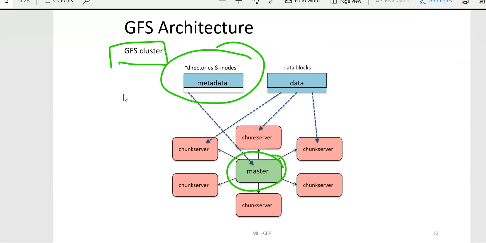
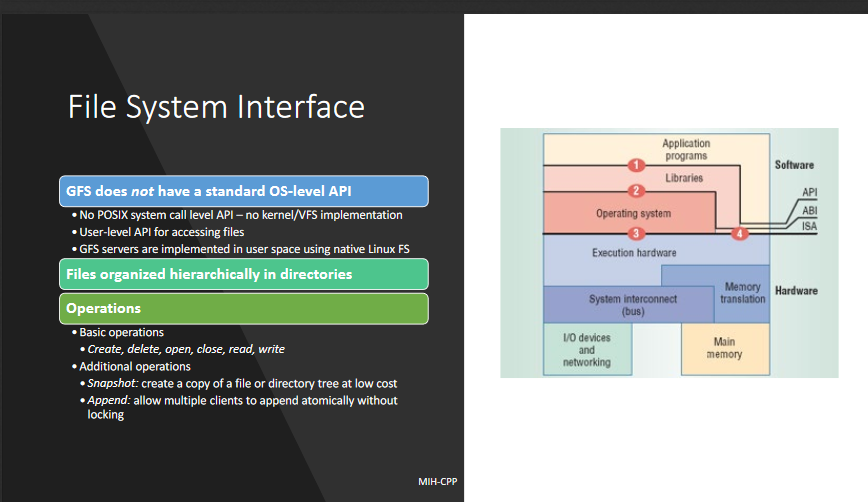
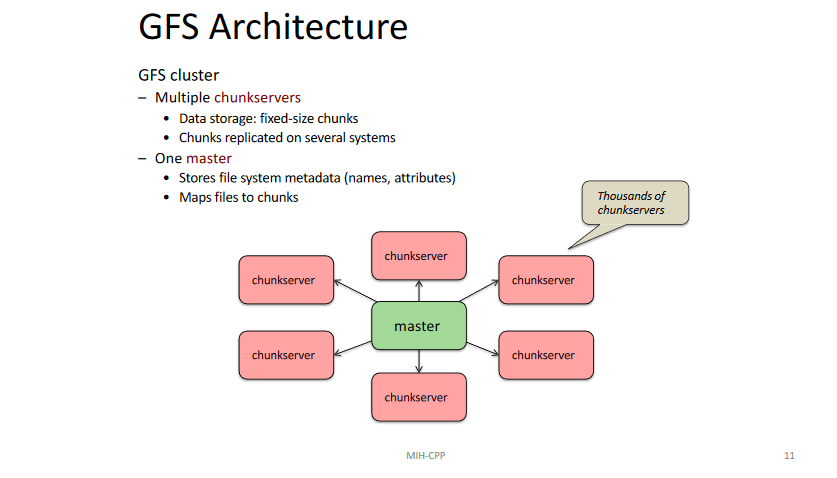
Google File System (is a parallel system)

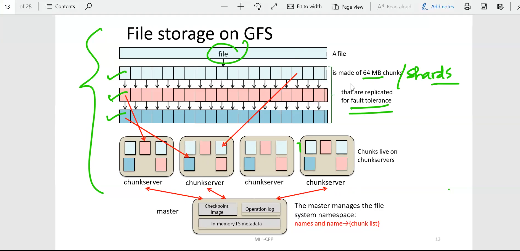
Network Attached Storage

* File-level computer data storage connected to IP network
  + Provides data access to dif group of clients
  + Removes responsibility of file serving from other servers on network.
  + Can’t provide the parallelism that we NEED for HUGE inputs of data
* Client Server File Systems
  + Central Servers
    - Point of congestion, single point of failure
  + Alleviate wit replication and client cahing
    - Coda, tokens
    - Limited replication can lead to congestion
  + File data is centralized
    - File server stores all data from file (not split across servers)
    - If replication in place, client downloads all data for file from one server
  + File sizes limited to capacity available on server
    - What if need 1000 TB file? What then?? BIG DATAAAAA
* Parallel File Systems
  + File data can span multiple servers
    - So metadata can be on separate servers s
      * metadata is info about file like name, access pemissions, timestamps, file size, and locations of data blocks
    - 2 main components: file data can span multiple server, and metadata on separate server
    - We want to design such a way so info about file can be found sep from actual location instead of going into storage and looking for it.
* GFS GOALS
  + To be scalable distributed file system
  + Designed for large data-intensive applications
  + Fault-tolerant, runs on commodity HW
  + High performance to large num of clients
* GFS Design Assumptions
  + Assumptions for conventional file systems don’t work
    - Ex: “most files are small, lots have short lifetime”
  + Component failures are the NORM not an exception
    - File sys has THOUSANDS of storage machine
    - Some will not work at any given time
  + Files are HUGE, multiple-TB are the norm
    - I/O operations and block choices affected
  + FILE ACESS:
    - Most files appended not overwritten
      * Random writes within file almost NEVER done
      * Once created, files usually read, often sequentially
    - Workload mostly consists of:
      * Reads (large streaming reads, small random reads 🡨 these dominate)
      * Large appends
      * Hundreds of processes may append to file at the same time
    - GFS stores a modest # of files for its scale, a few million about
* BASIC IDEA
  + Use separate servers to store metadata
    - Like server, block number, that identify which blocks on which serves hold file data
    - Need more bandwidth for data access than metadata access
    - Metadata small, file data can be HUGE
  + Use large logical blocks
    - Most normal file systems optimized for small files
    - GFS uses HUUUGE blocks 1000x larger
    - List of blocks that make up file becomes easier to manage
  + Replicate data
    - Expect some servers to be down, so store copies of data blocks on multiple



