# Large Scale File Systems

Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology

# 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.


[2]

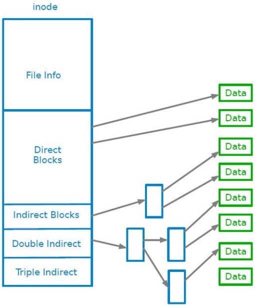
# Reminder

# 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.

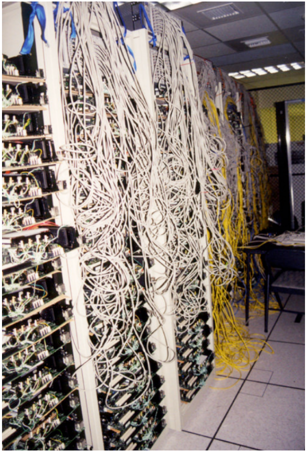
# Google File System (GFS)

# 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: $\geq 100$ MB in size.
- Large streaming reads and small random reads.
- Append to files rather than overwrite.

# Files and Chunks (1/2)

- Files are split into chunks.
- Chunks
  - Single unit of storage.
  - Transparent to user.
  - Default size: either 64MB or 128MB

# Files and Chunks (2/2)

- Why is a chunk in GFS so large?

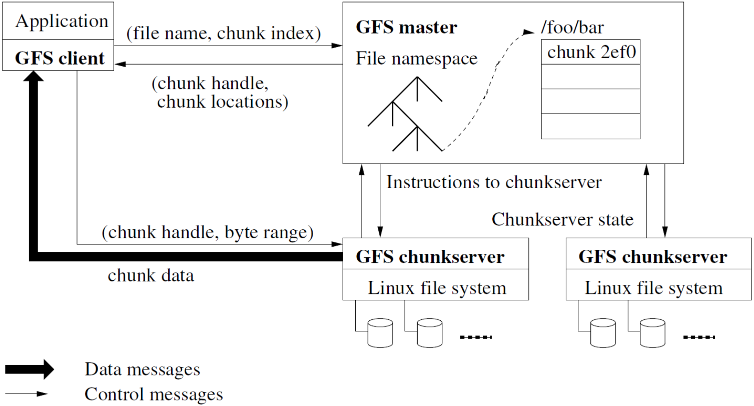
# Files and Chunks (2/2)

- 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 $\frac{\text{seek time}}{\text{transfer time}}$ small.

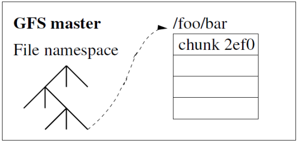
# GFS Architecture




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

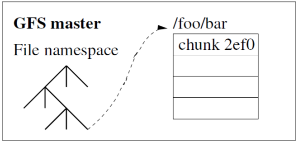
# GFS Master

- Manages file namespace operations.



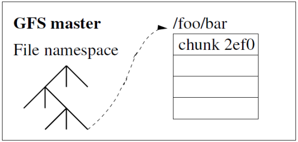
# GFS Master

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



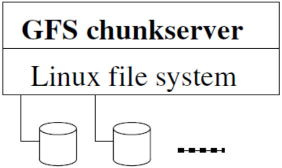
# GFS Master

- 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



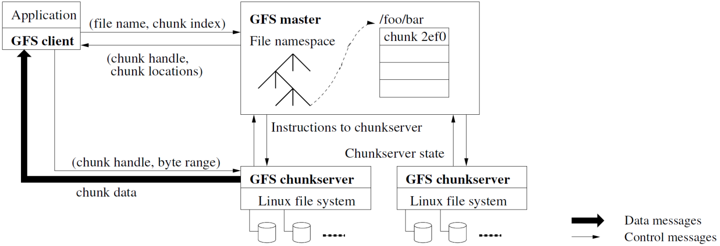
# GFS Chunk Server

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



# 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

- Represents its namespace as a lookup table mapping full pathnames to metadata.

# Namespace Management and Locking

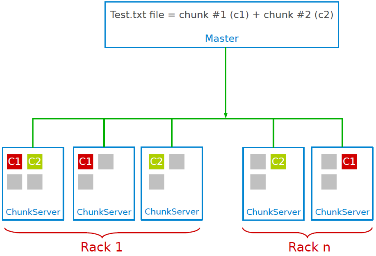
- Represents its namespace as a lookup table mapping full pathnames to metadata.
- Each master operation acquires a set of locks before it runs.
- Read lock on internal nodes, and read/write lock on the leaf.

