

Distributed Systems

15. Distributed File Systems II

Paul Krzyzanowski
pxk@cs.rutgers.edu

Review

- NFS
 - RPC-based access
- AFS
 - Long-term caching
- CODA
 - Read/write replication & disconnected operation
- DFS
 - AFS + tokens for consistency and efficient caching
- SMB/CIFS
 - RPC-like access with strong consistency
 - Oplocks (tokens) to support caching

Microsoft Dfs

- "Distributed File System"
 - Provides a logical view of files & directories
 - Organize multiple SMB shares into one file system
 - Provide location transparency & redundancy
- Each computer hosts **volumes**
 - `\\servername\dfsname`

Each Dfs tree has one root volume and one level of leaf volumes.
- A volume can consist of multiple shares
 - Alternate path: load balancing (read-only)
 - Similar to Sun's automounter
- Dfs = SMB + naming/ability to mount server shares on other server shares

Redirection

- A share can be replicated (read-only) or moved through **Microsoft's Dfs**
- Client opens old location:
 - Receives `STATUS_DFS_PATH_NOT_COVERED`
 - Client requests referral: `TRANS2_DFS_GET_REFERRAL`
 - Server replies with new server

Clustered (Distributed) File Systems

Google File System (GFS)

GFS Goals

- Scalable distributed file system
- Designed for large data-intensive applications
- Fault-tolerant; runs on commodity hardware
- Delivers high performance to a large number of clients

Design Assumptions

- Assumptions for conventional file systems don't work
 - E.g., "most files are small", "lots have short lifetimes"
- For Google:
 - Component failures are the norm, not an exception
 - File system = thousands of storage machines; some % not working
 - Files are huge. Multi-GB and multi-TB files are the norm
 - It doesn't make sense to work with billions of *n*KB-sized files
 - I/O operations and block size choices are also affected

Design Assumptions

- File access:
 - Most files are appended, not overwritten
 - Random writes within a file are almost never done
 - Once created, files are mostly read; often sequentially
 - Workload is mostly:
 - Mostly reads: large streaming reads, small random reads
 - Large appends
 - Hundreds of processes may append to a file concurrently
- FS will store a modest number of files for its scale
 - approx. a few million
- Designing the FS API with the design of apps benefits the system
 - Apps can handle a relaxed consistency model

File System Interface

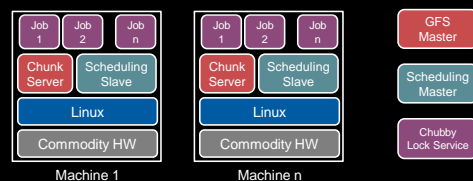
- GFS does not have a standard OS-level API
 - No POSIX API
 - No kernel/VFS implementation
 - User-level API
 - GFS is implemented in user space using native Linux FS
- Files organized hierarchically in directories
- Operations
 - Basic operations
 - *Create, delete, open, close, read, write*
 - Additional operations
 - *Snapshot*: create a copy of a file or directory tree at low cost
 - *Append*: allow multiple clients to append atomically without locking

GFS Master & Chunkservers

- GFS cluster
 - Multiple **chunkservers**
 - Data storage: fixed-size chunks
 - Chunks replicated on several systems
 - One **master**
 - File system metadata
 - Mapping of files to chunks

Core Part of Google Cluster Environment

- Google Cluster Environment
 - Core services: GFS + cluster scheduling system
 - Typically 100s to 1000s of active jobs
 - 200+ clusters, many with 1000s of machines
 - Pools of 1000s of clients
 - 4+ PB filesystems, 40 GB/s read/write loads



Chunks and Chunkservers

- Chunk size = 64 MB
- Chunkserver stores a 32-bit checksum with each chunk
 - In memory & logged
- **Chunk Handle**
 - Globally unique 64-bit number
 - Assigned by the master when the chunk is created
- Chunkservers store chunks on local disks as Linux files
- Each chunk is replicated on multiple chunkservers
 - Three replicas (different levels can be specified)
 - Popular files may need more replicas to avoid hotspots

Master

- Maintains all file system metadata
 - Namespace
 - Access control info
 - Filename to chunks mappings
 - Current locations of chunks
- Manages
 - Chunk leases (locks)
 - Garbage collection (freeing unused chunks)
 - Chunk migration (copying/moving chunks)
- Master replicates its data for fault tolerance
- Periodically communicates with all chunkservers
 - Via heartbeat messages
 - To get state and send commands

Client Interaction Model

- GFS client code linked into each app
 - No OS-level API
 - Interacts with master for metadata-related operations
 - Interacts directly with chunkservers for data
 - All reads & writes go directly to chunkservers
 - Master is not a point of congestion
- Neither clients nor chunkservers cache data
 - Except for the system buffer cache
- Clients cache metadata
 - E.g., location of a file's chunks

One master = simplified design

- All metadata stored in master's memory
 - Super-fast access
- Namespaces and name-to-chunk maps
 - Stored in memory
 - Also persist in an *operation log* on the disk
 - Replicated onto remote machines for backup
- **Operation log**
 - similar to a journal
 - All operations are logged
 - Periodic checkpoints (stored in a B-tree) to avoid playing back entire log
- Master does not store chunk locations persistently
 - This is queried from all the chunkservers: avoids consistency problems

