14-736: DISTRIBUTED SYSTEMS

LECTURE 14 * SPRING 2019 * KESDEN

DIVERSION: RAIDS

- "Redundant Array of Inexpensive Disks"
  - Original goal: Use cheap disks for robust bulk storage
- "Redundant Array of Independent Disks"
  - Disks aren't really differentiated by reliability anymore
  - Goal: More robust
  - Goal: Larger volume
  - Goal: Higher throughput
- Big idea: Spread data over multiple drives in parallel to get higher throughput while using parity for robustness.

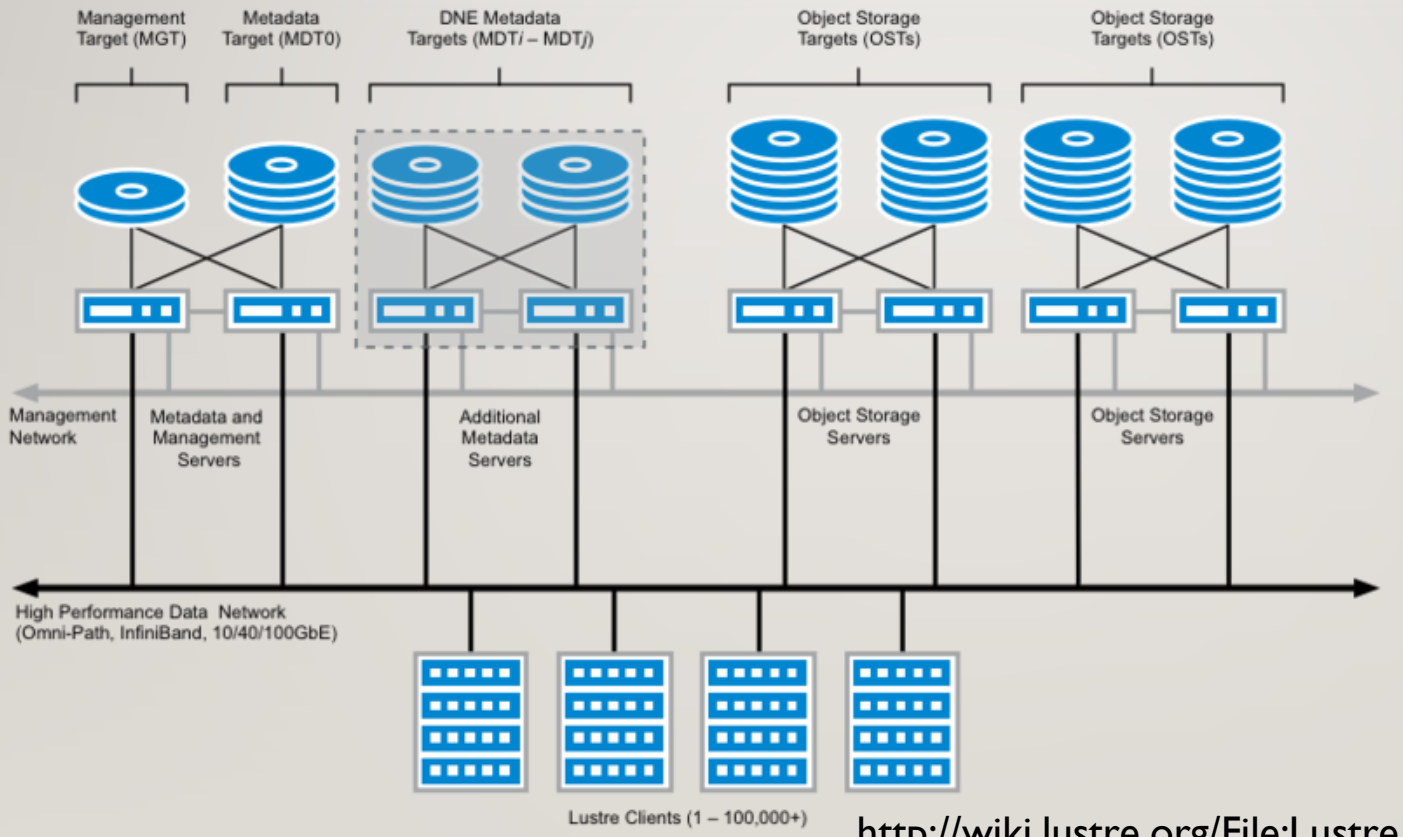
DIVERSION: RAID LEVELS

- Raid 0: Stripe data across drives for improved throughput
  - No extra redundancy, elevated risk
- Raid 1: Mirroring, parallel data to multiple devices for robustness
  - No extra throughput
- Raid 2: Use Hamming codes for parity
  - Requires $\log_2$ parity bits
  - Really expensive
- Raid 3: Bitwise parity on parity disk.
  - Requires only one parity disk for N storage disks.
  - Bitwise parity is slow, dedicated parity disk is bottleneck
- Raid 4: Blockwise parity improves performance
- Raid 5: Rotating parity block among disks relieves bottleneck
- Raid 6: Raid5 + dual parity.
  - Supports up to 2 HDD failures
  - Slow rebuilds
- Raid 10: Raid 1 + 0
- Raid 50: Raid 5 + 0

LUSTRE: *LINUX CLUSTER*

- Developed by Peter Braam, at the time a researcher at CMU
  - Cluster File Systems, Inc. → Sun → Oracle → Open
- Used for high-performance clusters
- Divides and conquers like RAIDs for throughput
  - Pairings for reliability

LUSTRE: *LINUX CLUSTER*



http://wiki.lustre.org/File:Lustre_File_System_Overview_(DNE)_lowres_v1.png

MOGILEFS

- Think about sharing sites, e.g. video, photo, etc
  - Uploads
  - No editing
  - Whole file delivery, no random access
  - Managed by software, not humans
    - No need for hierarchical namespace
  - Protections enforced by application, not FS
  - Goal: Fast delivery to clients

MOGILEFS

- Replicated storage
- *Class* determines number of replicas
- HTTP + MySQL for portability
- *Namespaces* vs directory tree
  - Think albums, etc.
- Portable:
  - User-level
  - Perl implementation
  - Any lower-level file system

HDFS Architecture

Based Upon: http://hadoop.apache.org/docs/r3.0.0-alpha1/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