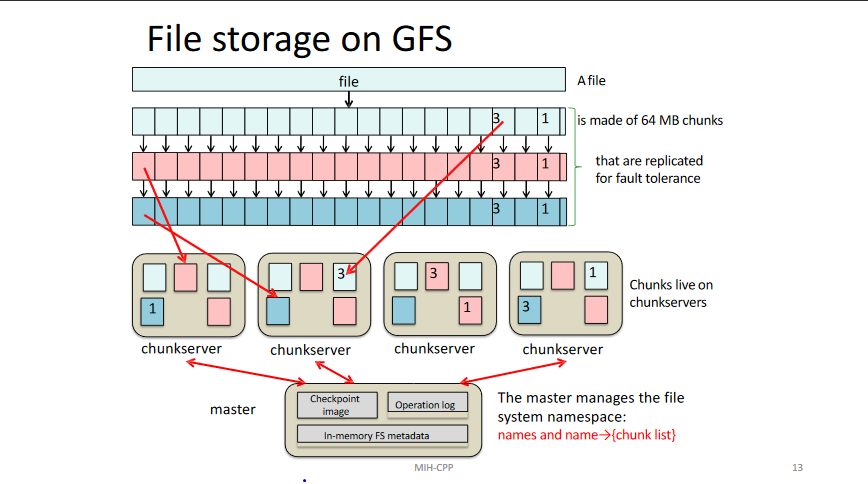
In pic below, so for GFS a file broken up into 64 MB shards, then copied, each copy on own server to help with fault tolerance.

Reason: help with bottleneck of traditional NAs by splitting data and metadata

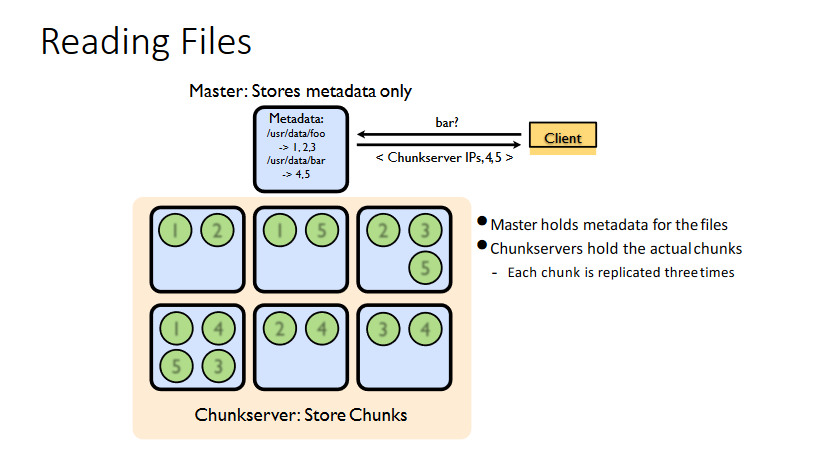
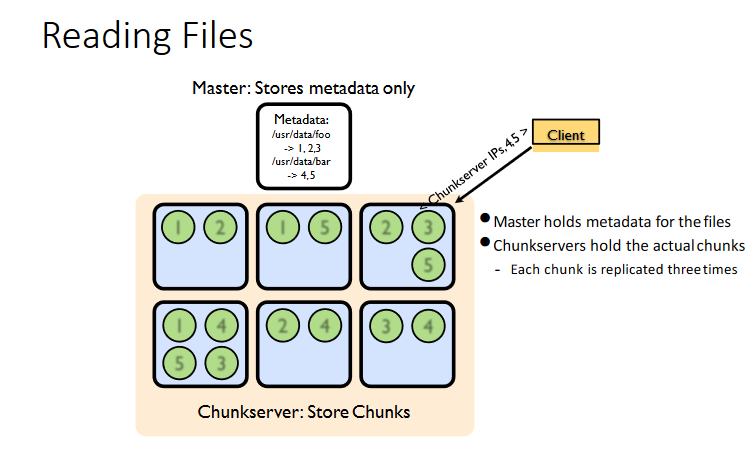
In clusters, HUGE heat issues could easily break so that’s why have copy