# Namespace Management and Locking

- 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.
- Read lock on directory prevents its deletion, renaming or snapshot
- Allowed concurrent mutations in the same directory

# 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.

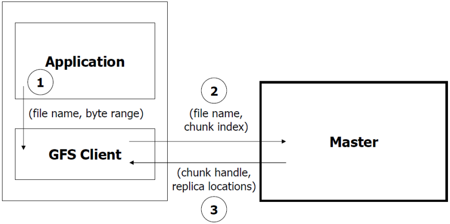
# Stale Replica Detection

- 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

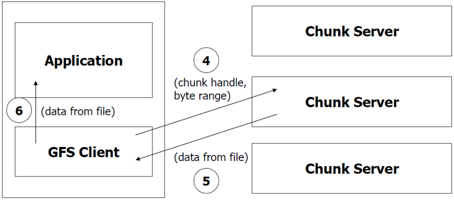
# Read Operation (1/2)

- 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.
- 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 content or metadata of a chunk.

# Update Order (1/2)

- Update (mutation): an operation that changes the content 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.
- The other replicas are designated as secondaries
- Primary defines the update order.
- All secondaries follows this order.

# Primary Leases (1/2)

- For correctness there needs to be one single primary for each chunk.

# Primary Leases (1/2)

- For correctness 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)

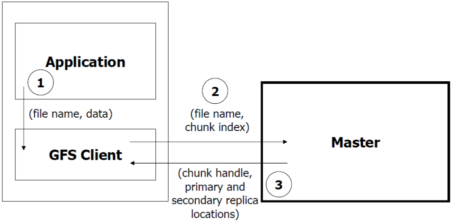
- The chunk-server holds the lease for a period $T$ after it gets it, and behaves as primary during this period.

- 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.

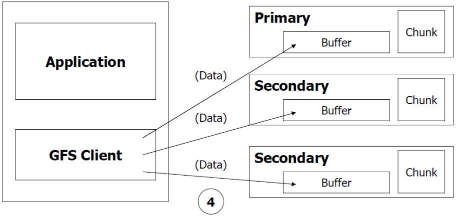
# Write Operation (1/3)

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



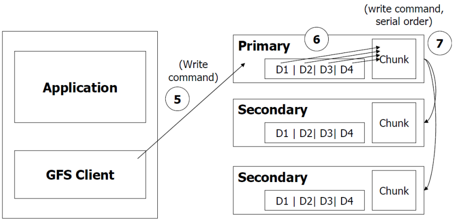
# Write Operation (2/3)

- 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.
- 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.

# Append Operation (1/2)

- 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.

# Append Operation (2/2)

- 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.
- 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.
- 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.

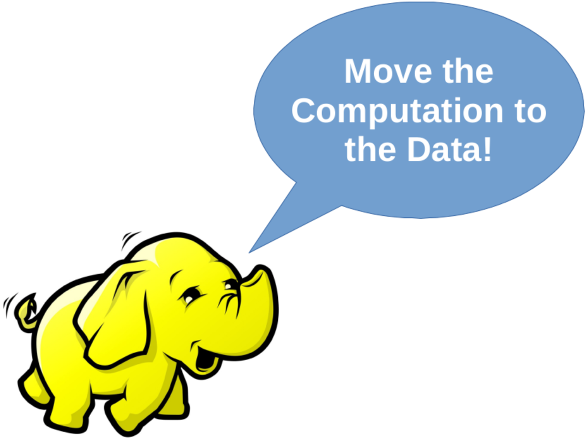
# High Availability

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

- Heartbeat messages:
  - Checking liveness of chunk-servers
  - Piggybacking garbage collection commands
  - Lease renewal

# Flat Datacenter Storage (FDS)

# Motivation and Assumptions (1/5)



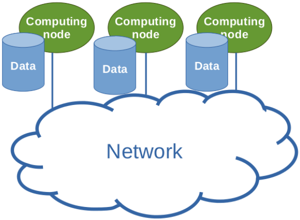

- Why move computation close to data?
  - Because remote access is slow due to oversubscription.

