Cloud Computing

Storage Systems

Eva Kalyvianaki
ek264@cam.ac.uk

Contents

- **The Google File System** SOSP 2003
  - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
  - https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

- **Bigtable: A Distributed Storage System for Structured Data** OSDI 2006
  - Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
  - https://storage.googleapis.com/pub-tools-public-publication-data/pdf/68a74a85e1662fe02ff3967497f31fda7f32225c.pdf

Requirements of cloud applications

- Most cloud applications are data-intensive and test the limitations of the existing infrastructure. Requirements:
  - Rapid application development and short-time to the market
  - Low latency
  - Scalability
  - High availability
  - Consistent view of the data

- These requirements cannot be satisfied simultaneously by existing database models; e.g., relational databases are easy to use for application development but do not scale well

Google File System (GFS) Motivation

- GFS ➔ developed in the late 1990s; uses thousands of storage systems built from inexpensive commodity components to provide petabytes of storage to a large user community with diverse needs
- Motivation
  1. Component failures is the norm
     - Appl./OS bugs, human errors, failures of disks, power supplies, …
  2. Files are huge (muti-GB to -TB files)
  3. The most common operation is to append to an existing file; random write operations to a file are extremely infrequent. Sequential read operations are the norm
  4. The consistency model should be relaxed to simplify the system implementation but without placing an additional burden on the application developers

GFS Assumptions

- The system is built from inexpensive commodity components that often fail.
- The system stores a modest number of large files.
- The workload consists mostly of two kinds of reads: large streaming reads and small random reads.
- The workloads also have many large sequential writes that append data to files.
- The system must implement well-defined semantics for many clients simultaneously appending to the same file.
- High sustained bw is more important than low latency.

GFS API

- It provides a familiar interface, though not POSIX.
- Supports: create, delete, open, close, read and write

- Plus: snapshot and record append
- snapshot

  creates a file copy or a directory tree at a low cost
- record append

  allows multiple clients to append data to the same file concurrently while guaranteeing atomicity.

