Google File System

- **GFS** (**G**oogle **F**ile **S**ystem) is the **distributed file system** used by most Google services
  - Driver in development was managing the Google Web search index
  - **Applications may use GFS directly**
    - The database Bigtable is an application that was especially designed to run on-top of GFS
  - GFS itself runs on-top of standard POSIX-compliant Linux file systems
- **Hadoop's file system (HDFS)** was coded inspired by GFS papers, only open source…

GFS

- **Design constraints and considerations**
  - Run on potentially unreliable **commodity hardware**
  - **Files are large** (usually ranging from 100 MB to multiple GBs of size)
    - e.g. satellite imaginary, or a Bigtable file
  - Billions of files need to be stored
  - Most **write operations** are **appends**
    - **Random** writes or updates are rare
    - Most files are write-once, read-many (WORM)
    - Appends are much more **resilient** in distributed environments than random updates
    - Most Google applications rely on **Map and Reduce** which naturally results in file appends

GFS

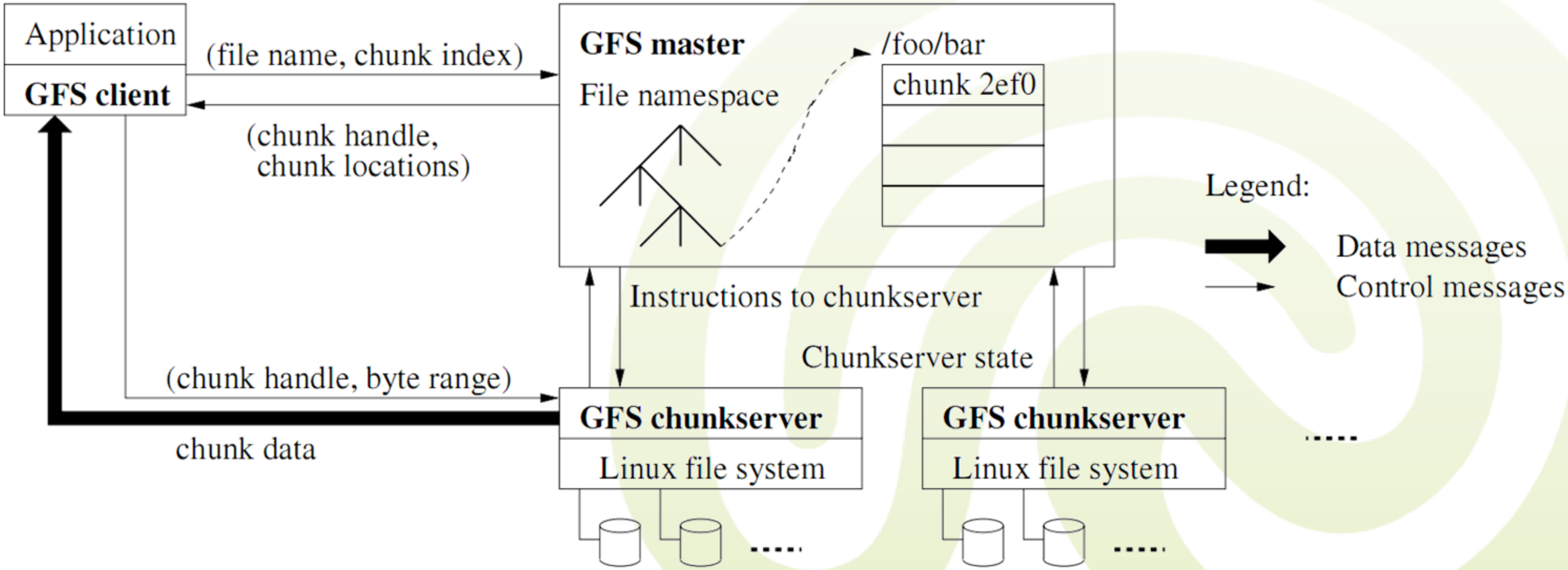
- Two common types of **read operations**
  - **Sequential streams** of large data quantities
    - e.g. streaming video, transferring a web index chunk, etc.
    - Frequent streaming renders **caching** useless
  - **Random reads** of small data quantities
    - However, random reads are usually "always forward", e.g. similar to a sequential read skipping large portions of the file
- Focus of GFS is on high **overall bandwidth**, not latency
  - In contrast to system like e.g. Amazon Dynamo
- File system **API** must be simple and expandable
  - **Flat file name space** suffices
    - File path is treated as string
      - » No directory listing possible
    - Qualifying file names consist of namespace and file name
  - No POSIX compatibility needed
  - Additional support for file appends and snapshot operations