HDFS ARCHITECTURE

GREGORY KESDEN, CSE-291 (STORAGE SYSTEMS) FALL 2017

BASED UPON: HTTP://HADOOP.APACHE.ORG/DOCS/R3.0.0-ALPHA1/HADOOP-PROJECT-DIST/HADOOP-HDFS/HDFSDESIGN.HTML

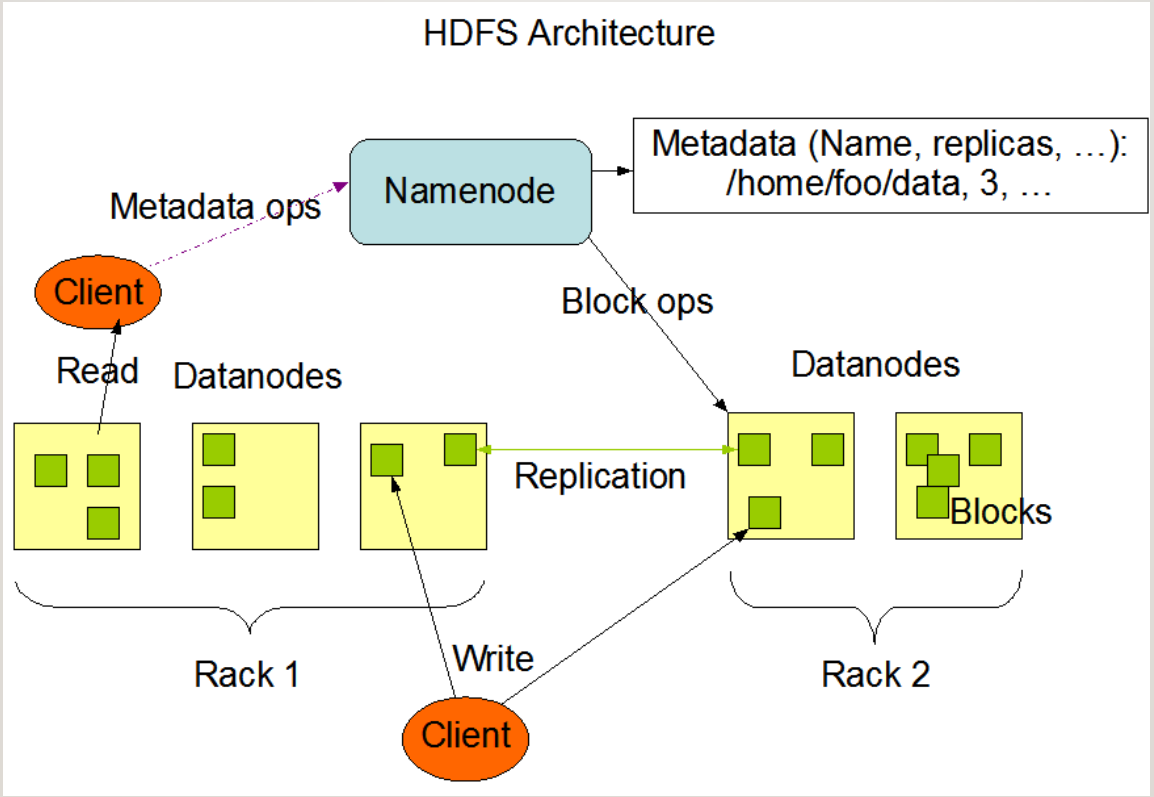
ASSUMPTIONS

- At scale, hardware failure is the norm, not the exception
  - Continued availability via quick detection and work-around, and eventual automatic rull recovery is key
- Applications stream data for batch processing
  - Not designed for random access, editing, interactive use, etc
  - Emphasis is on throughput, not latency
- Large data sets
  - Tens of millions of files many terabytes per instance

ASSUMPTIONS, *CONTINUED*

- Simple Coherency Model = Lower overhead, higher throughput
  - Write Once, Read Many (WORM)
  - Gets rid of most concurrency control and resulting need for slow, blocking coordination
- "Moving computation is cheaper than moving data"
  - The data is huge, the network is relatively slow, and the computation per unit of data is small.
  - Moving (Migration) may not be necessary – mostly just placement of computation
- Portability, even across heterogeneous infrastructure
  - At scale, things can be different, fundamentally, or as updates roll-out

OVERALL ARCHITECTURE



NAMENODE

- Master-slave architecture
- 1x NameNode (coordinator)
  - Manages name space, coordinates for clients
  - Directory lookups and changes
  - Block to DataNode mappings
- Files are composed of blocks
  - Blocks are stored by DataNodes
- Note: User data never comes to or from a NameNode.
  - The NameNode just coordinates

DATANODE

- Many DataNodes (participants)
  - One per node in the cluster. Represent the node to the NameNode
  - Manage storage attached to node
  - Handles read(), write() requests, etc for clients
  - Store blocks as per NameNode
    - Create and Delete blocks, Replicate Blocks

NAMESPACE

- Hierarchical name space
  - Directories, subdirectories, and files
- Managed by NameNode
- Maybe not needed, but low overhead
  - Files are huge and processed in entirety
  - Name to block lookups are rare
  - Remember, model is streaming of large files for processing
  - Throughput, not latency, is optimized