The Architecture of a GFS Cluster

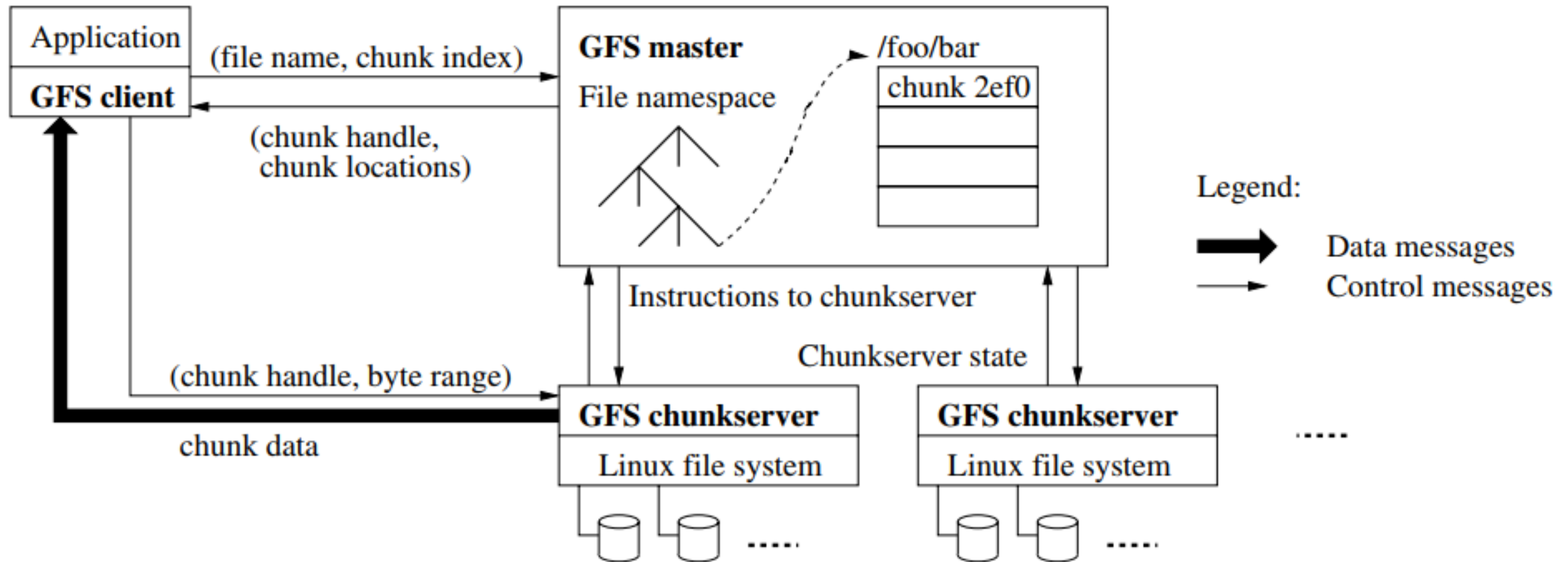


Figure 1: GFS Architecture

The Architecture of a GFS Cluster

- Single master, multiple chunkservers and clients, running on Linux machines.
- Fixed-size chunks, 64-bit unique and immutable chunk handle.
- Chunks are stored on local disks on chunkservers, three replicas.
- Master maintains all file system metadata: access control, mapping from files to chunks, chunks locations, etc.
- GFS client code implements the fs API and communicates with master and chunkservers to read/write data for applications.
- No caching by the client or the chunkservers.

- Single master??? Is this a good idea?
  - Simple design, masters makes more sophisticated chunk placement and replication decisions using global knowledge.

GFS – Design Decisions

- Segment a file in large chunks
- Implement an atomic file append operation allowing multiple applications operating concurrently to append to the same file
- Build the cluster around a high-bandwidth rather than low-latency interconnection network. Separate the flow of control from the data flow. Exploit network topology by sending data to the closest node in the network.
- Eliminate caching at the client site. Caching increases the overhead for maintaining consistency among cashed copies
- Ensure consistency by channeling critical file operations through a master, a component of the cluster which controls the entire system
- Minimise the involvement of the master in file access operations to avoid hot-spot contention and to ensure scalability
- Support efficient checkpointing and fast recovery mechanisms
- Support an efficient garbage collection mechanism

