Distributed File System

Mirek Riedewald

Key Learning Goals

- What does the distributed file system have in common with traditional single-machine file systems? What is different?
- Why is the single master node (namenode in HDFS) not likely to become a bottleneck?
- Do Hadoop MapReduce and Spark need to be able to update files?

Introduction

- The file system is an essential component of the operating system (e.g., Windows, Linux, MacOS). It stores data permanently, organized into files and directories (a.k.a. folders). Not surprisingly, the file system plays a crucial role for scalable data processing as well.
- We cover the Google File System (GFS), which inspired the popular Hadoop File System (HDFS).
  - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proc. of the nineteenth ACM Symposium on Operating Systems Principles* (SOSP '03), 29-43

What Does a File System Do?

- Create, rename, delete a file or directory.
- Open or close a file.
- Move to a specific position (an *offset*) in a file.
- Read or write a certain number of bytes in the file, starting from a given offset or the current position.
- Append new data at the end of a file.
- Move a file from one directory to another.

What Is Special about a Distributed File System (DFS)?

- To avoid I/O bottlenecks, the data should be distributed over many servers.
- The size of a single disk drive should not limit file size. For instance, it should be easy to store a 100 terabyte file, even if none of the disks is bigger than 10 terabytes. This means that the big file needs to be chunked up and distributed over multiple disks *automatically*.
- All operations should be as simple as in a regular file system, i.e., applications should not have to explicitly deal with file chunks and file locations. For instance, if machine X creates a file stored on machine Y, then machine Z should be able to access it easily.
- The DFS has to prevent concurrency bugs: What if two processes are trying to create a file with the same name, or write to the same file? How do file system accesses get synchronized? Who coordinates the different machines participating in the file system?

Motivation for Use of a DFS

- The abstraction of a single global file system simplifies programming warehouse-scale computers (i.e., clusters of commodity machines). A client can access a file that is possibly distributed over many machines as if it was a regular file stored locally.
- This frees the programmer from having to worry about messy details such as:
  - Into how many chunks should a large file be split and on which machines in the cluster should these chunks be stored?
  - Keeping track of the chunks and which chunk(s) to modify when the user tries to change a sequence of bytes somewhere in the file.
  - Management of chunk replicas: Replicas are needed to provision against failures. (Recall that in a large cluster of commodity machines failures are common.) These replicas have to be kept in sync with each other. What should be done when a machine fails? Which other machine should receive a new copy of a file chunk that was stored on the failed machine? When the failed machine comes back online, should the additional copy be removed?
  - Coordinating concurrent file access: What if two processes are trying to update the same file chunk? What if the processes access different chunks of the same file? How do we make sure that file creation is atomic, i.e., no two processes can create the same file concurrently?