Made for FAILURE it is A FEATURE cause there is just SOO MUCH

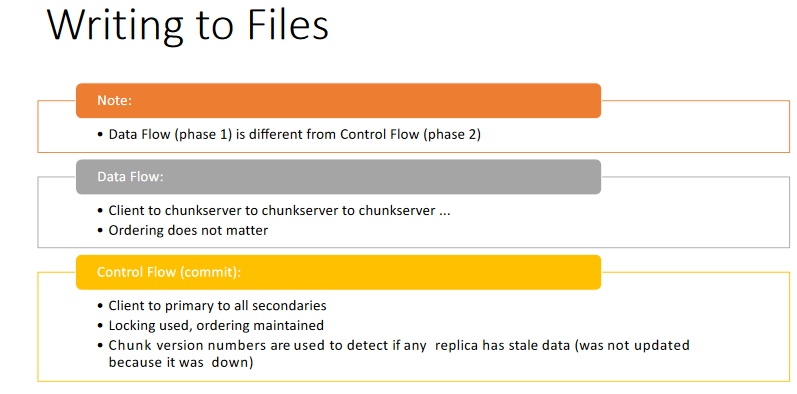
Each copy dif channel servers so if one dies BAM have another option, once one dies master COPIES and puts it on another etc, AT ANY TIME ALWAYS 3 COPIES

GFS ­­­­architecture

* Typically made of thousands of chunk servers
  + Data storage consists of fixed size chunks
  + Chunks replicated on several systems
  + Chunk size typically 64 MB default
    - 32 bit checksum with each chunk
  + Chunk Handler
    - Globally unique 64 bit number
    - Assigned by master when created
  + Where chunks stored?
    - Local disk as linux files
  + Each chunk replicated across multiple nodes
    - Three replicas (default)
    - More replicas for popular files to avoid hotspots
* One master
  + Store file system smetadata like names and attributes
  + Maps files to chunks
  + Maintains all file system metadata
    - Like namespace, access control info, filename to chunk mapping…
    - Manages chunk leases (locks), garbage colletion chunk migration
    - Oeriodically comm with all nodes via heartbeat msg. get state and send cmd
    - Fault Tolerance- operation log repl on mult mcahines
    - New master can be started if master fails
* WHY LARGE CHUNKS?
  + With default size 64 MB, compared to linux which is typically 4kb
  + Reduces need for freq comm with master to get chunk location info – one query can give info on location of lots of bytes of data
  + Clients can easily cache info to refer to all data of large files
  + Cached data has timeouts to reduce possibility of reading stale data
  + Large chunks make feasible to keep TCP connection open to chunkserver for extended time
  + Master stores 64 bytes of metadata for each 64 MB chunk
* MaSTER DESIGN
  + All metadata stored in master memory so super fast access
  + Namespaces and name-to-chunk\_list maps
    - Stored in memory
    - Persist in operation log on disk
      * Replicated onto remote machines for backup
    - Operation logs
      * Similar to journal, all operations logged
      * Periodic checkpoints to avoid playin back entire log
    - Master does not store chunk locations persistently
      * Queried from all chunservers: avoids consistency problems
* HOW DO CLIENTS INTERACT?
  + GFS Client code linked into each application
    - No Os level API, you have to use library
    - Interact with master for metadata operations
    - Interacts directly with chunk servers for file data
  + Clients cache metadata
    - Location of file chunks cached
    - No data cached by client or chunkservers
  + READING FILES?
    - 1) Contact master Node
    - 2) Get file’s metadata: list chunk handlers
  + Get location of each of chunk handles
    - Multiple replicated chunkservers per chunk
  + Contact any available chunkserver for chunk data



* WRITING FILES
  + Less frequent then reading
  + Master grants a chunk lease to one of replicas
    - This replica will be primary replica chunkserver Primary can request lease extensions
    - If needed Master can increase chunk version number and inform replicas
  + STEPS: PHASE ONE

1. Send data (deliver data but don’t write to file)
2. Client given list of replicas
   1. Identify primary and secondaries
3. Client delivers data to closest replica
   1. Pipelining forward
4. Chunkservers store this data in cache, not part of file
5. GOAL:
   1. Maximize bandwidth with pipelining
   2. Minimize latency by forwarding data as soon as received
   * PHASE TWO WRITE DATA (commit it to file)
     + Client waits for replicas to acknowledge receiving data
     + Send write request to primary
       - Primary responsible for serialization of writes (applying then forwarding)
       - Once all ack, have been received primary acknowledges the client.