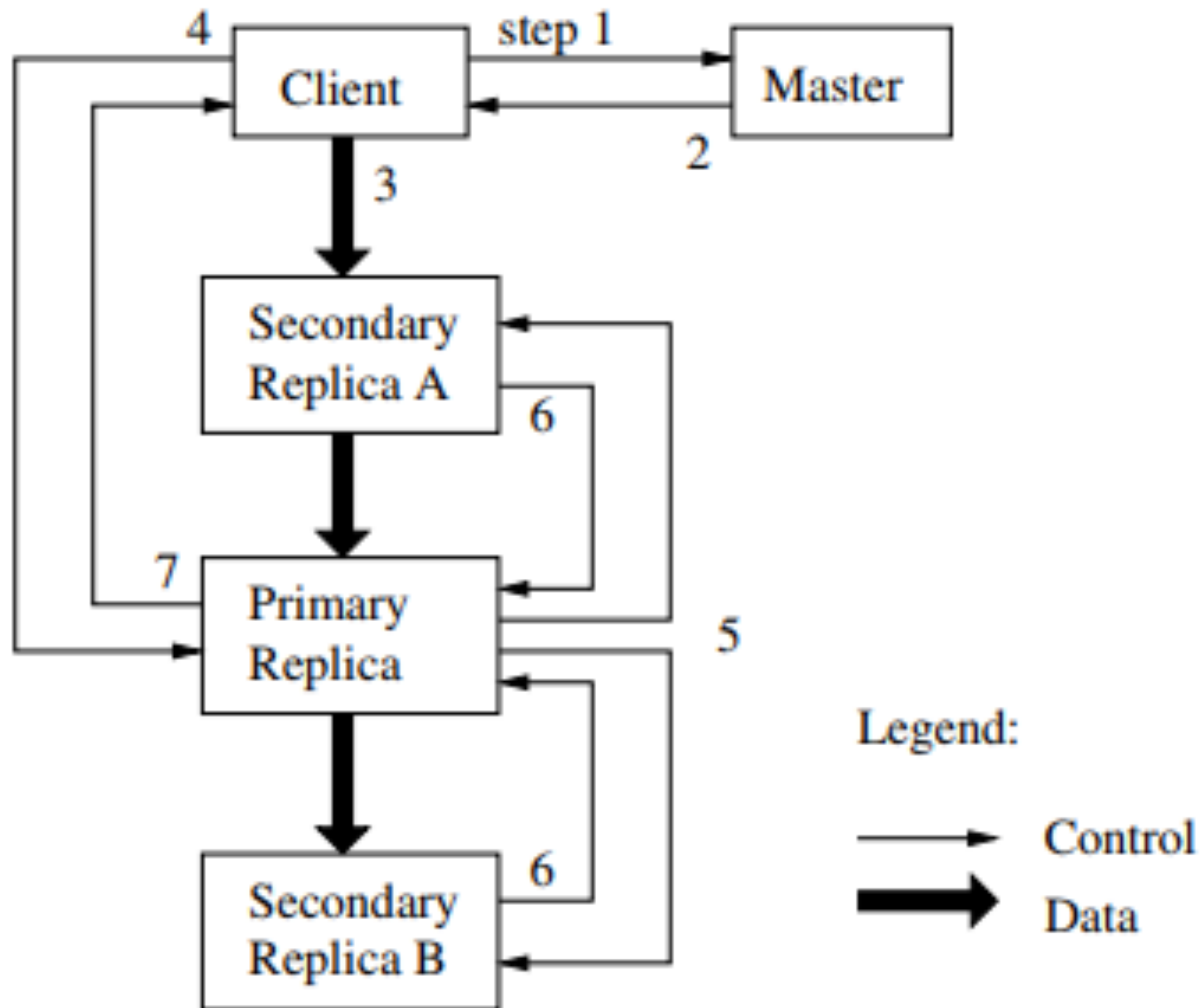
GFS Chunks

- GFS files are collections of fixed-size segments called chunks
- The chunk size is 64 MB; this choice is motivated by the desire to optimise the performance for large files and to reduce the amount of metadata maintained by the system
- A large chunk size increases the likelihood that multiple operations will be directed to the same chunk thus, it reduces the number of requests to locate the chunk and, it allows the application to maintain a persistent TCP network connection with the server where the chunk is located
- Large chunk size reduces the size of metadata stored on the master
- A chunk consists of 64 KB blocks
- Problem with small files of small number of chunks → hot spots → increase the replication factor

Consistency Model

- Mutations are writes or record appends
- Each mutation is performed at all chunk's replicas.
- Use of leases for consistent mutation order:
  - Master grants a chunk lease to one of the replicas, *primary*
  - The primary picks a serial order of all mutations to the chunk
  - All replicas follow this order when applying mutations
  - Global mutation order is defined by:
  1. The lease grant order chosen by the master, and
  2. Within a lease by the serial numbers assigned by the primary.
- Leases are initially 60 secs
- If the masters looses the primary, it grants a new lease to another replica after the old lease expires.

Write Control and Data Flow



Atomic Record Appends

- Client specifies only the data
- GFS appends it to the file at an offset at GFS's choosing and returns the offset to the client
- Primary checks if appending would cause the chunk to exceed the maximum size, if so:
  1. Pads the chunk to the maximum size, and
  2. Indicates client to retry on the next chunk

Master Operation

Namespace and Locking

- Each master operation acquires a set of locks before it runs
- Allows concurrent mutations in the same directory
- Locks are acquired in a consistent total order to prevent deadlocks