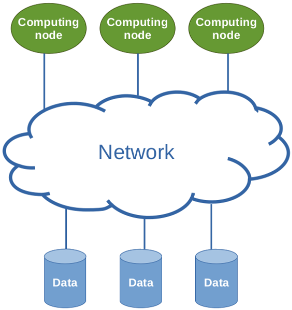
# Motivation and Assumptions (2/5)

- Locality adds complexity.
- Need to be aware of where the data is.
  - Non-trivial scheduling algorithm.
  - Moving computations around is not easy.
- Need a data-parallel programming model.

# Motivation and Assumptions (3/5)

- Datacenter networks are getting faster.
- Consequences
  - The networks are not oversubscribed.
  - Support full bisection bandwidth: no local vs. remote disk distinction.
  - Simpler work schedulers and programming models.



# Motivation and Assumptions (4/5)

- File systems like GFS manage metadata centrally.
- On every read or write, clients contact the master to get information about the location of blocks in the system.

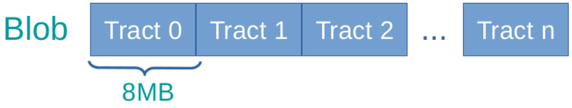
# Motivation and Assumptions (4/5)

- File systems like GFS manage metadata centrally.
- On every read or write, clients contact the master to get information about the location of blocks in the system.
  - Good visibility and control.
  - Bottleneck: use large block size
  - This makes it harder to do fine-grained load balancing like our ideal little-data computer does.

# Motivation and Assumptions (5/5)

- Let's make a digital socialism
- Flat Datacenter Storage

# Blobs and Tracts




- Data is stored in logical blobs.
  - Byte sequences with a 128-bit Global Unique Identifiers (GUID).
- Blobs are divided into constant sized units called tracts.
  - Tracts are sized, so random and sequential accesses have same throughput.
- Both tracts and blobs are mutable.

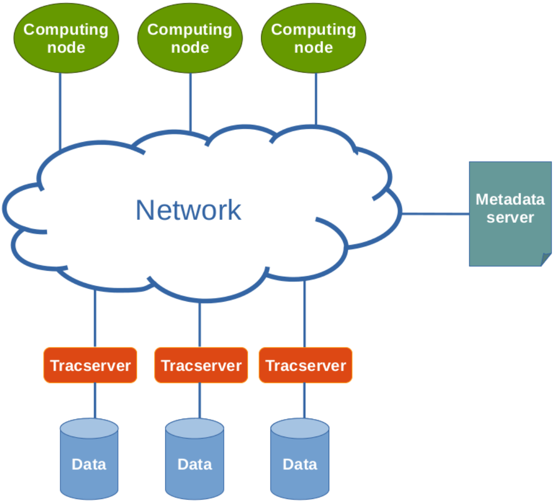
# FDS API (1/2)

```
// create a blob with the specified GUID
CreateBlob(GUID, &blobHandle, doneCallbackFunction);

// write 8MB from buf to track 0 of the blob
blobHandle->WriteTrack(0, buf, doneCallbackFunction);

// read track 2 of blob into buf
blobHandle->ReadTrack(2, buf, doneCallbackFunction);
```

# FDS API (2/2)

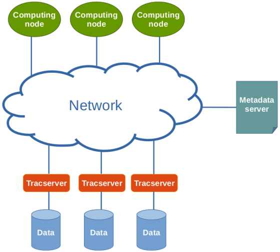
- Reads and writes are atomic.
- Reads and writes not guaranteed to appear in the order they are issued.
- API is non-blocking.
  - Helps the performance: many requests can be issued in parallel, and FDS can pipeline disk reads with network transfers.

# FDS Architecture



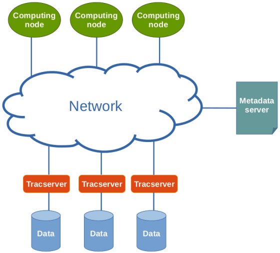
# Trackserver

- Every disk is managed by a process called a tractserver.
- Trackservers accept commands from the network, e.g., `ReadTrack` and `WriteTrack`.
- They do not use file systems.
  - They lay out tracts directly to disk by using the raw disk interface.