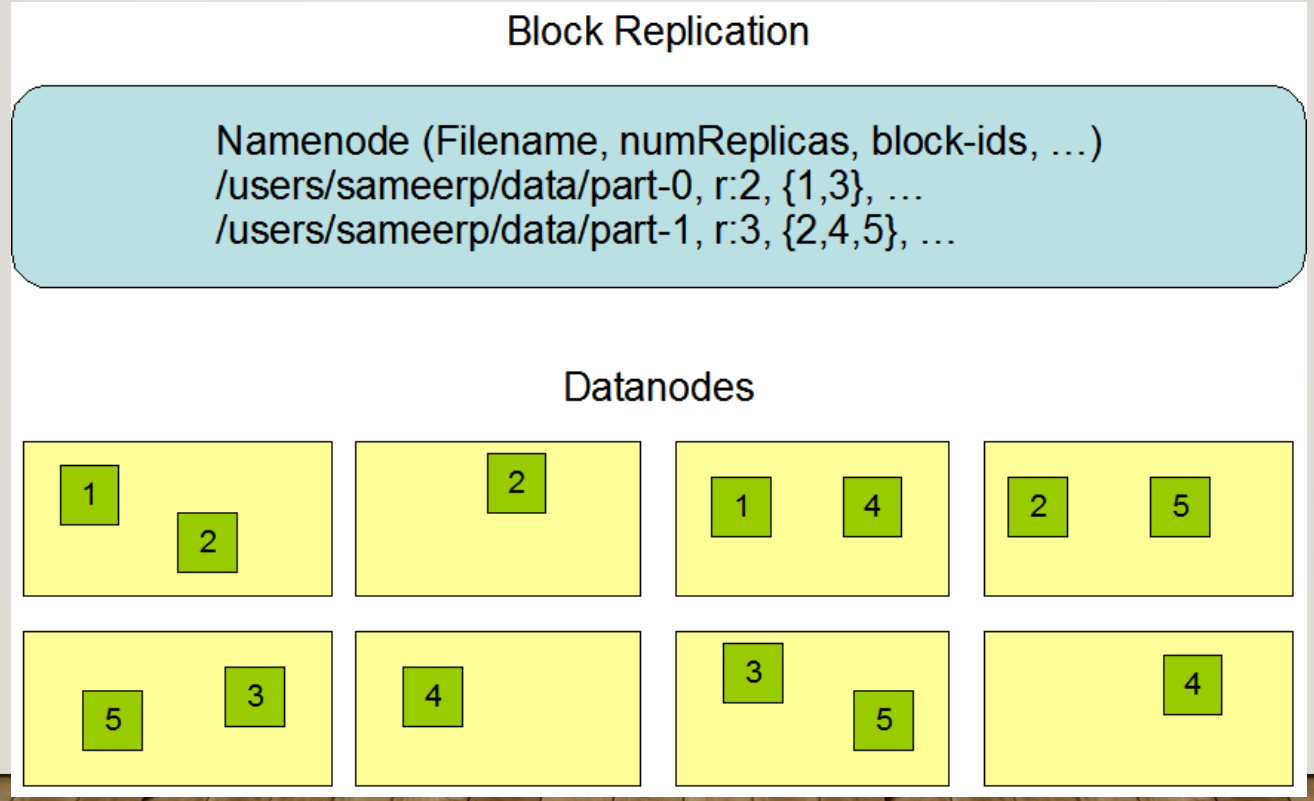
ACCESS MODEL

- (Just to be really clear)
- Read anywhere
  - Streaming is in parallel across blocks across DataNodes
- Write only at end (append)
- Delete whole file (rare)
- No edit/random write, etc

REPLICATION

- Blocks are replicated by default
  - Blocks are all same size (except tail)
  - Fault tolerance
  - Opportunities for parallelism
- NameNode managed replication
  - Based upon heartbeats, block reports (per dataNode report of available blocks), and replication factor for file (per file metadata)

REPLICATION



LOCATION AWARENESS

- Site + 3-Tier Model is default

REPLICA PLACEMENT AND SELECTION

- Assume bandwidth within rack greater than outside of rack
- Default placement
  - 2 nodes on same rack, one different rack (Beyond 3? Random, below replicas/rack limit)
  - Fault tolerance, parallelism, lower network overhead than spreading farther
- Read from closest replica (rack, site, global)

FILESYSTEM METADATA PERSISTENCE

- EditLog keeps all metadata changes.
  - Stored in local host FS
- FSImage keeps all FS metadata
  - Also stored in local host FS
- FSImage kept in memory for use
  - Periodically (time interval, operation count), merges in changes and checkpoints
  - Can truncate EditLog via checkpoint
- Multiple copies of files can be kept for robustness
  - Kept in sync
  - Slows down, but okay given infrequency of metadata changes.

FAILURE OF DATANODES

- Disk Failure, Node Failure, Partitioning
  - Detect via heartbeats (long delay, by default), blockmaps, etc
  - Re-Replicate
- Corruption
  - Detectable by client via checksums
  - Client can determine what to do (nothing is an option)
- Metadata

DATABLOCKS, STAGING

- Data blocks are large to minimize overhead for large files
- Staging
  - Initial creation and writes are cached locally and delayed, request goes to NameNode when 1st chunk is full.
  - Local caching is intended to support use of memory hierarchy and throughput needed for streaming. Don't want to block for remote end.
  - Replication is from replica to replica, "Replication pipeline"
    - Maximizes client's ability to stream