Replica Management

- Chunks replicas are spread across racks
- Traffic for a chunk exploits the aggregate bw of multiple racks.
- New chunks are placed on servers with low disk-space-utilisation, with few "recent" creations, and across racks
- Re-replication once the no of available replicas is below the goal
- Master rebalances replicas periodically for better disk space and load balancing

Conclusions

- Component failures are the norm
- System optimised for huge files that are mostly appended and then read
- Fault-tolerance is achieved by constant monitoring, replicating crucial data and automatic recovery, chunk replication, checksumming to detect data corruption
- High-aggregate throughput by separating file system control from data transfer. Master involvement in common operation is minimised by a large chunk size and chunk leases → a centralised master is not a bottleneck

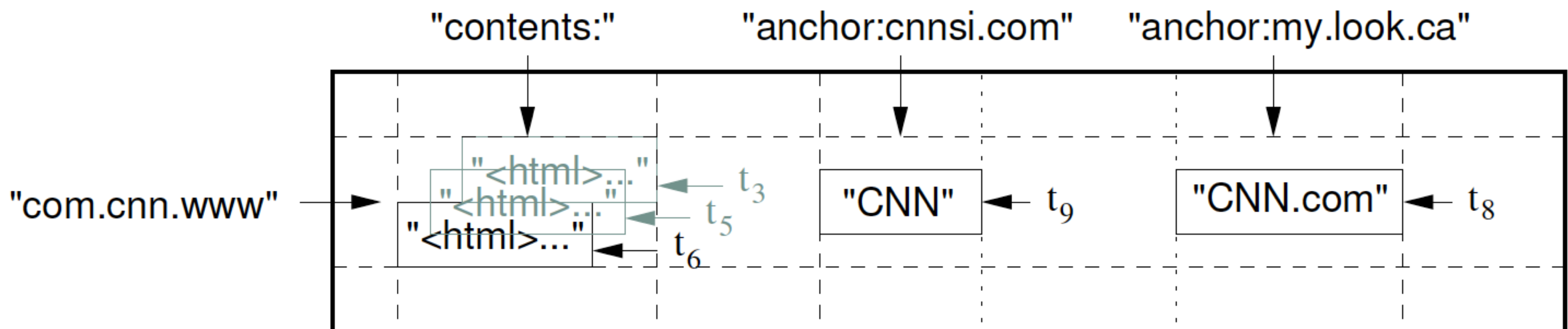
Bigtable: A Distributed Storage System for Structured Data

- Bigtable: a **distributed** storage for **structured data** designed to **scale big**, petabytes of data and thousands of machines.
- Used by many Google products:
  - Google Earth, Google Analytics, web indexing, …
- Handles diverse workload:
  - Throughput-oriented batch-processing
  - Latency-sensitive apps to end users
- Clients can control locality and whether to server their data from memory or disk

Data Model

- "A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map."

  (row:string, column:string, time:int64) → string



Tablets

- Data is maintained in lexicographic order by row key.
- The row range of a table can be dynamically partitioned.
- Each range is called a **tablet**. The unit of distribution.

- Nearby rows will be served by the same server
- Good locality properties by properly selecting the row keys

Building Blocks

- GFS stores logs and data files
- Bigtable clusters runs on a shared pool of machines (co-location).
- It depends on a cluster management system for scheduling jobs
- The Google SSTable file format is used to store Bigtable data
  - SSTable: a persistent, ordered immutable map from keys to values
  - It contains a sequence of 64KB blocks of data
  - A block index to locate blocks; lookups with a single disk seek, find the block from the in-memory index (loaded in mem when SSTable is opened) and then getting the block from disk.
- Bigtable uses the Chubby persistent distributed lock service to:
  - Ensure that there is at most one active master at any time,
  - Store the bootstrap location of Bigtable data,
  - Store Bigtable schema, …
- Chubby uses Paxos to ensure consistency