GFS

- A **GFS cluster** represents a single file system for a certain set of applications
- Each **cluster** consists of
  - A **single master server**
    - The single master is one of the key features of GFS!
  - Multiple **chunk servers** per master
    - Accessed by multiple clients
  - Running on commodity Linux machines
- **Files** are split into **fixed-sized chunks**
  - Similar to file system **blocks**
  - Each labeled with a 64-bit unique global ID
  - Stored at a **chunk server**
  - Usually, each chunk is three times **replicated** across chunk servers

GFS

- **Application requests** are initially handled by a master server
  - Further, chunk-related communication is performed directly between application and chunk server



GFS

- **Master server**
  - **Maintains all metadata**
    - Name space, access control, file-to-chunk mappings, garbage collection, chunk migration
  - **Queries for chunks** are handled by the master server
    - Master returns only chunk **locations**
    - A client typically asks for multiple chunk locations in a single request
    - The master also optimistically provides chunk locations immediately following those requested
- **GFS clients**
  - Consult master for metadata
  - **Request data directly from chunk servers**
    - No caching at clients and chunk servers due to the frequent streaming

GFS

- **Files** (cont.)
  - Each file consists of multiple chunks
  - For each file, there is a meta-data entry
    - File **namespace**
    - File to chunk **mappings**
    - Chunk **location** information
      - Including **replicas**!
    - Access **control information**
    - Chunk **version numbers**

GFS

- **Chunks** are rather **large** (usually 64MB)
  - **Advantages**
    - Less chunk location requests
    - Less overhead when accessing large amounts of data
    - Less overhead for storing meta data
    - Easy caching of chunk metadata
  - **Disadvantages**
    - Increases risk for **fragmentation** within chunks
    - Certain chunks may become hot spots

GFS

- Meta-Data is kept in **main-memory of master server**
  - Fast, easy and efficient to periodically scan through meta data
    - **Re-replication** in the presence of chunk server failure
    - Chunk **migration** for load balancing
    - **Garbage collection**
  - Usually, there are **64Bytes** of metadata per 64MB chunk
    - Maximum capacity of GFS cluster limited by available main **memory of master**
  - In practice, query load on master server is low enough such that it never becomes a bottle neck

GFS

- Master server relies on **soft-states**
  - Regularly sends **heart-beat messages** to chunk servers
    - Is chunk server down?
    - Which **chunks** does chunk server store?
      - Including replicas
    - Are there any disk failures at a chunk server?
    - Are any replicas corrupted?
      - Test by comparing checksums
  - Master can send **instructions** to chunk server
    - **Delete** existing chunks
    - **Create** new empty chunk

GFS

- All modifications to meta-data are **logged** into an **operation log** to safeguard against GFS master failures
  - Meta-data updates are not that frequent
  - The operation log contains a historical record of critical metadata changes, **replicated** on multiple remote machines
  - **Checkpoints** for fast recovery
    - Operation log can also serve to reconstruct a timeline of changes
  - Files and chunks, as well as their versions are all **uniquely** and **eternally identified** by the logical times at which they were created
  - In case of failure, the master **recovers** its file system state by replaying the operation log
    - Usually, a **shadow master** is on **hot-standby** to take over during recovery

GFS

- Guarantees of GFS
  - **Namespace mutations** are always atomic
    - Handled by the master with locks
    - e.g. creating new files or chunks
    - Operation is only treated as successful when operation is performed and all log replicas are flushed to disk

