Google File System

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

What is the Problem?

What is the Problem?

- Crawl the whole web.
- Store it all on one big disk.
- Process users' searches on one big CPU.

What is the Problem?

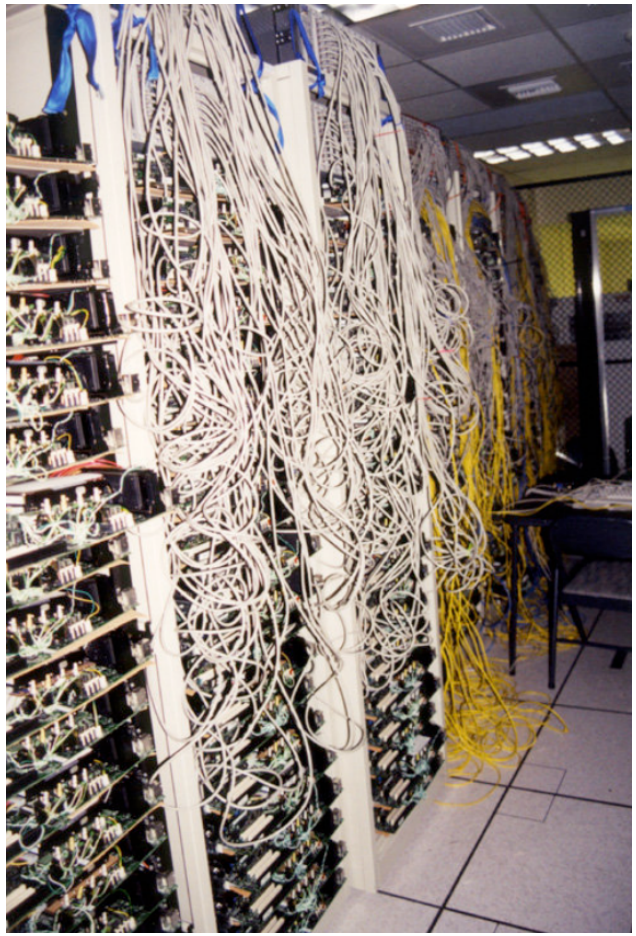
- Crawl the whole web.
- Store it all on one big disk.
- Process users' searches on one big CPU.
- Does not scale.

Motivation and Assumptions (1/3)

- Lots of cheap PCs, each with disk and CPU.
  - How to share data among PCs?

Motivation and Assumptions (2/3)

- 100s to 1000s of PCs in cluster.
  - Failure of each PC.
  - Monitoring, fault tolerance, auto-recovery essential.



Motivation and Assumptions (3/3)

- Large files: ≥ 100 MB in size.

Motivation and Assumptions (3/3)

- Large files: ≥ 100 MB in size.
- Large streaming reads and small random reads.

Motivation and Assumptions (3/3)

- Large files: ≥ 100 MB in size.
- Large streaming reads and small random reads.
- Append to files rather than overwrite.

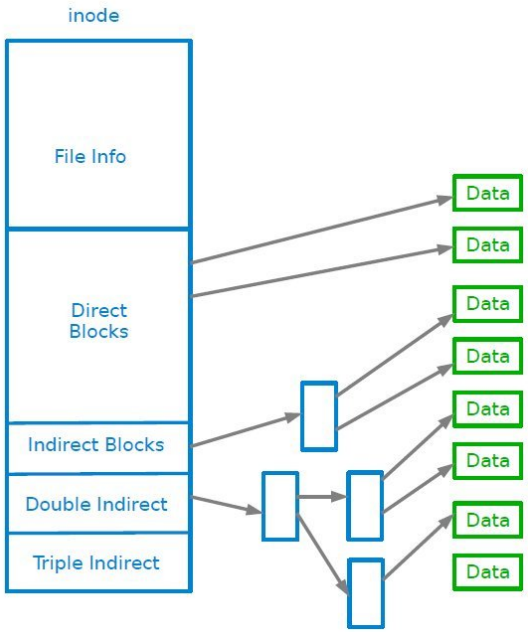
Reminder

What is Filesystem?

- Controls how data is stored in and retrieved from disk.

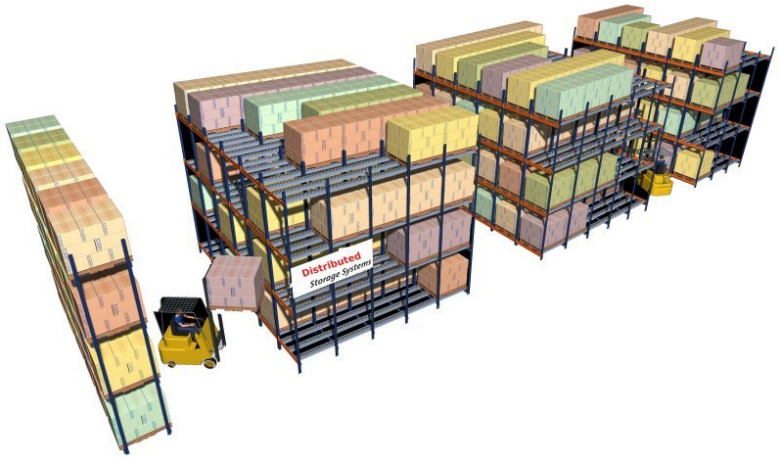
What is Filesystem?

- Controls how data is stored in and retrieved from disk.



Distributed Filesystems

- When data outgrows the storage capacity of a single machine: partition it across a number of separate machines.
- Distributed filesystems: manage the storage across a network of machines.



Google File System (GFS)

GFS

- Appears as a single disk
- Runs on top of a native filesystem.
- Fault tolerant: can handle disk crashes, machine crashes, ...
- Hadoop Distributed File System (HDFS) is an open source Java product similar to GFS.

GFS is Good for ...

- Storing large files
  - Terabytes, Petabytes, etc...
  - 100MB or more per file.

GFS is Good for ...

- Storing large files
  - Terabytes, Petabytes, etc...
  - 100MB or more per file.
- Streaming data access
  - Data is written once and read many times.
  - Optimized for batch reads rather than random reads.

GFS is Good for ...

- Storing large files
  - Terabytes, Petabytes, etc...
  - 100MB or more per file.
- Streaming data access
  - Data is written once and read many times.
  - Optimized for batch reads rather than random reads.
- Cheap commodity hardware
  - No need for super-computers, use less reliable commodity hardware.

GFS is Not Good for ...

- Low-latency reads
  - High-throughput rather than low latency for small chunks of data.

GFS is Not Good for ...

- Low-latency reads
  - High-throughput rather than low latency for small chunks of data.
- Large amount of small files
  - Better for millions of large files instead of billions of small files.

GFS is Not Good for ...

- Low-latency reads
  - High-throughput rather than low latency for small chunks of data.
- Large amount of small files
  - Better for millions of large files instead of billions of small files.
- Multiple writers
  - Single writer per file.
  - Writes only at the end of file, no-support for arbitrary offset.

Files and Chunks (1/2)

- Files are split into chunks.
- Chunks
  - Single unit of storage: a contiguous piece of information on a disk.
  - Transparent to user.
  - Chunks are traditionally either 64MB or 128MB: default is 64MB.

Files and Chunks (2/2)

- Why is a chunk in GFS so large?