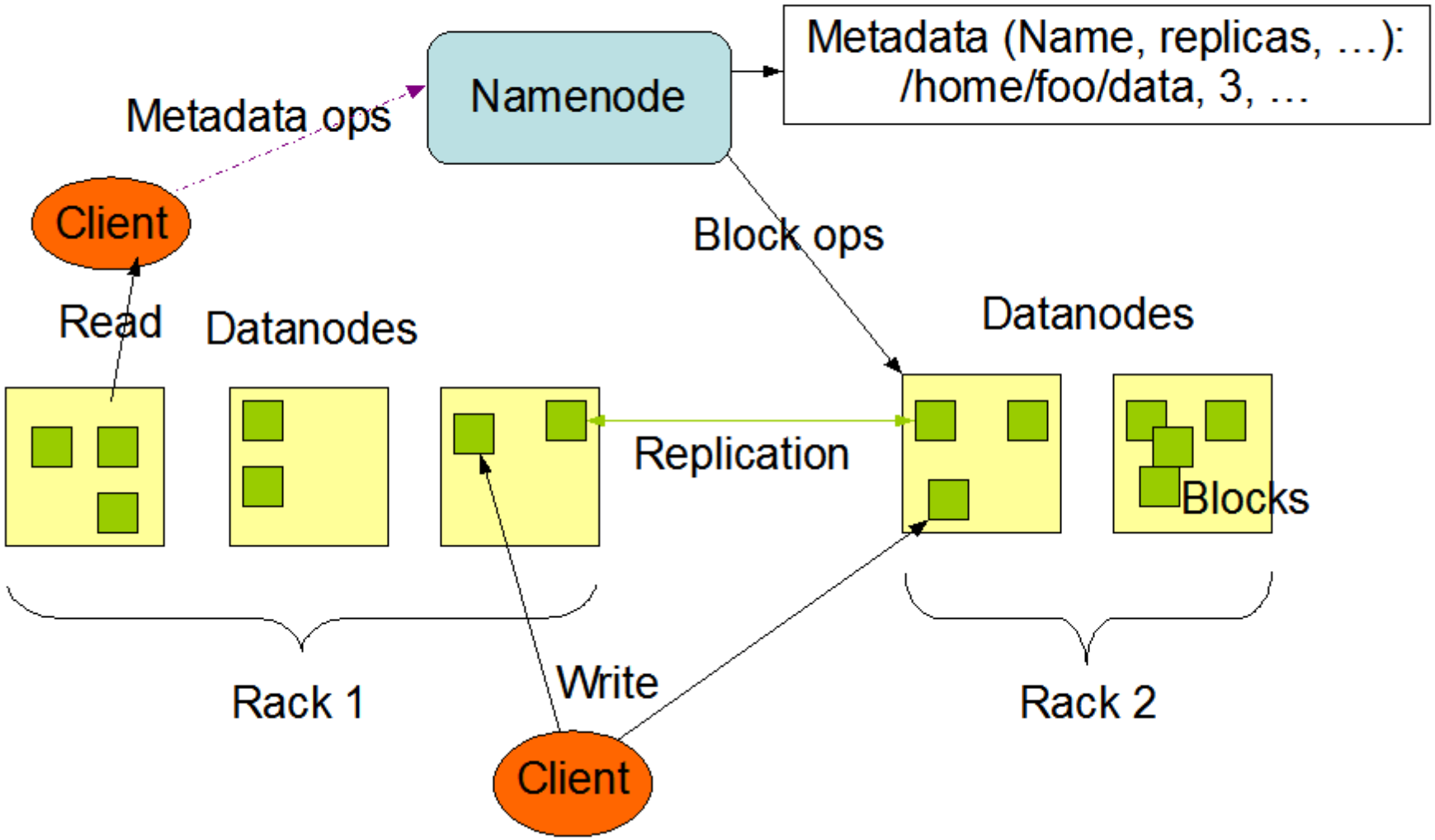
Main GFS Design Choices

- A single master service is used to ensure consistency and achieve consensus for concurrent file system operations.
  - In HDFS this is called a *namenode*.
- To avoid a bottleneck at the master, it only performs lightweight tasks. These are tasks related to metadata management and location of data chunks.
- Since the master is a single point of failure, it should be able to recover quickly.
- Data management and data transfer is performed directly by chunkservers. This distributes the heavy tasks.
  - In HDFS these are called *datanodes*.
- HDFS further reduces problem complexity by not supporting any file modifications, other than concatenation. Its write-once-read-many approach guarantees high read throughput.
  - To “modify” a file, one has to create a new copy with the updated content.

Other GFS Design Goals

- The distributed file system should be able to handle a modest number of large files. Google aimed for a few million files, most of them 100 MB or larger in size. In particular, the Google File System (GFS) was designed to handle multi-gigabyte and larger files efficiently.
- High sustained bandwidth is more important than low latency. This is based on the assumption that the file system would have to deal mostly with large bulk accesses.
  - Small reads and writes of a few bytes at random locations in a file would be costly in a distributed file system due to the high access latency.
- GFS should support a typical file system interface:
  - Files are organized in directories.
  - The file system supports operations such as create, delete, open, close, read, and write with syntax similar to a local file system.

HDFS Architecture



Source: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed in August 2018.

GFS and HDFS Architecture Details

- All machines are commodity Linux machines.
- Files are divided into fixed-size chunks, typically 64 MB or 128 MB in size a few years ago, now also 256 MB. These chunks are stored on the chunkservers' local disks as Linux files. Chunks are typically replicated on multiple chunkservers, e.g., three times.
- The master maintains all file system metadata such as namespace, access control information, the mapping from files to chunks, and chunk locations.
  - It uses locks to avoid concurrency-related conflicts, e.g., when creating a new file or directory.

