What does it do on a high level:
- Provide a (huge) file system suitable for the given assumptions below, optimized for sequential reads and appends

Background / Considerations / Assumptions:
- Component failures are norm: lots of inexpensive commodity hardware
- Files are huge: GBs, TBs are norm, impacts IO pattern and block sizes
- Most files are mutated by appending new data rather than overwriting existing. Random writes are extremely rare. Once written, data is usually read sequentially, decides which operation to optimize / guarantee atomicity for (pattern: large streaming read, small random read, large appends, concurrent appends)
- High sustained bandwidth more important than latency

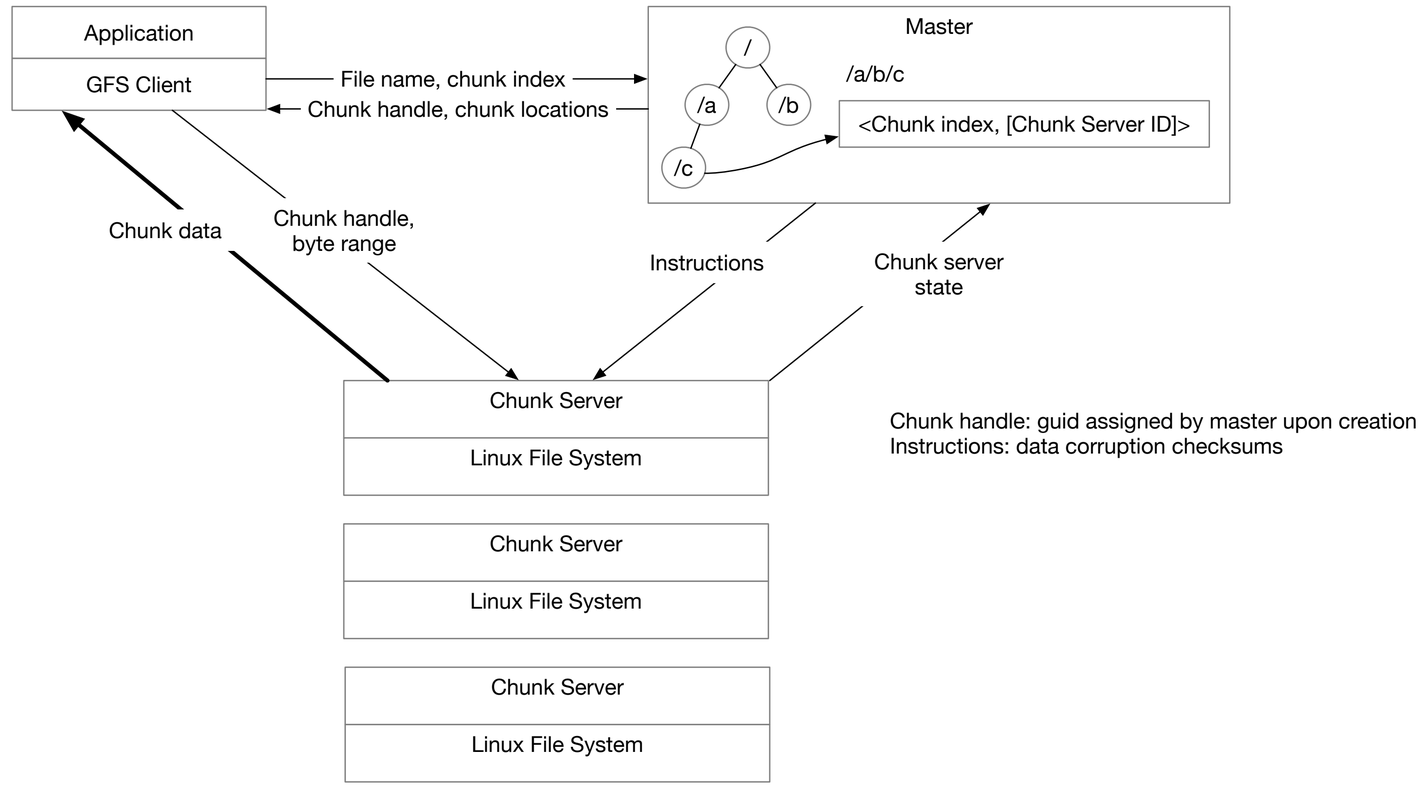
Operations:
Create, delete, open, close, read, write, snapshot, record (concurrent append)

Architecture:
- Single master
  - Stores metadata, handles chunk management (gc, migration), collects heartbeat from chunk servers
  - Rationale: simplify design
  - Requirement: minimum involvement
- Chunk servers: stores file content (Linux file system), each chunk is replicated x3
- Chunk content is not cached, <chunk index, [chunk server ID]> is cached by client

Details:
- 64MB chunk size, each chunk is a plain Linux file
  - Rationale: reduce master interaction, reduce metadata size, reduce number of chunk servers talked to
  - Downside: small file fragmentation and hotspot
- Metadata: namespace, file -> chunks, chunk -> locations. All in memory. First two are kept persistent using a log, log is replicated to remote machines. Master asks chunk servers about the third upon startup, when chunk server join, and in periodic heartbeats
  - Rationale: use simplest design (poll vs persist) for data that change often, persist only the critical
  - When updating persisted info, tell client it's done only after remote backups are successfully made as well
  - Master replays to log to recover, and checkpoints the log (B-tree) to prevent it from growing oversize

Consistency:
- Relaxed (possible for clients to read stale data, consistent when concurrent appends but not concurrent arbitrary writes)
- Namespace mutation is atomic and globally ordered (master lock)
- Data mutation
  - Consistency: all clients see the same data
  - Defined: consistent and what clients see reflects mutations in its entirety
  - Serial successful *write*: defined
  - Concurrent successful *write*: consistent, undefined (mingled fragments)
  - Failed *write*: inconsistent
- Record append: different from writes at arbitrary offsets, record append causes data to be appended atomically at least once at an offset of GFS's choosing
  - Concurrent successful *record append*: defined
- GFS achieves defined serial mutation by applying mutations to a chunk in the same order on all its replicas and using chunk version numbers to detect any stale replica (due to missed mutations when chunk server was down)
- Although master does not return stale chunk servers to the client, the client may still read from one due to cached chunk locations, however this is rare (cache lifetime) and when client retries with master the data won't be stale
- Because of this consistency semantic: applications append, checkpoints, and write self-validating self-identifying records; Reader deals with potential duplicates and padding due to concurrent appends using checksums, or expect unique ids in written entries and discard if sees duplicates

Interactions:
- Read: client asks master for where a chunk is, then reads from closest copy
- Write



Chunk handle: guid assigned by master upon creation
Instructions: data corruption checksums