CSCI 5673
Distributed Systems

Lecture Set Ten

Big Data Processing on Commodity Clusters

Lecture Notes by
Shivakant Mishra
Computer Science, CU Boulder
Last update: March 07, 2017

Big Data Processing in Commodity Clusters

- Large data size: order of TBs
- Varying characteristics: bulk, streaming, realtime, graph, …
- Study the following
  - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. SOSP 2003.
  - Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. ODSI 2004.
  - Hadoop: *http://hadoop.apache.org/*
  - Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI 2012.
  - Storm paper

Google File System

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System", SOSP 2003

Motivation

Google needed a good distributed file system

•Redundant storage of massive amounts of data on cheap and unreliable computers

Why not use an existing file system?

•Google's problems are different from anyone else'
  - Different workload and design priorities

•GFS is designed for Google apps and workloads

•Google apps are designed for GFS

Google File System (GFS)

- A scalable distributed file system for large distributed data-intensive applications
  - fault tolerance while running on inexpensive commodity hardware
  - high aggregate performance to a large number of clients

- Widely deployed within Google – multiple GFS clusters
  - As the storage platform for the generation and processing of data used by our service
  - Also R&D efforts requiring large data sets

GFS

- Observations:
  - 100s to 1000s of inexpensive commodity hardware and software => frequent failures!
  - Multi-GB files are common for Google
    - Each file contains many application objects like Web documents – Google search needs to archive the Web!
    - Billions of such objects, and TBs of data
    - *Don't want to manipulate at KB-level*

GFS

- Observations:
  - Most files modified via append, not random write
    - Appending becomes the focus of performance optimization and atomicity guarantees
    - Small writes are supported but not efficiently
  - Most files read sequentially in large batches
    - Random read access occurs in small KB chunks

GFS

- Design goals:
  - Fault tolerant over commodity hardware/software
  - Built for a few million large files (100 MB+)
  - Optimized for large sequential writes, i.e. appends
  - Optimized for large sequential reads and small random reads: files seldom modified
  - Atomic append to synchronize 100s of clients wanting to concurrently append to same file
  - Throughput more important than latency

GFS

- Supports usual file operations:
  - Read, write, open, close, create, delete
  - Also *record append* (atomic append with multiple clients)
  - Also *snapshot* operation to take a quick copy of a file or directory tree
- Files organized hierarchically in directory trees

GFS Architecture

- GFS cluster: A single *master* and many (100s-1000s) of *chunkservers*
  - Files divided into fixed-size 64 MB chunks, identified by 64-bit, globally unique *handle*