GFS

- **Data mutations** follow a **relaxed consistency model**
  - A chunk is **consistent,** if all clients see the same data, independently of the queried replica
  - A chunk is **defined,** if all its modifications are visible
    - i.e. writes have been atomic
    - GFS can recognize defined and undefined chunks
  - In most cases, all chunks should be consistent and defined
    - …but not always.
    - Only using append operations for data mutations minimizes probability for undefined or inconsistent chunks

GFS

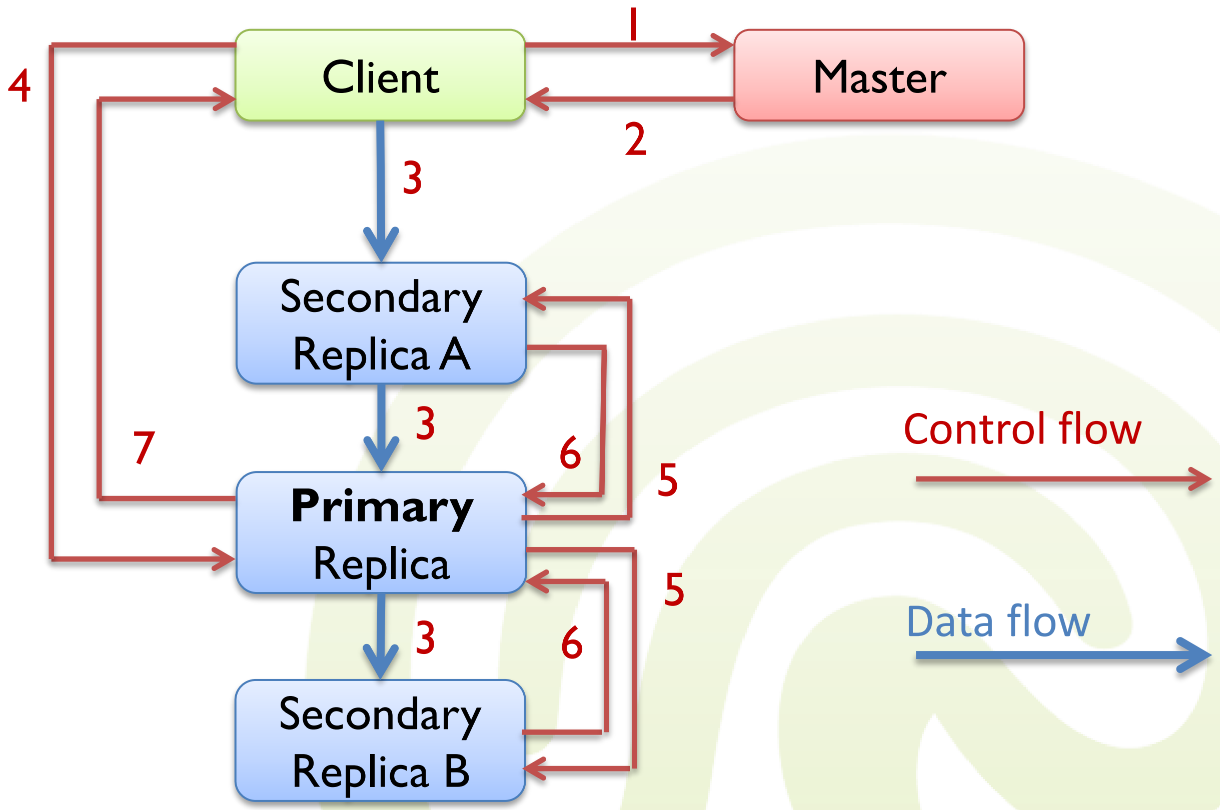
- **Mutation operations**
  - To encourage consistency among replicas, the master grants a **lease** for each chunk to a chunk server
    - Server owning the lease is responsible for that chunk
      - i.e. has the **primary** replica and is responsible for mutation operations
    - Leases are granted for a **limited time** (e.g. 1 minute)
      - Granting leases can be piggybacked to heartbeat messages
      - Chunk server may request a lease extension, if it currently mutates the chunk
      - If a chunk server fails, a new leases can be handed out after the original one expired
        - » No inconsistencies in case of partitions