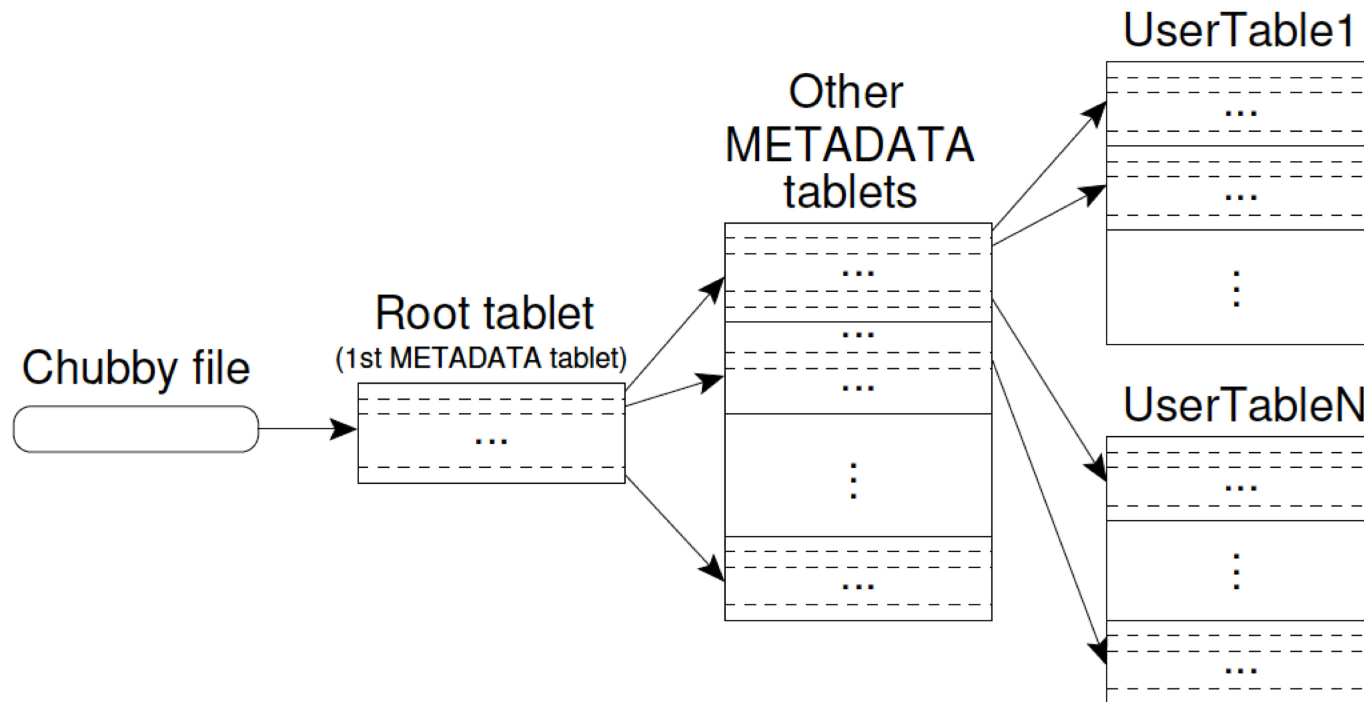
Implementation

- Three major components:
  1. A library linked into every client
  2. One master server
  3. Multiple tablet servers
- Master server: assigns tablets to table servers, adds and monitors tablet servers, balances tablet-server load, …
- Each tablet server: manages a set of tables, handles reads/writes to its tablets, splits too large tablets.
- Clients communicate directly with tablet servers for reads/writes. Bigtable clients do not rely on the master for tablet location → lightly loaded master

- Bigtable cluster stores a number of tables → a table consists of a set of tables → each table has data related to a row range
- At first a table has one tablet then splits into more tablets

100-200MB

Table Location



Addresses 2^{34} tablets

Table Assignment

- Each tablet is assigned to one tablet server at-a-time.
- Master keeps track of live tablet servers, current assignments, and unassigned tablets
- Upon a master starting
  - Acquires master lock in Chubby
  - Scans live tablet servers
  - Gets list of tablets from each tablet server, to find out assigned tablets
  - Learns set of existing tablets → adds unassigned tablets to list

Table Serving