A Note on Chunk Size

- The number of chunks is limited by the master's memory size. (For fast accesses, the master manages all metadata in memory.)
  - In GFS, there are only 64 bytes of metadata per 64 MB chunk. Note that most chunks are "full," because all but the last chunk are completely filled and in large files the number of chunks is large.
  - GFS uses less than 64 bytes of namespace data per file.

- Advantages of large chunk size:
  - There will be fewer interactions with the master. Recall that GFS is designed to support large sequential reads and writes. Reading a block of 128 MB would involve 2-3 chunks of size 64 MB, but 128-129 chunks of size 1 MB.
  - For the same file size, less chunk location information is needed, reducing the amount of information managed by the master. Similarly, for large chunk sizes, metadata even for terabyte-sized working sets could be cached in the clients' memory.

- Disadvantage of large chunk size:
  - Fewer chunks result in fewer options for load balancing. For example, consider clients reading at offsets 1 million, 3 million, and 5 million. Using a 1 MB chunk size, each client would access a different chunk. Using a 64 MB chunk size, all three would access the same one. This can result in hotspots (for reading and writing) or concurrency-related delays (for writing).
    - Read hotspots can be addressed by a higher replication factor, distributing the load over more replicas.
    - Read hotspots can also be addressed by letting clients read from other clients who already got the chunk.

A Note on Chunk-Replica Placement

- The goals of choosing machines to host the replicas of a chunk are scalability, reliability, and availability. Achieving this is difficult, because in a typical GFS setting there are hundreds of chunkservers spread across many machine racks, accessed from hundreds of clients in the same or different racks. Communication may cross multiple network switches. And bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the same rack.
- Spreading replicas across different racks is good for fault tolerance. Furthermore, read operations benefit from the aggregate bandwidth of multiple racks. On the other hand, it is bad for writes and file creation, as the data now flows through multiple racks.
- The master can move replicas or create/delete them to react to system changes and failures.