GFS

- Mutation operations have a **separated data flow** and **control flow**
  - Idea: maximize bandwidth utilization and overall system throughput
  - Primary replica chunk server is responsible for control flow

GFS

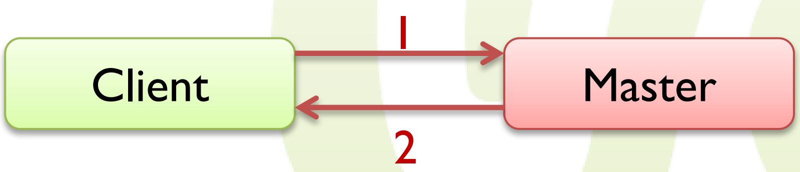
- **Mutation workflow overview**



GFS

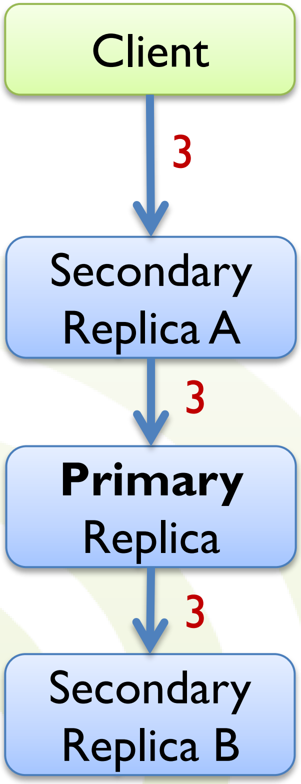
- **Application originates mutation request**

1. GFS client translates **request** from (filename, data) to (filename, chunk index), and sends it to master
   - Client "knows" which chunk to modify
     - Does not know where the chunk and its replicas are located
2. Master responds with **chunk handle** and (primary + secondary) **replica locations**

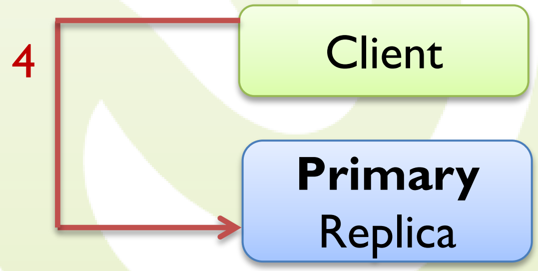


3. Client pushes write data to all replicas

- Client selects the "best" replica chunk server and transfers all new data
  - e. g. closest in the network, or with highest known bandwidth
  - Not necessarily the server holding the lease
  - New data: the new **data** and the **address range** it is supposed to replace
    - Exception: appends
- Data is stored in chunk servers' internal buffers
  - New data is stored as fragments in buffer
- New data is pipelined forward to next chunk server
  - … and then the next
  - Serially pipelined transfer of the data
  - Try to optimize bandwidth usage

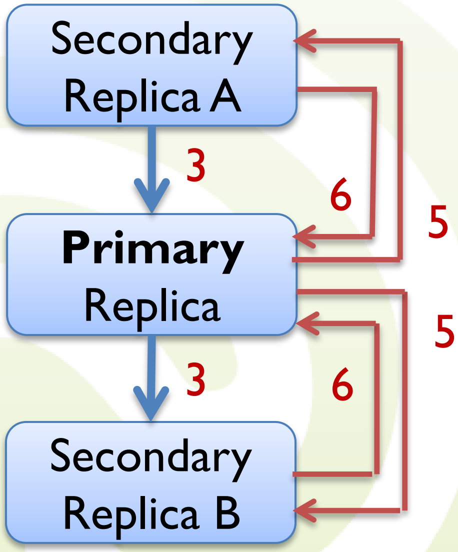


4. After all replicas received the data, the **client** sends a write request to the **primary chunk server**
   - Primary determines **serial order** for new data fragments stored in its buffer and **writes** the fragments in that order **to the chunk**
     - Write of fragments is thus **atomic**
       - No additional write request are served during write operation
     - Possibly multiple fragments from one or multiple clients