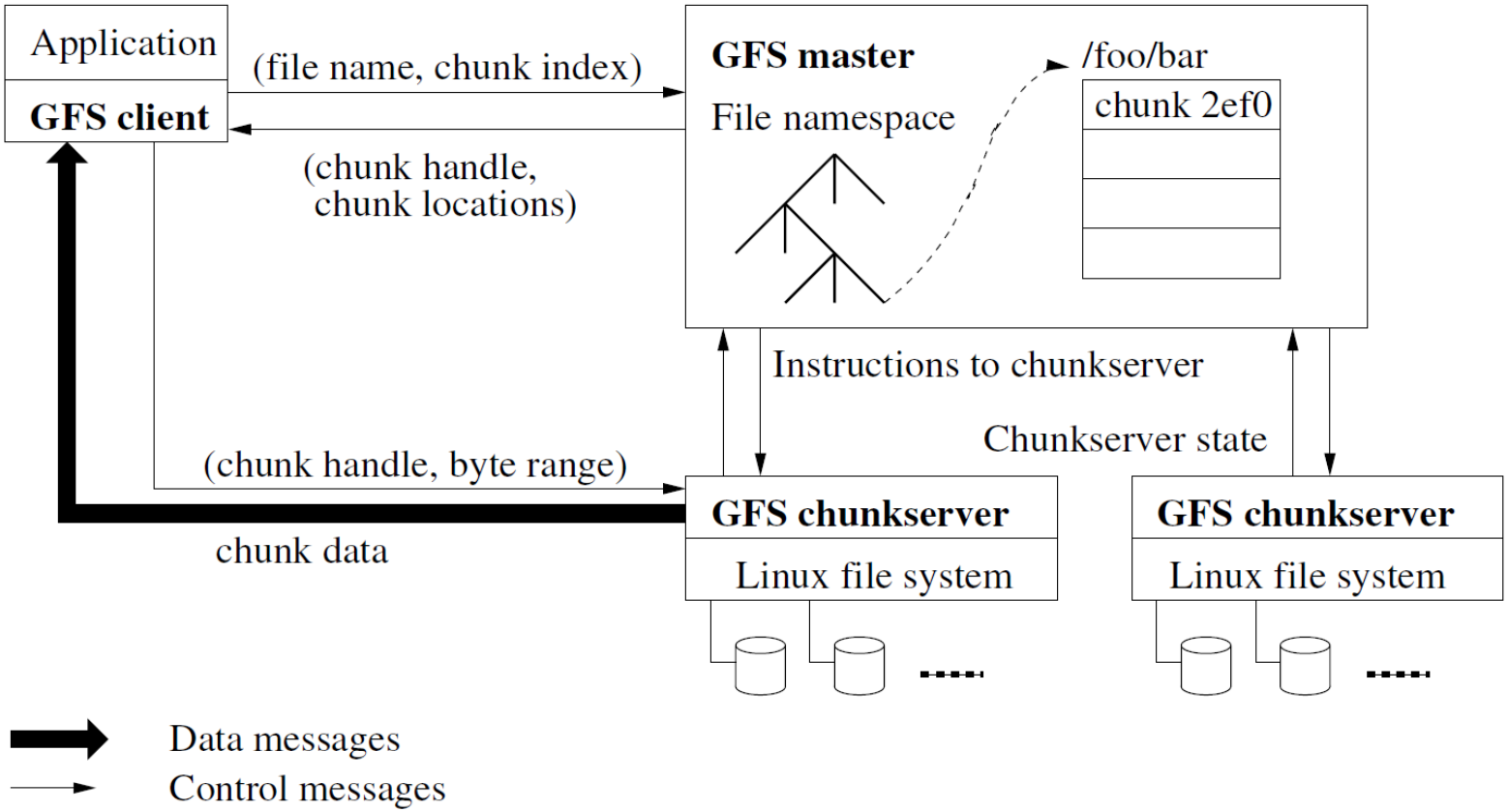
Files and Chunks (2/2)

- Why is a chunk in GFS so large?
  - To minimize the cost of seeks.
- Time to read a chunk = seek time + transfer time
- Keeping the ratio $\frac{seektime}{transfertime}$ small: we are reading data from the disk almost as fast as the physical limit imposed by the disk.

Files and Chunks (2/2)

- Why is a chunk in GFS so large?
  - To minimize the cost of seeks.
- Time to read a chunk = seek time + transfer time
- Keeping the ratio $\frac{seektime}{transfertime}$ small: we are reading data from the disk almost as fast as the physical limit imposed by the disk.
- Example: if seek time is 10ms and the transfer rate is 100MB/s, to make the seek time 1% of the transfer time, we need to make the chunk size around 100MB.

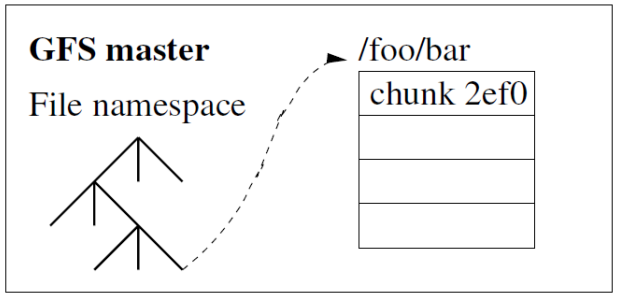
GFS Architecture



- Main components:
  - GFS master
  - GFS chunk server
  - GFS client

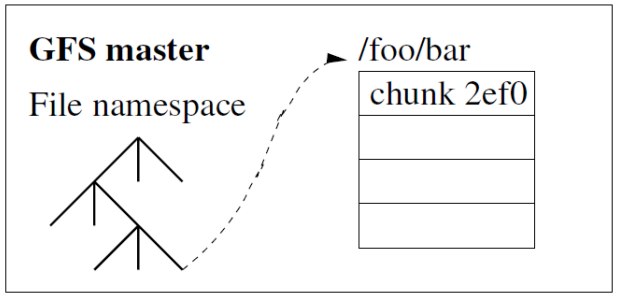
GFS Master

- Manages file namespace operations.



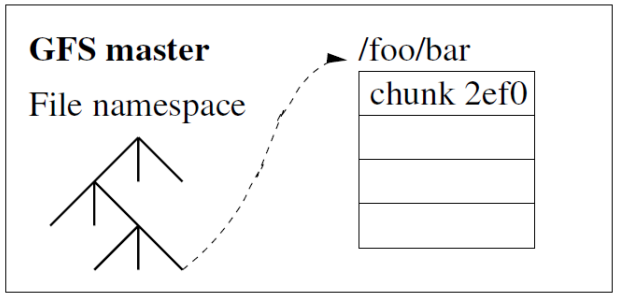
GFS Master

- Manages file namespace operations.
- Manages file metadata (holds all metadata in memory).
  - Access control information
  - Mapping from files to chunks
  - Locations of chunks



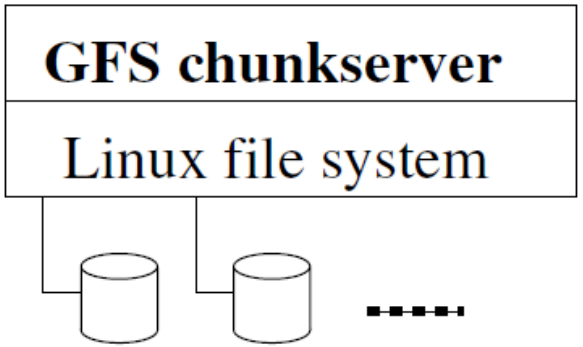
GFS Master

- Manages file namespace operations.
- Manages file metadata (holds all metadata in memory).
  - Access control information
  - Mapping from files to chunks
  - Locations of chunks
- Manages chunks in chunk servers.
  - Creation/deletion
  - Placement
  - Load balancing
  - Maintains replication
  - Garbage collection



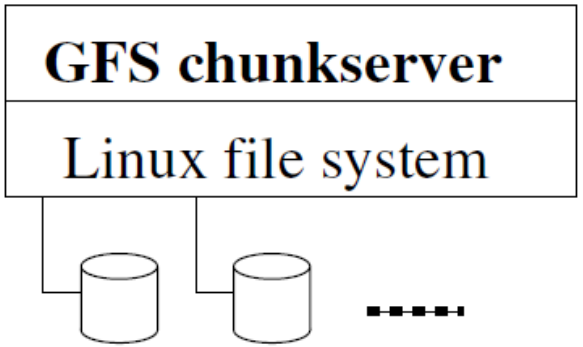
GFS Chunk Server

- Manage chunks.