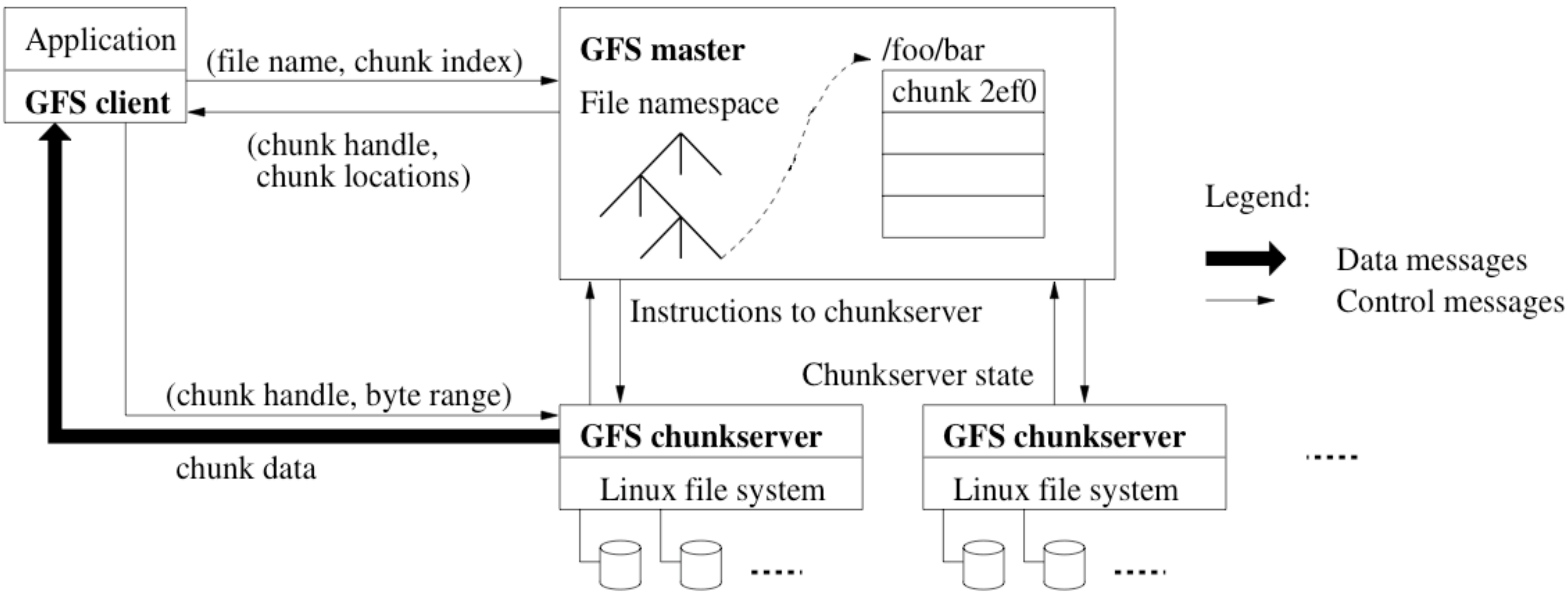


Figure 1: GFS Architecture (Read example)

GFS Architecture

- The master and chunkservers are just Linux boxes
  - Chunks are just stored as Linux files, but bypasses vnode layer since POSIX API not supported
- Chunks are replicated three times: Chunk size: 64 MB
- Read/write by specifying the chunk handle and byte range
- Master maintains all file metadata, coordinates chunkservers
  - Queries chunkservers with heartbeat

GFS Architecture

- Minimal caching
  - No caching of file data at client or chunkserver, other than standard Linux buffer caching
    - Offer little benefit because most applications stream through huge files or have working sets too large to be cached.
    - Simplifies file system design, don't have to worry about cache coherence issues
  - Some caching of file metadata at client

GFS Architecture: Chunk size

- 64 MB
- Large chunk size
  - Reduces interaction with the master
  - More operation on a single chunk: reduce network overhead
  - Reduces the metadata size

GFS Single Master

- Tells the client the chunk handle and where replicas are located
  - Client then fetches/writes the data from/to the chunkserver directly, bypassing the master
  - Client chooses “most likely the closest” replica (based on IP)
    - Cuts down on network traffic in case replica is on another rack
    - If they’re all on the same rack, likely no big diff

GFS Single Master

- *The Master is not a bottleneck*
  - Optimized for large streamed reads and writes
  - File manager is not in the way for large I/O

- Some other performance optimizations:
  - Client can ask for multiple chunks in one request
  - Server can reply with multiple "next" chunks

GFS Single Master

- Stores file metadata:
  - the file and chunk namespaces,
  - the mapping from files to chunks, and
  - the locations of each chunk's replicas.
- All metadata is stored in memory for fast access
- Some metadata is backed up
  - Namespaces and file-to-chunk mapping is stored in an operation log
  - Replica locations are not persisted, but are queried from chunkservers at startup and regularly thereafter

GFS Single Master

- Delegates replica consistency management
- Garbage collects orphaned chunks
- Migrates chunks between chunkservers (load balancing)

GFS Fault Tolerance

- File metadata is stored in an operation log
  - Operation log: historical record of critical metadata changes
  - Any changes to metadata (directory, chunks) are logged as transactions before committing
  - Replay the log on master failure from latest checkpoint to recover entire file system
  - The operation log is replicated as well