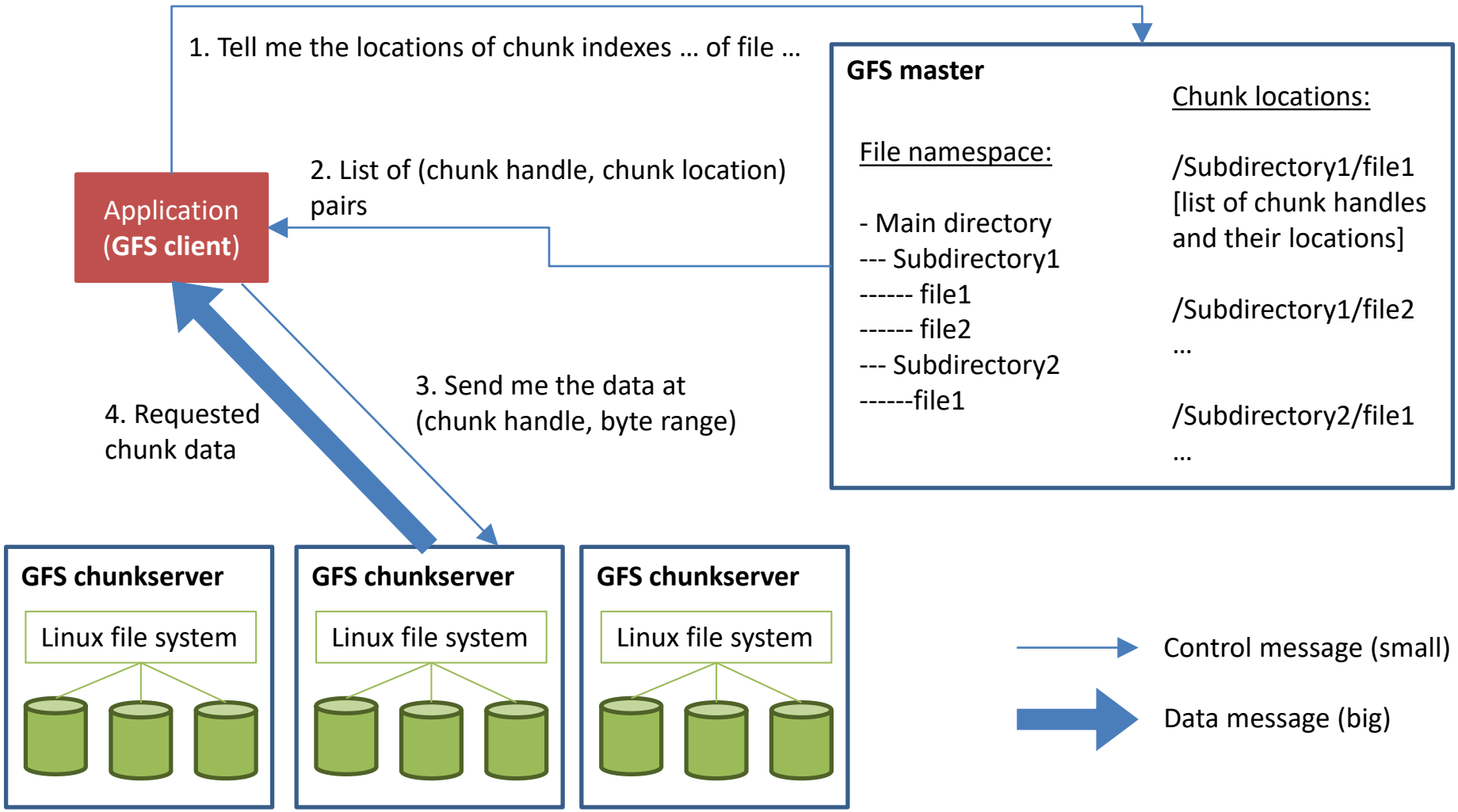
High-Level GFS Functionality

- The master controls all system-wide activities such as chunk lease management, garbage collection, and chunk migration.
- The master communicates with the chunkservers through HeartBeat messages to give instructions and collect their state.
- The client who tries to access a file gets metadata (in particular the locations of relevant file chunks) from the master, but then accesses files directly through the chunkservers.
- There is no GFS-level file caching for several reasons:
  - Caching offers little benefit for streaming access (i.e., reading a file “in one direction”) or for large working sets.
  - This avoids having to deal with cache coherence issues in the distributed setting.
  - On each chunkserver, standard (local) Linux file caching is sufficient for good performance.

Read Access

- Consider a client that wants to read B bytes from file F, starting at offset O. It contacts the master to request the location of the corresponding chunks.
  - It is easy to calculate the chunk number from O, B, and chunk size S. The first chunk needed has number $\left\lfloor \frac{O}{S} \right\rfloor$, and the last has number $\left\lfloor \frac{O+B}{S} \right\rfloor$.
- Knowing the locations of the replicas of the requested chunk, the client requests the data from a nearby chunkserver. This data transfer does not involve the master at all.
- Many clients can read from different chunkservers in parallel, enabling GFS to support a high aggregate data transfer rate.
- Since chunk locations rarely change, the client typically buffers this location info for some time. This way it can avoid contacting the master for future accesses to the same chunk.