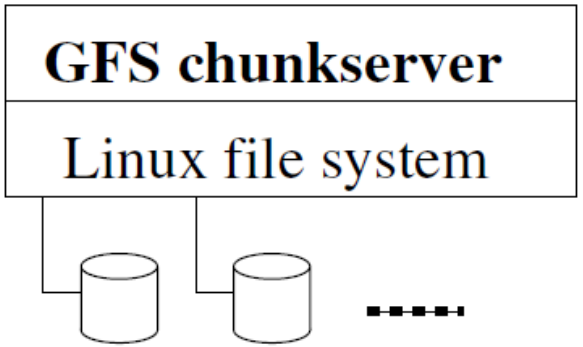
GFS Chunk Server

- Manage chunks.
- Tells master what chunks it has.



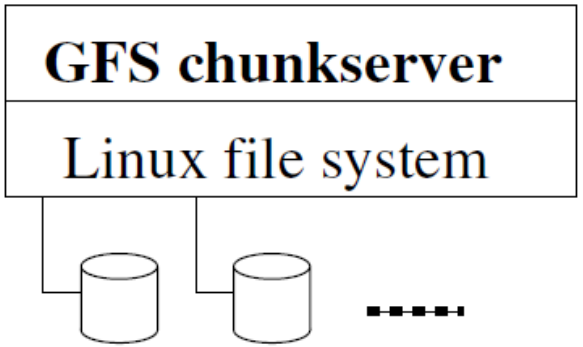
GFS Chunk Server

- Manage chunks.
- Tells master what chunks it has.
- Store chunks as files.



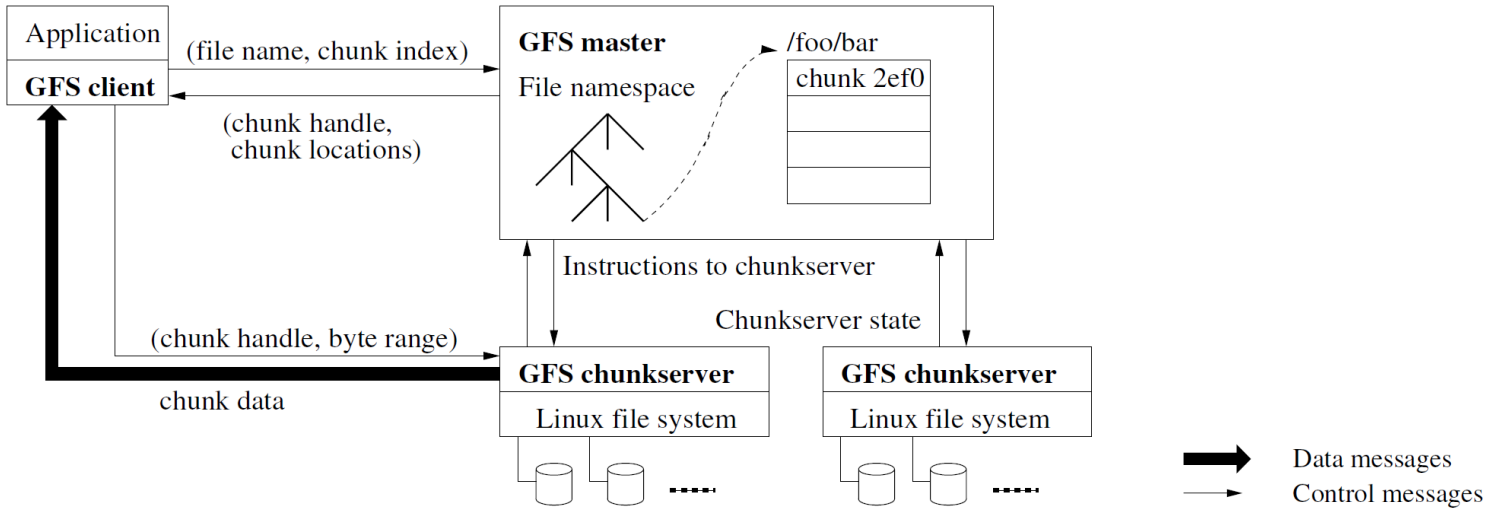
GFS Chunk Server

- Manage chunks.
- Tells master what chunks it has.
- Store chunks as files.
- Maintain data consistency of chunks.



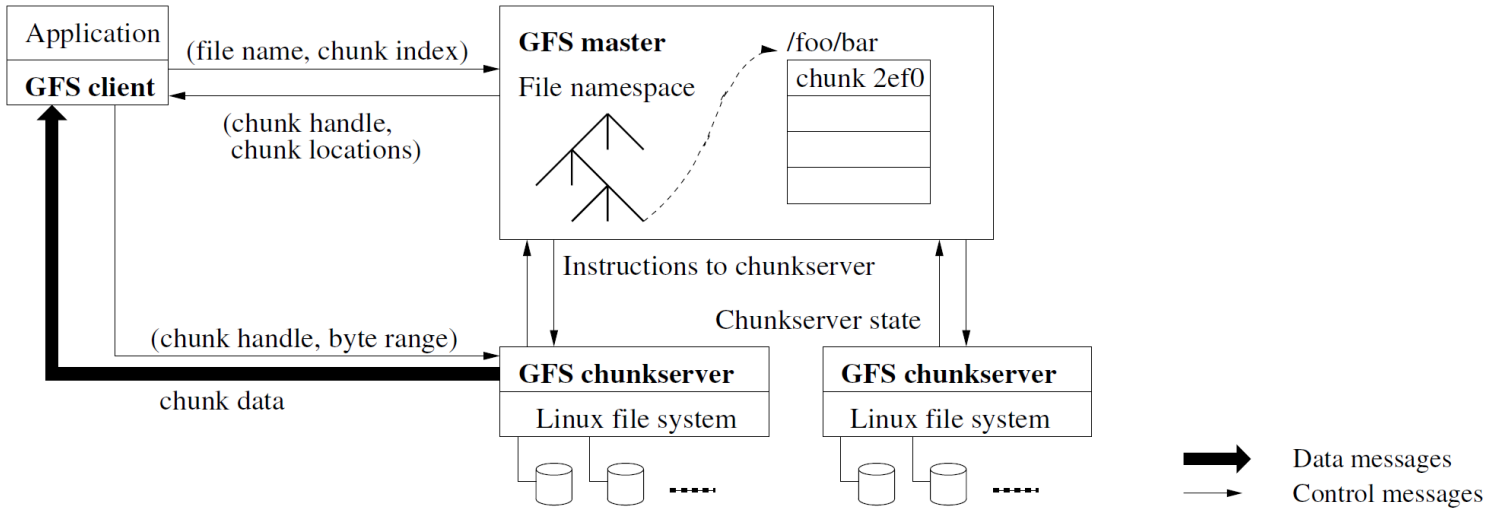
GFS Client

- Issues control (metadata) requests to master server.