GFS Fault Tolerance

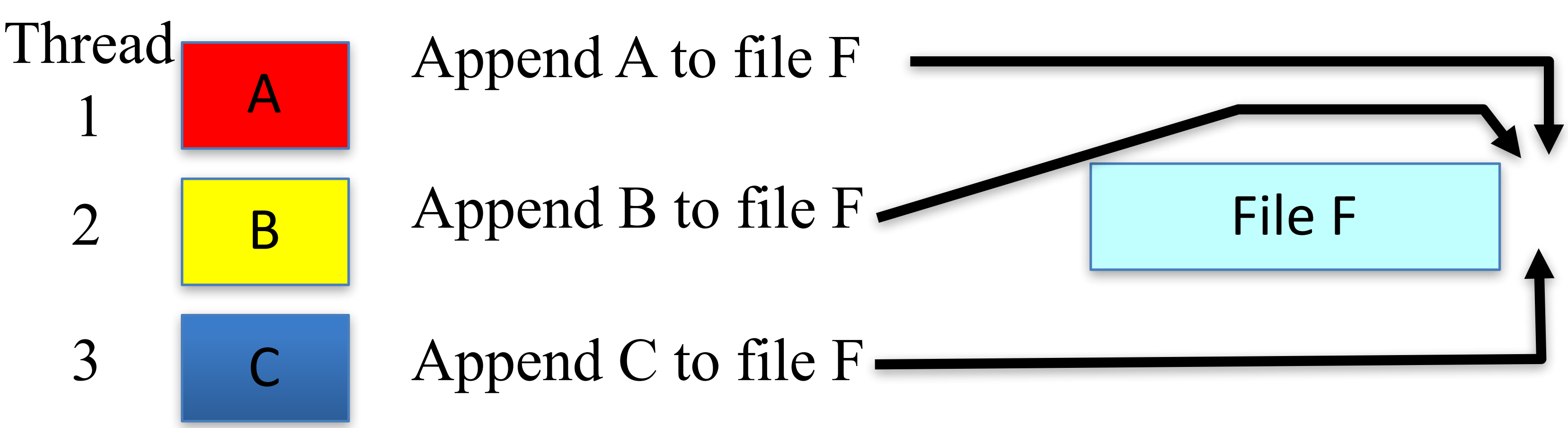
- Master server periodically queries each chunkserver
  - If the chunkserver is down, then it does not return that replica to a client's file request
  - Stale replicas (failed and missed a mutation) are also detected by looking at the chunk's version #
    - The chunk # is incremented for each new lease, and is persisted to all active replicas. A down replica will miss this update, will be stale
  - Stale replicas are garbage-collected

GFS Fault Tolerance

- Master server periodically queries each chunkserver (cont.)
  - If a new replica needs to be made, then re-replicate by copying highest priority chunk to chunkserver
    - Higher priority for chunks that have lost more replicas, and that are live
    - Placing new replica – see later slide
- Fast recovery
  - "Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated."

GFS Record Append

- Many producers want to append to the end of a long file
  - Relaxed consistency: GFS guarantees the data is appended at least once, at a location of GFS' choosing
  - Appending is atomic as one continuous sequence of bytes



GFS Record Append

- Apps should use append instead of write(offset)
  - could become inconsistent?
- All file mutations are executed in the same order on each replica
  - use *leases* to maintain a consistent mutation order across replicas.
    - The master grants a chunk lease to one of the replicas, which we call the *primary*.
    - The primary picks a serial order for all mutations to the chunk.
    - All replicas follow this order when applying mutations