# Metadata Server

- Metadata server coordinates the cluster and helps clients rendezvous with tractservers.
- It collects a list of active tractservers and distribute it to clients.
- This list is called the tract locator table (TLT).
- Clients can retrieve the TLT from the metadata server once, then never contact the metadata server again.



# Track Locator Table (1/2)

- TLT contains the address of the tractserver(s) responsible for tracts.
- Clients use the blob's GUID (g) and the tract number (i) to select an entry in the TLT: tract locator

```
TractLocator = (Hash(g) + i) mod TLT Length
```

| Locator | Disk 1 | Disk 2 | Disk 3 |
|---|---|---|---|
| 0 | A | B | C |
| 1 | A | D | F |
| 2 | A | C | G |
| 3 | D | E | G |
| 4 | B | C | F |
| ... | ... | ... | ... |
| 1,526 | LM | TH | JE |

# Track Locator Table (2/2)

- The only time the TLT changes is when a disk fails or is added.
- Reads and writes do not change the TLT.
- In a system with more than one replica, reads go to one replica at random, and writes go to all of them.

# Per-Blob Metadata

- Per-blob metadata: blob's length and permission bits.
- Stored in tract -1 of each blob.
- The trackserver is responsible for the blob metadata tract.
- Newly created blobs have a length of zero, and applications must extend a blob before writing. The extend operation is atomic.

# Fault Tolerance

# Replication

- Replicate data to improve durability and availability.
- When a disk fails, redundant copies of the lost data are used to restore the data to full replication.

# Replication

- Replicate data to improve durability and availability.
- When a disk fails, redundant copies of the lost data are used to restore the data to full replication.
- Writes a tract: the client sends the write to every tractserver it contains.
  - Applications are notified that their writes have completed only after the client library receives write ack from all replicas.
- Reads a tract: the client selects a single tractserver at random.

# Failure Recovery (1/2)

- **Step 1**: Tractservers send heartbeat messages to the metadata server. When the metadata server detects a tractserver timeout, it declares the tractserver dead.
- **Step 2**: invalidates the current TLT by incrementing the version number of each row in which the failed tractserver appears.
- **Step 3**: picks random tractservers to fill in the empty spaces in the TLT where the dead tractserver appeared.

| Row | Version | Replica 1 | Replica 2 | Replica 3 |
|---|---|---|---|---|
| 1 | 8 | A | F | B |
| 2 | 17 | B | C | L |
| 3 | 324 | E | D | G |
| 4 | 3 | T | A | H |
| 5 | 456 | F | B | G |
| 6 | 723 | G | E | B |
| 7 | 235 | D | V | C |
| 8 | 312 | H | E | F |

| Row | Version | Replica 1 | Replica 2 | Replica 3 |
|---|---|---|---|---|
| 1 | 9 | A | F | H |
| 2 | 18 | I | C | L |
| 3 | 324 | E | D | G |
| 4 | 3 | T | A | H |
| 5 | 457 | F | C | G |
| 6 | 724 | G | E | A |
| 7 | 235 | D | V | C |
| 8 | 312 | H | E | F |

# Failure Recovery (2/2)

- Step 4: sends updated TLT assignments to every server affected by the changes.

- Step 5: waits for each tractserver to ack the new TLT assignments, and then begins to give out the new TLT to clients when queried for it.

| Row | Version | Replica 1 | Replica 2 | Replica 3 |
|---|---|---|---|---|
| 1 | 8 | A | F | B |
| 2 | 17 | B | C | L |
| 3 | 324 | E | D | G |
| 4 | 3 | T | A | H |
| 5 | 456 | F | B | G |
| 6 | 723 | G | E | B |
| 7 | 235 | D | V | C |
| 8 | 312 | H | E | F |

| Row | Version | Replica 1 | Replica 2 | Replica 3 |
|---|---|---|---|---|
| 1 | 9 | A | F | H |
| 2 | 18 | I | C | L |
| 3 | 324 | E | D | G |
| 4 | 3 | T | A | H |
| 5 | 457 | F | C | G |
| 6 | 724 | G | E | A |
| 7 | 235 | D | V | C |
| 8 | 312 | H | E | F |

# 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

# Summary

- Flat Datacenter Storage (FDS)
- Blobs and tracks
- FDS architecture: Metadata server, trackservers, TLT
- FDS interactions: using GUID and track number
- Replication and failure recovery

Questions?