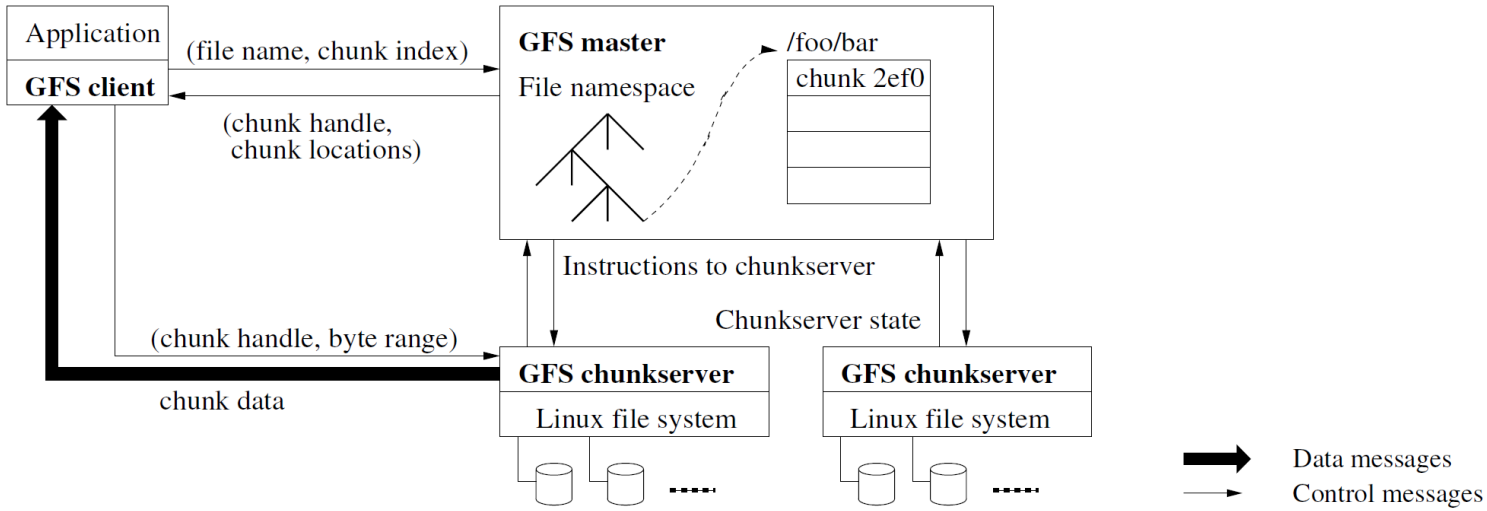
GFS Client

- Issues control (metadata) requests to master server.
- Issues data requests directly to chunk servers.



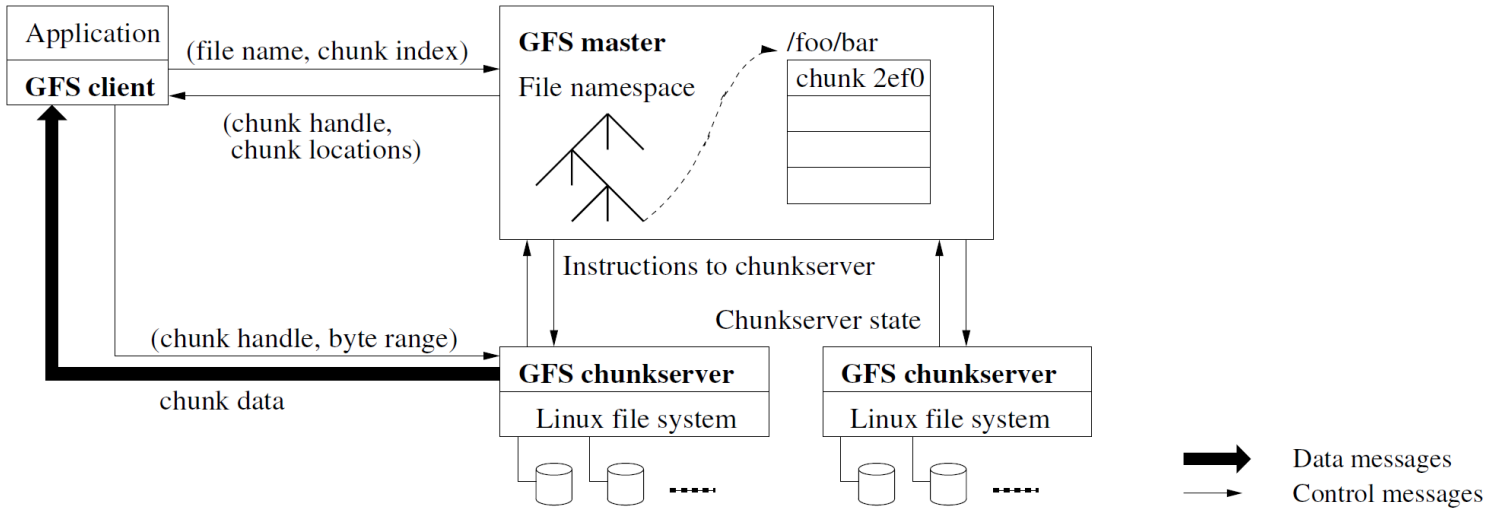
GFS Client

- Issues control (metadata) requests to master server.
- Issues data requests directly to chunk servers.
- Caches metadata.



GFS Client

- Issues control (metadata) requests to master server.
- Issues data requests directly to chunk servers.
- Caches metadata.
- Does not cache data.



The Master Operations

The Master Operations

- Namespace management and locking
- Replica placement
- Creating, re-replicating and re-balancing replicas
- Garbage collection
- Stale replica detection

Namespace Management and Locking

- No per-directory data structure.
- No hard or symbolic links.

Namespace Management and Locking

- No per-directory data structure.
- No hard or symbolic links.
- Represents its namespace as a lookup table mapping full pathnames to metadata.

Namespace Management and Locking

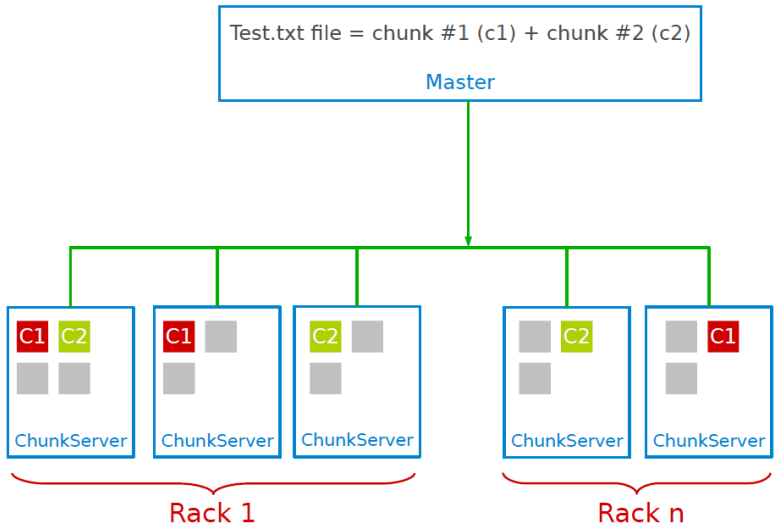
- No per-directory data structure.
- No hard or symbolic links.
- Represents its namespace as a lookup table mapping full pathnames to metadata.
- Each master operation acquires a set of locks before it runs.
- Read lock on internal nodes and read/write lock on the leaf.

Namespace Management and Locking

- No per-directory data structure.
- No hard or symbolic links.
- Represents its namespace as a lookup table mapping full pathnames to metadata.
- Each master operation acquires a set of locks before it runs.
- Read lock on internal nodes and read/write lock on the leaf.
- Allowed concurrent mutations in the same directory
- Read lock on directory prevents its deletion, renaming or snapshot

Replica Placement

- Maximize data reliability, availability and bandwidth utilization.
- Replicas spread across machines and racks, for example:
  - 1st replica on the local rack.
  - 2nd replica on the local rack but different machine.
  - 3rd replica on the different rack.
- The master determines replica placement.



Creation, Re-replication and Re-balancing

- Creation
  - Place new replicas on chunk servers with below-average disk usage.
  - Limit number of recent creations on each chunk servers.

Creation, Re-replication and Re-balancing

- Creation
  - Place new replicas on chunk servers with below-average disk usage.
  - Limit number of recent creations on each chunk servers.
- Re-replication
  - When number of available replicas falls below a user-specified goal.

Creation, Re-replication and Re-balancing

- Creation
  - Place new replicas on chunk servers with below-average disk usage.
  - Limit number of recent creations on each chunk servers.
- Re-replication
  - When number of available replicas falls below a user-specified goal.
- Rebalancing
  - Periodically, for better disk utilization and load balancing.
  - Distribution of replicas is analyzed.