GFS Record Append

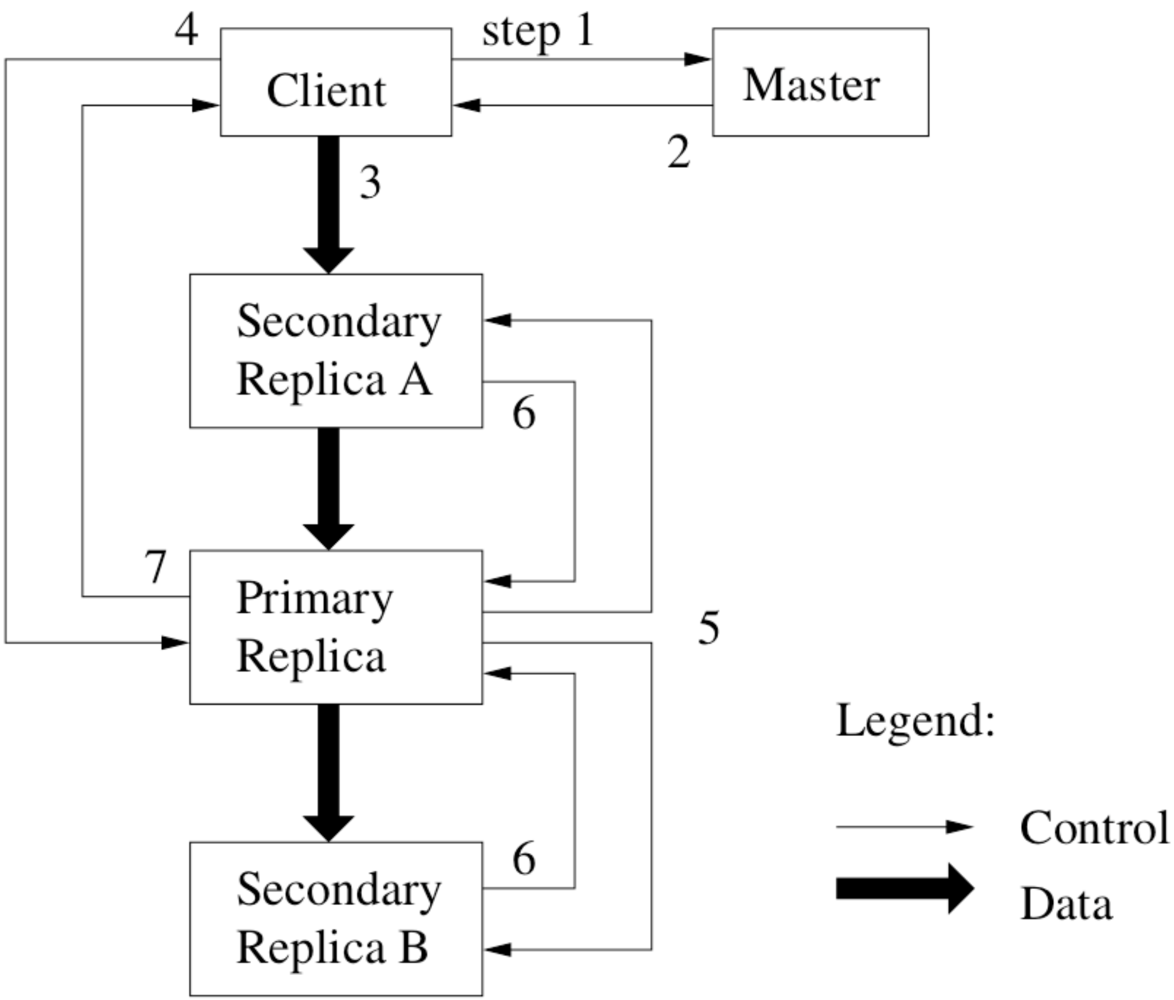


Figure 2: Write Control and Data Flow

1. Client asks for primary chunkserver and replicas
2. Client caches response
3. Client pushes data to all replicas in any order
4. After all replicas have ACKed receiving the data, client sends write request to primary
5. Primary chooses order of mutations, sends to secondary replicas
6. Secondaries reply
7. Returns success or failure (if so, retries)

GFS Record Append

- Pushes most of the I/O work into the array of replicated servers, and away from the master
  - Data flow and control flow are decoupled, thus keeping bandwidth and hosts maximally busy
- Follows control flow of file mutations with a little extra logic

GFS Record Append

- Maintains append-at-least-once semantics
  - i.e. success if returned at step 7 only if every replica has written the append at least once
  - "*GFS does not guarantee that all replicas are bytewise identical*. It only guarantees that the data is written at least once as an atomic unit."
  - So every record is in every replica at least once, though not necessary same byte offset
  - But is order the same across all replicas?

Replica Placement

- Spread replicas not just across machines, but also across racks
- Factors affecting placement include:
  - Higher network bandwidth for closely clustered machines, less for further away
  - But better fault tolerance for further away machines
  - Lightly loaded machines are candidates for replicas, on the theory that they'll soon be swamped with writes. This spreads the load.

Re-Balancing/Migration

- The Master *rebalances* replicas periodically
  - moves replicas for better disk space and load balancing.
  - The placement criteria for the new replica same as before
  - chooses an existing replica to remove.
    - prefers to remove those on chunkservers with below-average free space so as to equalize disk space usage.
  - Also gradually fills up a new chunkserver rather than instantly swamps it with new chunks and heavy write traffic

Other GFS topics

- Snapshots
- Garbage collection
- Namespace locking
- Data Integrity
- Performance results (see paper)

Contributions of GFS?

- Highly scalable to TBs of data, thousands of servers,
- Highly fault tolerant
- Well optimized to their workload
- Add more from class discussion here…

Limitations of GFS?

- Not general purpose enough
  - Most file systems have to deal with small files < 64 MB
  - Most file systems have to deal with random writes and reads often
- Relaxed consistency might introduce problems?
- Security?
- Different/changing workloads?