures, interruptions in r/w paths
    - Each server must checksum because chunks not byte-wise equal
      - Chunks are broken into 64 KB blocks
      - Each block has a 32 bit checksum
      - Checksums kept in memory and logged with metadata
      - Can overlap with IO since checksums all in memory
  - Client code attempts to align reads to checksum block boundaries
  - During idle periods, chunkservers can checksum inactive chunks to detect corrupted chunks that are rarely read
    - Prevents master from counting corrupted chunks towards threshold

Real World Clusters

- Cluster A:
  - Used regularly for R&D by 100+ engineers
  - Typical task reads through few MBs - few TBs, analyzes, then writes back
  - 342 chunkservers
  - 72 TB aggregate disk space
  - ~735,000 files in ~992,000 chunks
  - 13 GB metadata per chunkserver
  - 48 MB metadata on master
- Cluster B:
  - Used for production data processing
  - Longer tasks, process multi-TB datasets with little to no human intervention
  - 227 chunkservers
  - 180 TB aggregate disk space
  - ~737,000 files in ~1,550,000 chunks
  - 21 GB metadata per chunkserver
  - 60 MB metadata on master

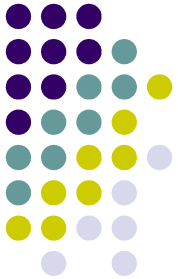
Measurements

- Read rates much higher than write rates
- Both clusters in heavy read activity
- Cluster A supports up to 750MB/read, B: 1300 MB/s
- Master was not a bottle neck

- Recovery time (of one chunkserver)
  - 15,000 chunks containing 600GB are restored in 23.2 minutes (replication rate ≅ 400MB/s)

Review

- High availability and component failure
  - Fault tolerance, Master/chunk replication, HeartBeat, Operation Log, Checkpointing, Fast recovery
- TBs of Space (100s of chunkservers, 1000s of disks)
- Networking (Clusters and racks)
- Scalability (single master, minimum interaction between master and chunkservers)
- Multi-GB files (64MB chunks)
- Sequential reads (Large chunks, cached metadata, load balancing)
- Appending writes (Atomic record appends)



References

- Andrew S. Tanenbaum, Maarten van Steen, Distributed System: *Principles and Paradigms*, Prentice Hall, 2002.
- Mullender, M. Satyanarayanan, Distributed Systems, *Distributed File Systems*, 1993.
- Peter J. Braam, The Coda File System, www.coda.cs.cmu.edu.
- S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03), Bolton Landing (Lake George), NY, Oct 2003.

Note: Images used in this presentation are from the textbook and are also available online.