Large Chunks

- Default size = 64MB
- Reduces need for frequent communication with master to get chunk location info
- Clients can easily cache info to refer to all data of large files
 - Cached data has timeouts to reduce possibility of reading stale data
- Large chunk makes it feasible to keep a TCP connection open to a chunkserver for an extended time
- Master stores <64 bytes of metadata for each 64MB chunk

Reading Files

- Contact the master
- Get file's metadata: chunk handles
- Get the location of each of the chunk handles
 - Multiple replicated chunkservers per chunk
- Contact any available chunkserver for chunk data

Writing to files

- Less frequent than reading
- Master grants a *chunk lease* to one of the replicas
 - This replica will be the *primary replica* chunkserver
 - Primary can request extensions, if needed
 - Master increases the chunk version number and informs replicas

Writing to files

- Sending data
 - A client is given a list of replicas
 - Identifying the primary and secondaries
 - Client writes to the closest replica chunkserver
 - Replica forwards the data to another replica chunkserver
 - That chunkserver forwards to another replica chunkserver
 - Chunkservers store this data in a cache
- Writing data
 - Client waits for replicas to acknowledge receiving the data
 - Then send a *write* request to the primary, identifying the data that was sent
 - The primary is responsible for serialization of writes
 - Assigns consecutive serial numbers to all writes that it received
 - Applies writes in serial-number order and forwards write requests in order to secondaries
 - Once all acknowledgements have been received, the primary acknowledges the client

Writing to files

- Note: Data Flow is different from Control Flow
- Data Flow:
 - Client to chunkserver to chunkserver to chunkserver...
 - Order does not matter
- Control Flow (*write*):
 - Client to primary to all secondaries
 - Order maintained
- Chunk version numbers are used to detect if any replica has stale data (not updated because it was down)

Namespace

- No per-directory data structure like most file systems
 - E.g., directory file contains names of all files in the directory
- No aliases (hard or symbolic links)
- Namespace is a single lookup table
 - Maps pathnames to metadata

Apache HDFS

HDFS: Hadoop Distributed File System

- Primary storage system for Hadoop applications
- Hadoop
 - Software library – framework that allows for the distributed processing of large data sets across clusters of computers
- Hadoop includes:
 - MapReduce™: software framework for distributed processing of large data sets on compute clusters.
 - Avro™: A data serialization system.
 - Cassandra™: A scalable multi-master database with no single points of failure.
 - Chukwa™: A data collection system for managing large distributed systems.
 - HBase™: A scalable, distributed database that supports structured data storage for large tables.
 - Hive™: A data warehouse infrastructure that provides data summarization and ad hoc querying.
 - Mahout™: A Scalable machine learning and data mining library.
 - Pig™: A high-level data-flow language and execution framework for parallel computation.
 - ZooKeeper™: A high-performance coordination service for distributed applications.

HDFS Design Goals & Assumptions

- Run on commodity hardware
- Highly fault tolerant
 - Hardware failure is the norm
- High throughput
 - Designed for large data sets
- OK to relax some POSIX requirements
- Large scale
 - Instance of HDFS may comprise 1000s of servers
 - Each server stores part of the file system's data

HDFS Design Goals & Assumptions

- Applications need streaming access to their data
- Not general-purpose apps that run on general-purpose file systems
- Design primarily for batch processing rather than interactive use
 - Design for high throughput rather than low latency
- Typical file size is gigabytes to terabytes

HDFS Design Goals & Assumptions

- Write-once, read-many file access model
- A file's contents will not change
 - Simplifies data coherency
 - Suitable for web crawlers and MapReduce applications

HDFS Architecture

- Written in Java
- Master/Slave architecture
- Single **NameNode**
 - Master server responsible for the namespace & access control
- Multiple **DataNodes**
 - Responsible for managing storage attached to its node
- A file is split into one or more blocks
 - Typical block size = 64 MB
 - Blocks are stored in a set of DataNodes

NameNode

- Executes metadata operations
 - *Open, close, rename*
 - Maps file blocks to DataNodes
 - Maintains HDFS namespace
 - Uses a **transaction log (EditLog)** to record every change that occurs to file system metadata
 - Entire file system namespace + file-block mappings is stored in memory
 - ... and stored in a file (FsImage) for persistence
- NameNode receives a periodic Heartbeat and Blockreport from each DataNode
 - Heartbeat = "I am alive" message
 - Blockreport = list of all blocks on a datanode

DataNode

- Responsible for serving read/write requests
- Blocks are replicated for fault tolerance
 - App can specify # replicas at creation time
 - Can be changed later
- Blocks are stored in the local file system at the DataNode

Reads & Replica Selection

- Client sends request to NameNode
 - Receives list of blocks and replica DataNodes per block
- Client tries to read from the closest replica closest
 - Prefer same rack
 - Else same data center
 - Location awareness is configured by the admin

Writes

- Client caches file data into a temp file
- When temp file \geq one HDFS block size
 - Client contacts NameNode
 - NameNode inserts file name into file system hierarchy & allocates a data block
 - Responds to client with the destination data block
 - Client writes to the block at the corresponding DataNode

 - When a file is closed, remaining data is transferred to a DataNode
 - NameNode is informed that the file is closed
 - NameNode commits file creation operation into a persistent store (log)
- Data writes are chained
 - Client writes to the first (closest) DataNode
 - That DataNode writes the data stream to the second DataNode
 - And so on...

The End