Garbage Collection

- File deletion logged by master.
- File renamed to a hidden name with deletion timestamp.

Garbage Collection

- File deletion logged by master.
- File renamed to a hidden name with deletion timestamp.
- Master regularly deletes files older than 3 days (configurable).
- Until then, hidden file can be read and undeleted.

Garbage Collection

- File deletion logged by master.
- File renamed to a hidden name with deletion timestamp.
- Master regularly deletes files older than 3 days (configurable).
- Until then, hidden file can be read and undeleted.
- When a hidden file is removed, its in-memory metadata is erased.

Stale Replica Detection

- Chunk replicas may become stale: if a chunk server fails and misses mutations to the chunk while it is down.

Stale Replica Detection

- Chunk replicas may become stale: if a chunk server fails and misses mutations to the chunk while it is down.
- Need to distinguish between up-to-date and stale replicas.

Stale Replica Detection

- Chunk replicas may become stale: if a chunk server fails and misses mutations to the chunk while it is down.
- Need to distinguish between up-to-date and stale replicas.
- Chunk version number:
  - Increased when master grants new lease on the chunk.
  - Not increased if replica is unavailable.

Stale Replica Detection

- Chunk replicas may become stale: if a chunk server fails and misses mutations to the chunk while it is down.
- Need to distinguish between up-to-date and stale replicas.
- Chunk version number:
  - Increased when master grants new lease on the chunk.
  - Not increased if replica is unavailable.
- Stale replicas deleted by master in regular garbage collection.

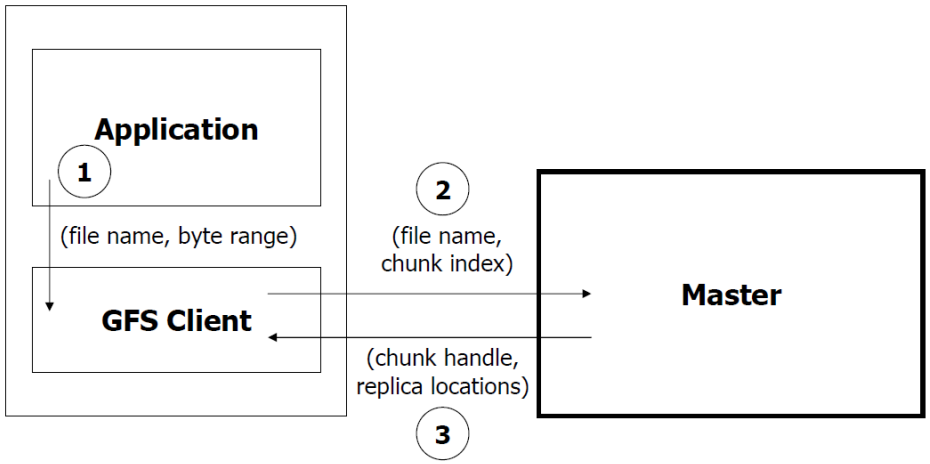
System Interactions

GFS API

- Not POSIX compliant
  - Supports only popular FS operations, and semantics are different.
- API:
  - Read operation: read
  - Update operations: write and append
  - Delete operation

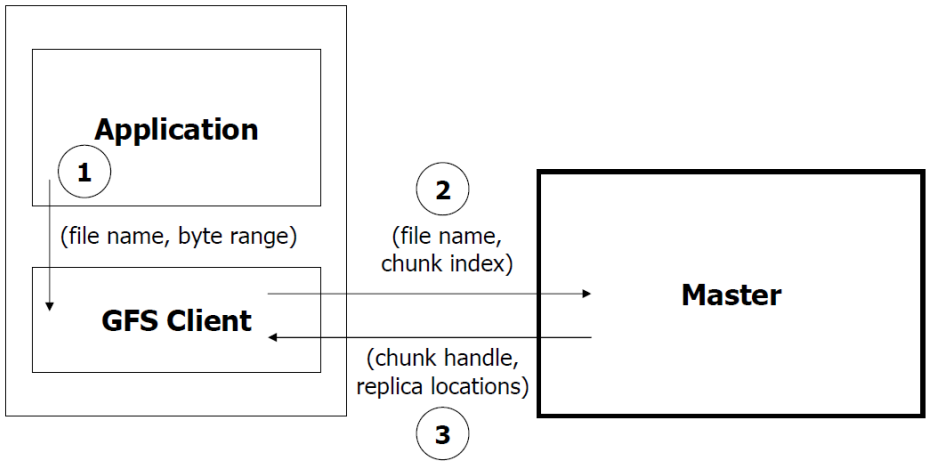
Read Operation (1/2)

- 1. Application originates the read request.



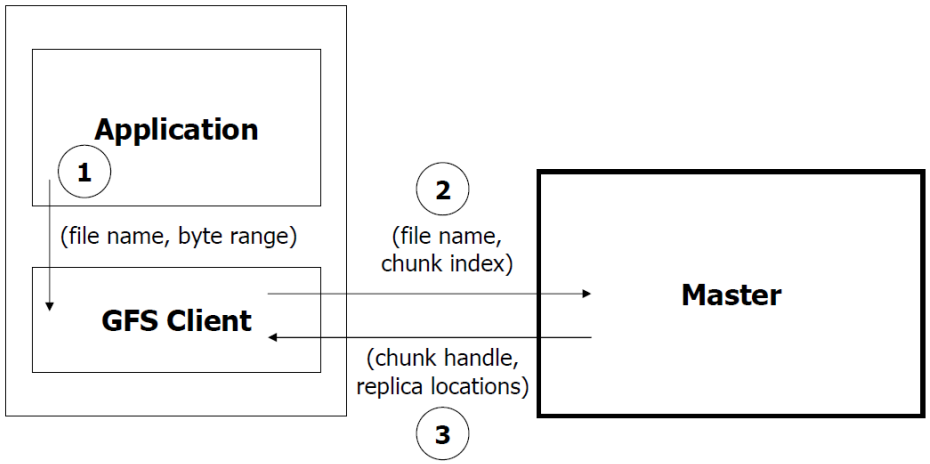
Read Operation (1/2)

- 1. Application originates the read request.
- 2. GFS client translates request and sends it to the master.



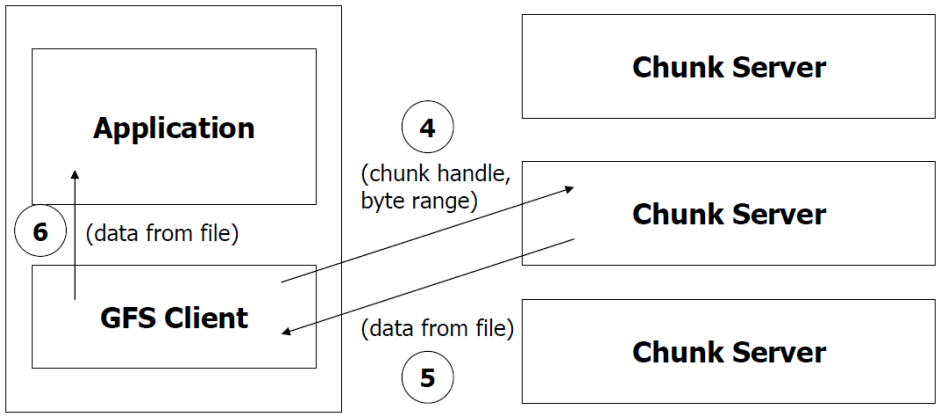
Read Operation (1/2)

- 1. Application originates the read request.
- 2. GFS client translates request and sends it to the master.
- 3. The master responds with chunk handle and replica locations.