Read Access Illustration

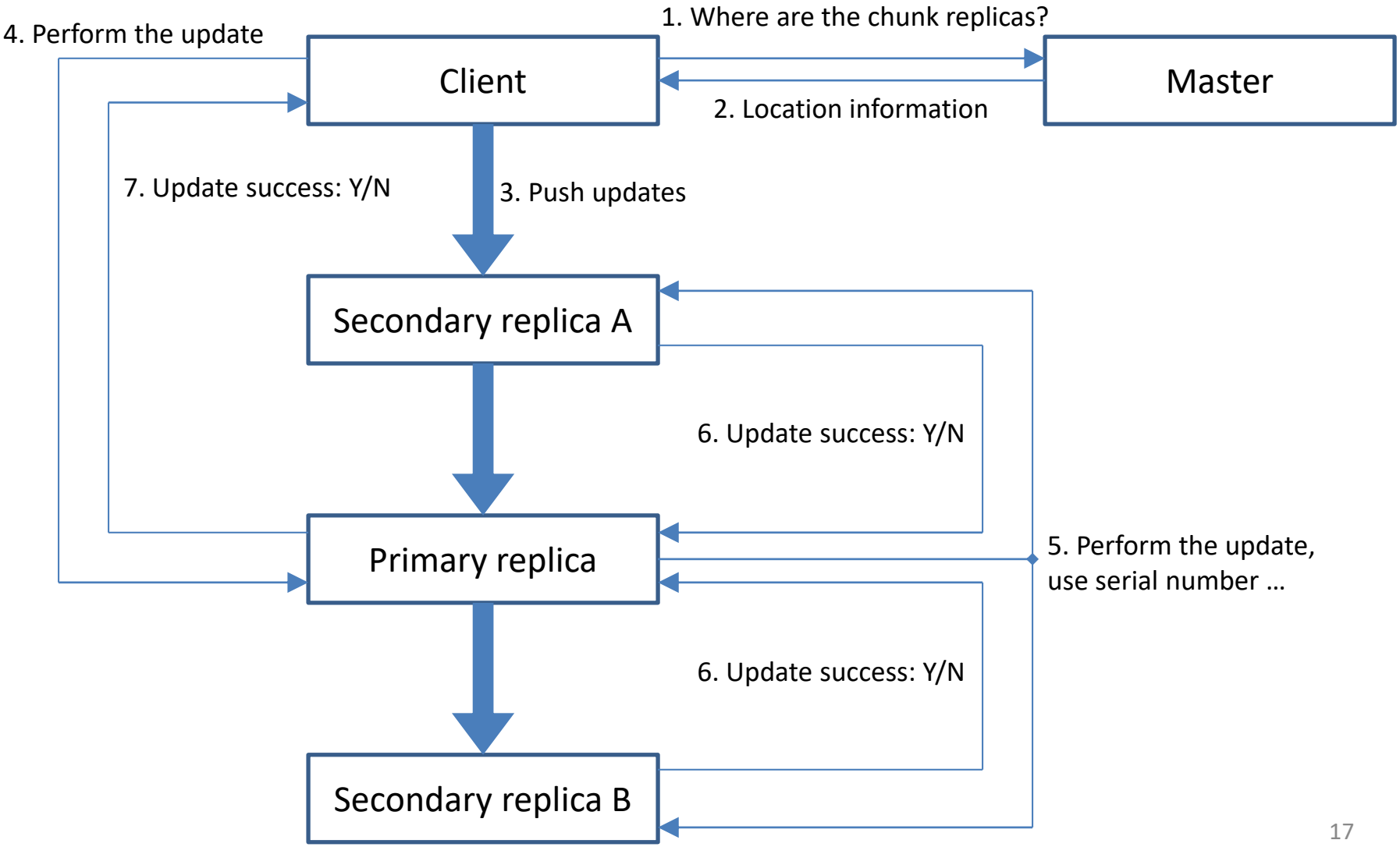


Updating a Chunk

Writing to a file is more challenging than reading, because all chunk replicas have to be in sync. This means that they have to apply the same updates in the same order. GFS achieves this by designating one replica of a chunk as the primary. It determines update order and notifies the secondaries to follow its lead. Large updates are broken into chunk-wise updates. File creation proceeds similarly, but does not need an update order, and therefore does not need to distinguish primary and secondary copies.

- The client contacts the master to find the locations of the replicas of the chunk it is trying to update.
- The master responds with the requested information.
- The client starts pushing the updates to all replicas.
- After receiving all acknowledgements that the update arrived, the client sends a write request to the primary who assigns it a serial number.
- The primary forwards the write request, including its serial number, to all other replicas.
- After completing the write request, the secondaries acknowledge success to the primary.
- The primary replies to the client about the outcome of the update operation. If the operation failed, the client would re-try.

Update Process Illustration



Pushing Updates and New Files

- In GFS, data flow is decoupled from control flow for efficient network use. Instead of the client multi-casting the chunk update to each replica, the data is pipelined linearly along a chain of chunkservers. There are several reasons for this design decision:
  - Pipelining makes use of the full outbound bandwidth for the fastest possible transfer, instead of dividing it in a non-linear topology.
  - It avoids network bottlenecks by forwarding to the “next closest” destination machine.
  - It minimizes latency: once a chunkserver receives data, it starts forwarding to the next one in the pipeline immediately. Notice that the machines are typically connected by a switched network with full-duplex links. This means that sending out data does not affect the bandwidth for receiving data (and vice versa).