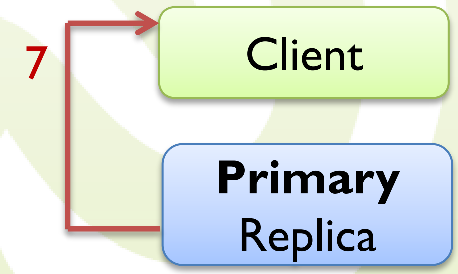
GFS

5. After the primary server successfully finished writing the chunk, it orders the replicas to write
   - The same serial order is used!
     - Also, the same timestamps are used
   - Replicas are inconsistent for a short time
6. After the replicas completed, the primary server is notified



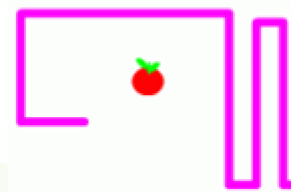
GFS

7. The primary notifies the client
   - Also, all error are reported to the client
     - Usually, errors are resolved by retrying some parts of the workflow
       - Some replicas may contain the same datum multiple times due to retries
       - Only guarantee of GFS: data will be written at least once atomically
     - Failures may render chunks inconsistent



GFS

- Google aims at using **append operations** for most mutations
  - For random updates, clients need to provide the **exact range** for the new data within the file
    - Easy to have collisions with other clients
      - i.e. client A write to range 1, client B overwrites range 1 because it assumed it as empty
      - Usually, locks would solve the problem
  - **Appends** can be easily performed in **parallel**
    - Just transfer new data to chunk server
      - Clients can transfer new data in parallel
      - Chunks server buffers data
    - Chunk server will find a correct position at the end of the chunk
      - Additional logic necessary for creating new chunks if current chunk cannot hold new data
  - Typical use case
    - Multiple producers append to the same file while simultaneously multiple consumer read from it
      - e.g. then of the web crawler and feature extraction engine



GFS

- **Master** takes care of **chunk creation** and distribution
  - New empty chunk **creation**, **re-replication**, **rebalances**
    - Master server notices if a chunk has to few replicas and can re-replicate
  - Master decides on chunk location. Heuristics:
    - Place new replicas on chunk servers with **below-average disk space utilization**. Over time this will equalize disk utilization across chunk servers
    - **Limit** the number of "**recent**" **creations** on each chunk server
      - Chunks should have different age to spread chunk correlation
    - **Spread** replicas of a chunk **across racks**

GFS

- After a file is deleted, GFS does not immediately reclaim the available physical storage
  - Just delete meta-data entry from the master server
  - File or chunks become **stale**
- Chunks or files may also become stale if a **chunk server misses** an update to a chunk
  - Updated chunk has a different Id than old chunk
  - Master server holds only links to new chunks
    - Master knows the current chunks of a file
    - Heartbeat messages with unknown (e.g. old) chunks are ignored
- During **regular garbage collection**, stale chunks are **physically deleted**

GFS

- **Experiences with GFS**
  - Chunk server workload
    - Bimodal distribution of small and large files
    - Ratio of **write** to **append** operations: 4:1 to 8:1
    - Virtually no overwrites
  - **Master workload**
    - Most request for chunk locations and open files
  - Reads achieve 75% of the network limit
  - Writes achieve 50% of the network limit

- **Summary and notable features GFS**
  - GFS is a **distributed file system**
    - Optimized for **file append operations**
    - Optimized for **large files**
  - File are split in rather large 64MB **chunks** and **distributed** and **replicated**
  - Uses single **master server** for file and chunk management
    - All meta-data in master server in main memory
  - Uses flat namespaces