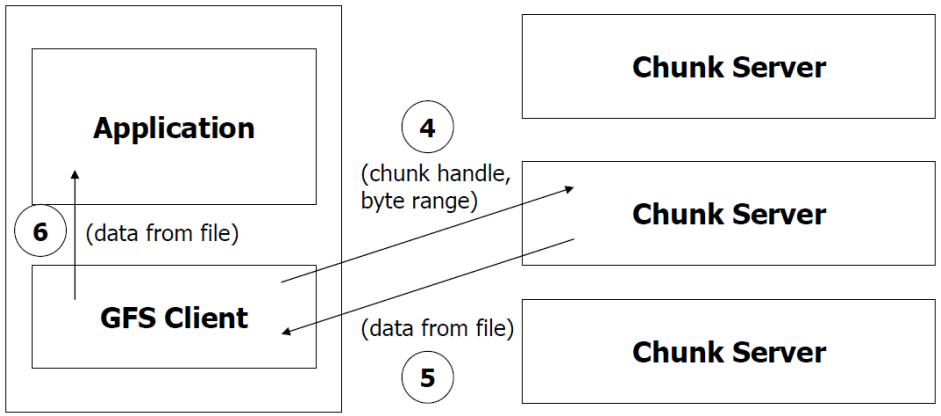
Read Operation (2/2)

- 4. The client picks a location and sends the request.



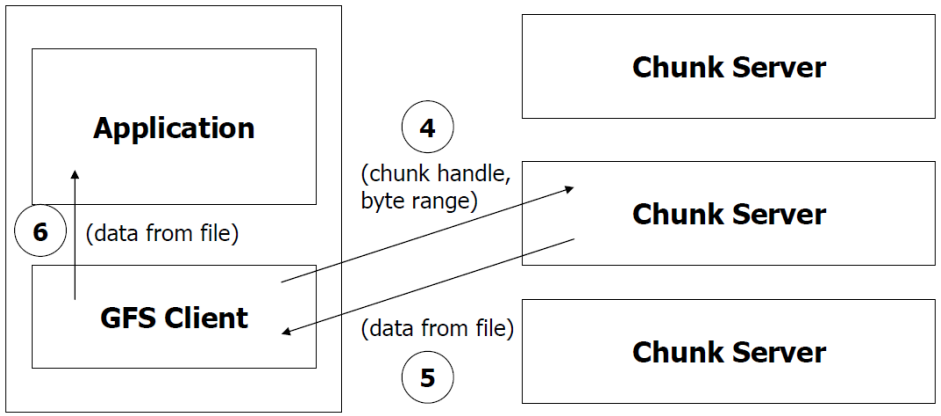
Read Operation (2/2)

- 4. The client picks a location and sends the request.
- 5. The chunk server sends requested data to the client.



Read Operation (2/2)

- 4. The client picks a location and sends the request.
- 5. The chunk server sends requested data to the client.
- 6. The client forwards the data to the application.



Update Order (1/2)

- Update (mutation): an operation that changes the contents or metadata of a chunk.

Update Order (1/2)

- Update (mutation): an operation that changes the contents or metadata of a chunk.
- For consistency, updates to each chunk must be ordered in the same way at the different chunk replicas.

Update Order (1/2)

- Update (mutation): an operation that changes the contents or metadata of a chunk.
- For consistency, updates to each chunk must be ordered in the same way at the different chunk replicas.
- Consistency means that replicas will end up with the same version of the data and not diverge.

Update Order (2/2)

- For this reason, for each chunk, one replica is designated as the primary.

Update Order (2/2)

- For this reason, for each chunk, one replica is designated as the primary.
- The other replicas are designated as secondaries

Update Order (2/2)

- For this reason, for each chunk, one replica is designated as the primary.
- The other replicas are designated as secondaries
- Primary defines the update order.

Update Order (2/2)

- For this reason, for each chunk, one replica is designated as the primary.
- The other replicas are designated as secondaries
- Primary defines the update order.
- All secondaries follows this order.

Primary Leases (1/2)

- For correctness, at any time, there needs to be one single primary for each chunk.

Primary Leases (1/2)

- For correctness, at any time, there needs to be one single primary for each chunk.
- At any time, at most one server is primary for each chunk.

Primary Leases (1/2)

- For correctness, at any time, there needs to be one single primary for each chunk.
- At any time, at most one server is primary for each chunk.
- Master selects a chunk-server and grants it lease for a chunk.

Primary Leases (2/2)

- The chunk-server holds the lease for a period T after it gets it, and behaves as primary during this period.

Primary Leases (2/2)

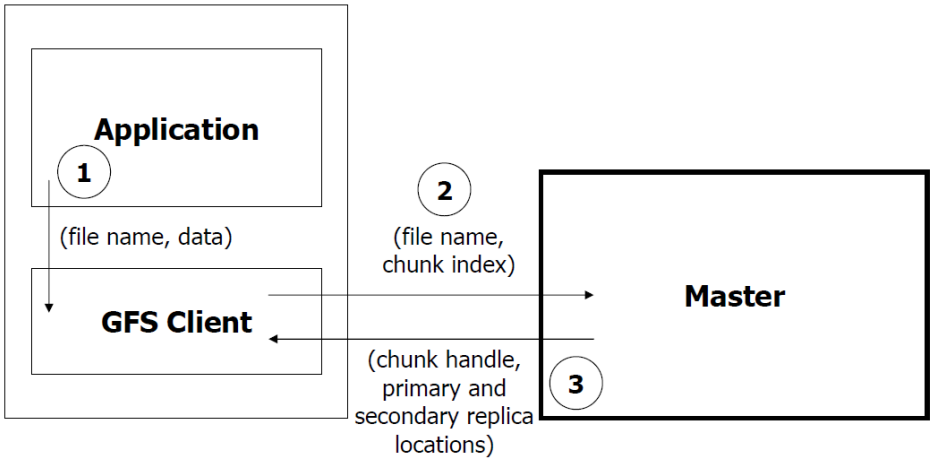
- The chunk-server holds the lease for a period T after it gets it, and behaves as primary during this period.
- The chunk-server can refresh the lease endlessly, but if the chunk-server can not successfully refresh lease from master, he stops being a primary.

Primary Leases (2/2)

- The chunk-server holds the lease for a period T after it gets it, and behaves as primary during this period.
- The chunk-server can refresh the lease endlessly, but if the chunk-server can not successfully refresh lease from master, he stops being a primary.
- If master does not hear from primary chunk-server for a period, he gives the lease to someone else.

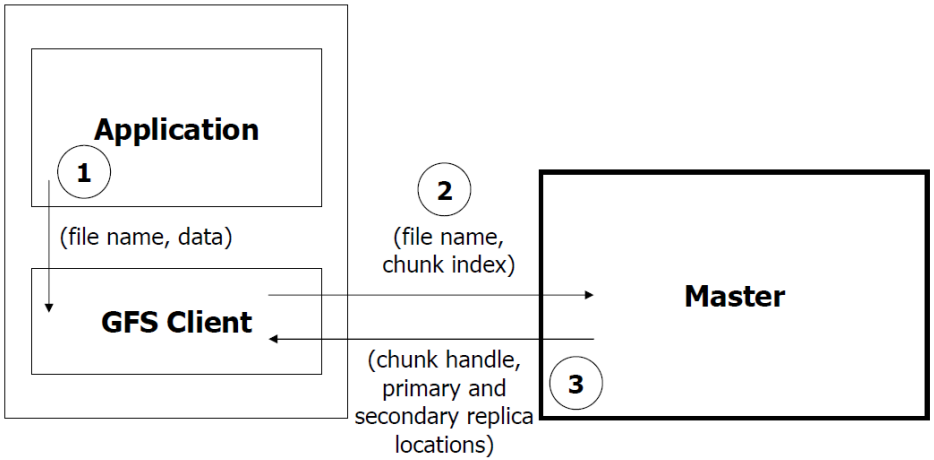
Write Operation (1/3)

- 1. Application originates the request.