Updates and Data Consistency

- It is challenging to ensure consistent update order across replicas when machines can fail or be slow to respond.
- GFS has mechanisms to deal with these problems, but for general updates (in contrast to append-only updates) it relies on a relaxed consistency model. This means that under certain circumstances, in particular when a client uses cached chunk location information, it might read from a stale chunk replica.
- Life would be easier without updates. There would be no need to keep replicas in sync and no risk of reading stale replicas.
  - This is why HDFS does not support updates. Hadoop MapReduce and Spark do not need update functionality. They can simply write to new output files.
  - For instance, each Reduce task in a MapReduce job creates a temp file in the local file system. It then atomically copies and renames it to a file in the global file system.

Achieving High Availability

- Master and chunkservers can restore their state and start up in *seconds*.
- Chunk replication ensures data availability in the presence of failures.
- Master data, in particular operation log and checkpoints, can be replicated to enable quick failover. When the master fails permanently, monitoring infrastructure outside GFS will start a new master with the replicated operation log. Since clients use a DNS alias, they are automatically re-routed to the new master.

Distributed File System Summary

- Distributed file systems support large-scale data processing workloads on commodity hardware.
- Component failures are treated as the norm, not the exception. GFS deals with failures through multiple mechanisms:
  - Constant monitoring of all participants.
  - Replication of crucial data.
  - A relaxed consistency model for updates, which does not negatively affect data consistency for MapReduce or Spark.
  - Fast, automatic recovery.
- GFS is optimized for huge files, append writes, and large sequential reads.
- It achieves high aggregate throughput for concurrent readers and writers through separation of file system control (through master) from data transfer (directly between chunkservers and clients).

File Management in the Cloud

- In the Cloud, GFS and HDFS are not required for managing files. In fact, it might be preferable to separate computation from storage. Why?
  - User perspective: The user can shut down the compute resources as soon as the job terminates to save money. The data will persist on the storage system.
  - Cloud provider perspective: It is easier to manage specialized services.
- On Amazon AWS, data usually resides on the S3 storage service. MapReduce and Spark jobs on EMR can use S3 as the data layer (instead of HDFS), or they copy data into HDFS at the start of the job, use HDFS for intermediate results, and then transfer the final result back to S3. Amazon also offers (as of August 2018) EMRFS, an implementation of the Hadoop file system designed for reading and writing files from EMR directly to S3.It uses URI prefix s3://.
- Note that Hadoop MapReduce and Spark can also read from other file systems, including the local one.
  - The standalone versions, which are running on a single machine in sequential mode, and are useful for debugging, by default access data in the local file system.

More Information about Amazon

- Source: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-file-systems.html (August 2018)

File system (prefix)	Desription
HDFS (hdfs:// or no prefix)	An advantage of HDFS is data awareness between the Hadoop cluster nodes managing the clusters and the Hadoop cluster nodes managing the individual steps. HDFS is used by the master and core nodes. One advantage is that it's fast; a disadvantage is that it's ephemeral storage which is reclaimed when the cluster ends. It's best used for caching the results produced by intermediate job-flow steps.
EMRFS (s3://)	EMRFS is an implementation of the Hadoop file system used for reading and writing regular files from Amazon EMR directly to Amazon S3. EMRFS provides the convenience of storing persistent data in Amazon S3 for use with Hadoop while also providing features like Amazon S3 server-side encryption, read-after-write consistency, and list consistency.
local file system	When a Hadoop cluster is created, each node is created from an EC2 instance that comes with a preconfigured block of preattached disk storage called an *instance store*. Data on instance store volumes persists only during the life of its EC2 instance. Instance store volumes are ideal for storing temporary data that is continually changing, such as buffers, caches, scratch data, and other temporary content.
Amazon S3 block file system (s3bfs://)	The Amazon S3 block file system is a legacy file storage system. We strongly discourage the use of this system.

References

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proc. of the nineteenth ACM Symposium on Operating Systems Principles* (SOSP '03), 29-43
  - https://scholar.google.com/scholar?cluster=9821092550821837 1&hl=en&as_sdt=0,22
- Amazon EMR documentation: Management guide
  - https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-what-is-emr.html