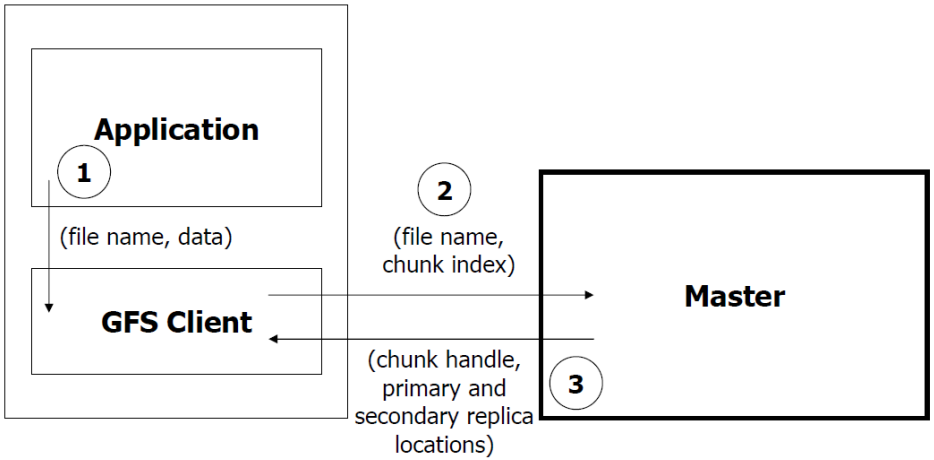
Write Operation (1/3)

- 1. Application originates the request.
- 2. The GFS client translates request and sends it to the master.



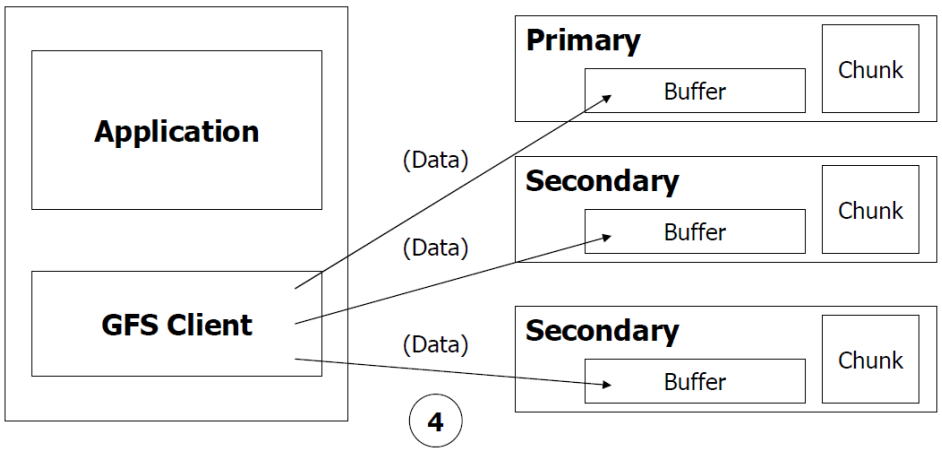
Write Operation (1/3)

- 1. Application originates the request.
- 2. The GFS client translates request and sends it to the master.
- 3. The master responds with chunk handle and replica locations.



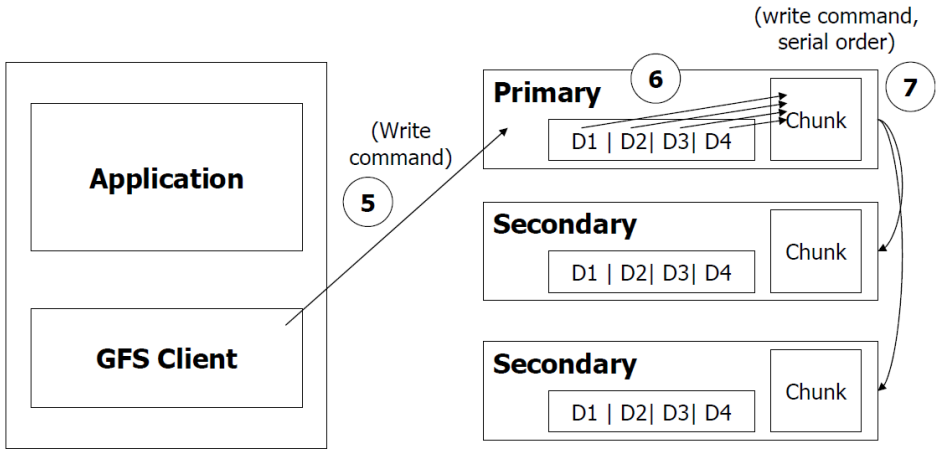
Write Operation (2/3)

- 4. The client pushes write data to all locations. Data is stored in chunk-server's internal buffers.



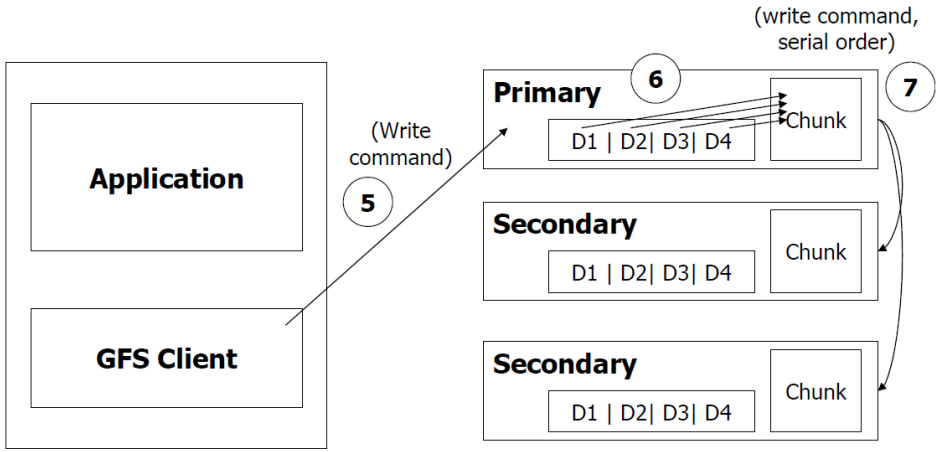
Write Operation (3/3)

- 5. The client sends write command to the primary.



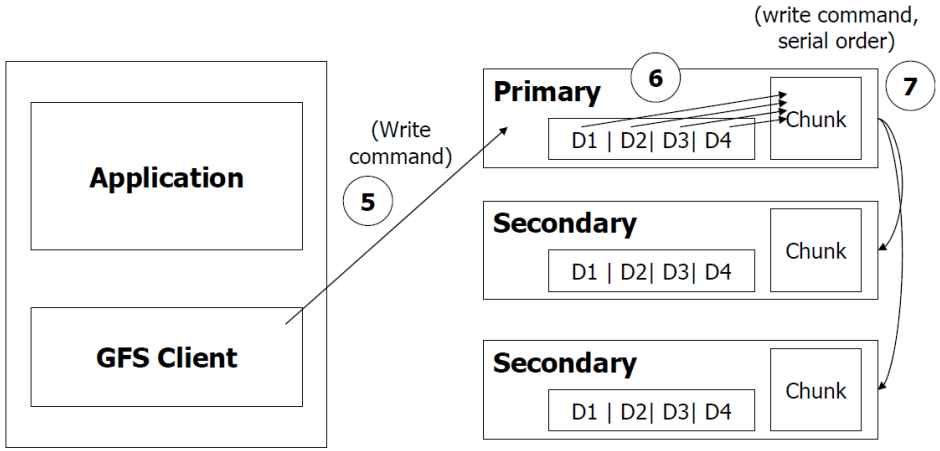
Write Operation (3/3)

- 5. The client sends write command to the primary.
- 6. The primary determines serial order for data instances in its buffer and writes the instances in that order to the chunk.



Write Operation (3/3)

- 5. The client sends write command to the primary.
- 6. The primary determines serial order for data instances in its buffer and writes the instances in that order to the chunk.
- 7. The primary sends the serial order to the secondaries and tells them to perform the write.



Write Consistency

- Primary enforces one update order across all replicas for concurrent writes.
- It also waits until a write finishes at the other replicas before it replies.

Write Consistency

- Primary enforces one update order across all replicas for concurrent writes.
- It also waits until a write finishes at the other replicas before it replies.
- Therefore:
  - We will have identical replicas.
  - But, file region may end up containing mingled fragments from different clients: e.g., writes to different chunks may be ordered differently by their different primary chunk-servers
  - Thus, writes are consistent but undefined state in GFS.

Record Append Operation (1/3)

- Operations that append data to a file.
  - Same as write, but no offset (GFS choses the offset)
- Important operation at Google
  - Merging results from multiple machines in one file.
  - Using file as producer-consumer queue.

Record Append Operation (2/3)

- 1. Application originates record append request.

Record Append Operation (2/3)

- 1. Application originates record append request.
- 2. The client translates request and sends it to the master.

Record Append Operation (2/3)

- 1. Application originates record append request.
- 2. The client translates request and sends it to the master.
- 3. The master responds with chunk handle and replica locations.

Record Append Operation (2/3)

- 1. Application originates record append request.
- 2. The client translates request and sends it to the master.
- 3. The master responds with chunk handle and replica locations.
- 4. The client pushes write data to all locations.

Record Append Operation (3/3)

- 5. The primary checks if record fits in specified chunk.

Record Append Operation (3/3)

- 5. The primary checks if record fits in specified chunk.
- 6. If record does not fit, then the primary:
  - Pads the chunk,
  - Tells secondaries to do the same,
  - And informs the client.
  - The client then retries the append with the next chunk.

Record Append Operation (3/3)

- 5. The primary checks if record fits in specified chunk.
- 6. If record does not fit, then the primary:
  - Pads the chunk,
  - Tells secondaries to do the same,
  - And informs the client.
  - The client then retries the append with the next chunk.
- 7. If record fits, then the primary:
  - Appends the record,
  - Tells secondaries to do the same,
  - Receives responses from secondaries,
  - And sends final response to the client

Delete Operation

- Meta data operation.

Delete Operation

- Meta data operation.
- Renames file to special name.

Delete Operation

- Meta data operation.
- Renames file to special name.
- After certain time, deletes the actual chunks.

Delete Operation

- Meta data operation.
- Renames file to special name.
- After certain time, deletes the actual chunks.
- Supports undelete for limited time.

Delete Operation

- Meta data operation.
- Renames file to special name.
- After certain time, deletes the actual chunks.
- Supports undelete for limited time.
- Actual lazy garbage collection.

Fault Tolerance

Fault Tolerance for Chunks

- Chunks replication (re-replication and re-balancing)
- Data integrity
  - Checksum for each chunk divided into 64KB blocks.
  - Checksum is checked every time an application reads the data.

Fault Tolerance for Chunk Server

- All chunks are versioned.
- Version number updated when a new lease is granted.
- Chunks with old versions are not served and are deleted.

Fault Tolerance for Master

- Master state replicated for reliability on multiple machines.

Fault Tolerance for Master

- Master state replicated for reliability on multiple machines.
- When master fails:
  - It can restart almost instantly.
  - A new master process is started elsewhere.

Fault Tolerance for Master

- Master state replicated for reliability on multiple machines.
- When master fails:
  - It can restart almost instantly.
  - A new master process is started elsewhere.
- Shadow (not mirror) master provides only read-only access to file system when primary master is down.

High Availability

- Fast recovery
  - Master and chunk-servers have to restore their state and start in seconds no matter how they terminated.

High Availability

- Fast recovery
  - Master and chunk-servers have to restore their state and start in seconds no matter how they terminated.
- Heartbeat messages:
  - Checking liveness of chunk-servers
  - Piggybacking garbage collection commands
  - Lease renewal


hadoop
HDFS

HDFS

- Sub-project of Apache Hadoop project
- Inspired by the Google File System
- Namenode: master
- Datanode: chunk server
- Block: chunk

HDFS Installation and Shell

HDFS Installation

- Three options
  - Local (Standalone) Mode
  - Pseudo-Distributed Mode
  - Fully-Distributed Mode

Installation - Local

- Default configuration after the download.
- Executes as a single Java process.
- Works directly with local filesystem.
- Useful for debugging.

Installation - Pseudo-Distributed (1/6)

- Still runs on a single node.
- Each daemon runs in its own Java process.
  - Namenode
  - Secondary Namenode
  - Datanode
- Configuration files:
  - `hadoop-env.sh`
  - `core-site.xml`
  - `hdfs-site.xml`

Installation - Pseudo-Distributed (2/6)

- Specify environment variables in `hadoop-env.sh`

```
export JAVA_HOME=/opt/jdk1.7.0_51
```

Installation - Pseudo-Distributed (3/6)

- Specify location of Namenode in `core-site.sh`

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://localhost:8020</value>
  <description>NameNode URI</description>
</property>
```

Installation - Pseudo-Distributed (4/6)

- Configurations of Namenode in `hdfs-site.sh`
- Path on the local filesystem where the Namenode stores the namespace and transaction logs persistently.

```
<property>
  <name>dfs.namenode.name.dir</name>
  <value>/opt/hadoop-2.2.0/hdfs/namenode</value>
  <description>description...</description>
</property>
```

Installation - Pseudo-Distributed (5/6)

- Configurations of Secondary Namenode in `hdfs-site.sh`
- Path on the local filesystem where the Secondary Namenode stores the temporary images to merge.

```
<property>
  <name>dfs.namenode.checkpoint.dir</name>
  <value>/opt/hadoop-2.2.0/hdfs/secondary</value>
  <description>description...</description>
</property>
```

Installation - Pseudo-Distributed (6/6)

- Configurations of Datanode in `hdfs-site.sh`
- Comma separated list of paths on the local filesystem of a Datanode where it should store its blocks.

```
<property>
  <name>dfs.datanode.data.dir</name>
  <value>/opt/hadoop-2.2.0/hdfs/datanode</value>
  <description>description...</description>
</property>
```

Start HDFS and Test

- Format the Namenode directory (do this only once, the first time).

```
hdfs namenode -format
```

Start HDFS and Test

- Format the Namenode directory (do this only once, the first time).

```
hdfs namenode -format
```

- Start the Namenode, Secondary namenode and Datanode daemons.

```
hadoop-daemon.sh start namenode
hadoop-daemon.sh start secondarynamenode
hadoop-daemon.sh start datanode
jps
```

Start HDFS and Test

- Format the Namenode directory (do this only once, the first time).

```
hdfs namenode -format
```

- Start the Namenode, Secondary namenode and Datanode daemons.

```
hadoop-daemon.sh start namenode
hadoop-daemon.sh start secondarynamenode
hadoop-daemon.sh start datanode
jps
```

- Verify the deamons are running:
  - Namenode: `http://localhost:50070`
  - Secondary Namenode: `http://localhost:50090`
  - Datanode: `http://localhost:50075`

HDFS Shell

```
hdfs dfs -<command> -<option> <path>
```

HDFS Shell

```
hdfs dfs -<command> -<option> <path>
```

```
hdfs dfs -ls /
hdfs dfs -ls file:///home/big
hdfs dfs -ls hdfs://localhost/
hdfs dfs -cat /dir/file.txt
hdfs dfs -cp /dir/file1 /otherDir/file2
hdfs dfs -mv /dir/file1 /dir2/file2
hdfs dfs -mkdir /newDir
hdfs dfs -put file.txt /dir/file.txt # can also use copyFromLocal
hdfs dfs -get /dir/file.txt file.txt # can also use copyToLocal
hdfs dfs -rm /dir/fileToDelete
hdfs dfs -help
```

Summary

Summary

- Google File System (GFS)
- Files and chunks
- GFS architecture: master, chunk servers, client
- GFS interactions: read and update (write and update record)
- Master operations: metadata management, replica placement and garbage collection

Questions?