Cristina Nita-Rotaru

7680: Distributed Systems

GFS. HDFS

Required Reading

- Google File System. S, Ghemawat, H. Gobioff and S.-T. Leung. SOSP 2003.
- http://hadoop.apache.org
- A Novel Approach to Improving the Efficiency of Storing and Accessing Small Files on Hadoop: A Case Study by PowerPoint Files, Bo Dong; Jie Qiu; Qinghua Zheng; Xiao Zhong; Jingwei Li; Ying Li;, IEEE International Conference on Services Computing (SCC) 2010

1: GFS

Google File System

- File system for map/reduce
- Example of how to handle storage failures trading consistency for simplicity and performance
- What is GFS
  - Scalable, distributed file system for large distributed data-intensive applications
  - Based on a different set of assumptions leading to different design choices than conventional file systems
  - Implemented as a userspace library

Application Characteristics

- Hundreds of clients must perform concurrent (atomic) appends with minimal synchronization
- Sustained bandwidth more important than latency
- Response time for individual read/write not important
- Non-traditional access patterns
- Files are very big, several gigs

Design Choices

- Fault-tolerance: Constant monitoring, error detection, automatic recovery part of the design
- Designed for big files: I/O operations and block sizes have to be revisited
- Non-traditional access patterns:
  - Most files are modified by appending rather than overwriting
  - Large repositories that must be scanned (archival, streams, intermediate data)
  - Result: appending is the focus of optimization
- Co-design applications and file system: looser consistency

Interface Design

- Not standard API (such as POSIX)
- Supports file/directory hierarchy and the usual: {create, delete, open, close, read, write}
- Files identified by path names
- Additionally:
  - Snapshot: low cost file / directory tree copying
  - Record append: concurrent appends, no locks

Architecture Overview

- Base unit is chunk:
  - Chunk: fixed-part of a file, typically 64 MB
  - Global ID: 64 bit, unique "chunk handle", assigned by master server upon chunk creation
  - Read/Write: need chunk handle + byte range
  - Each chunk is replicated, minimum three copies
  - Stored as a plain Linux file on a chunkserver
- Servers:
  - Single master - metadata
  - Multiple backups (chunkservers)
- Multiple clients

GFS Architecture

Clients cache for a limited time the **chunk handle and locations** so they could interact directly with the chunckservers
Neither the client or chunservers cache file data.

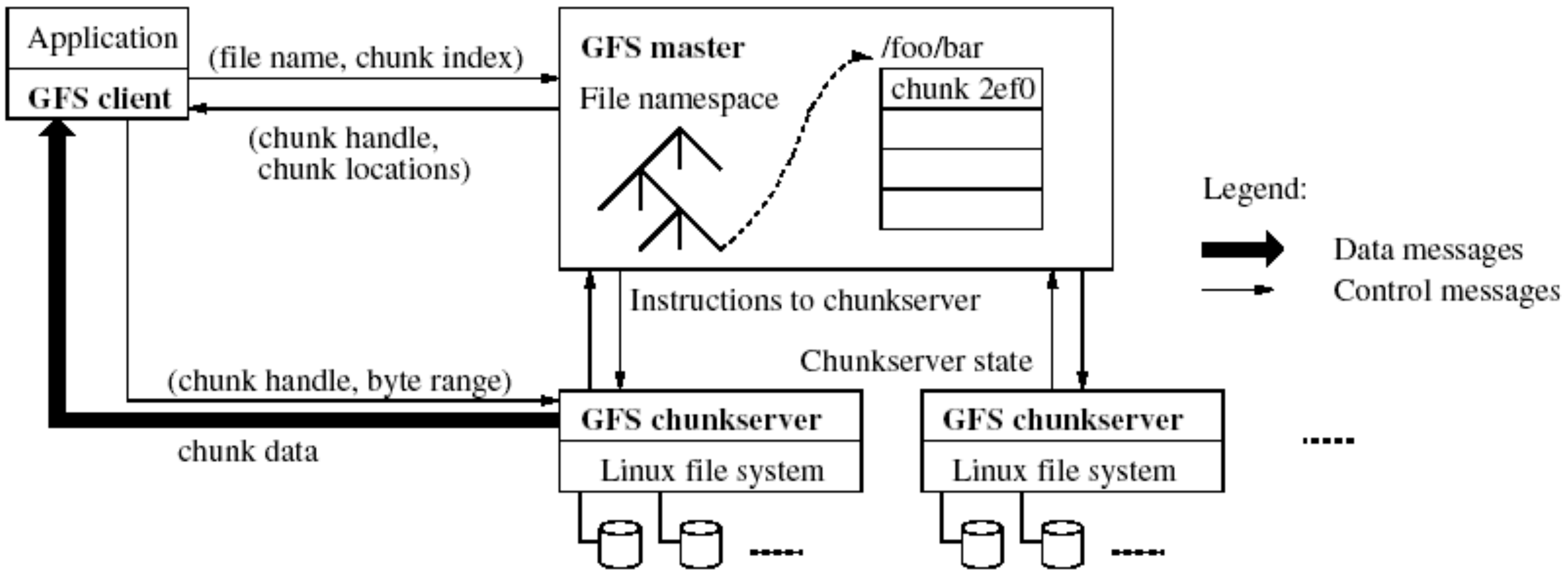


Figure 1: GFS Architecture

Metadata

- Master server is in charge of metadata
- File and chunk namespaces, logged to disk with chunk version
- Mapping from files to chunks, logged to disk with chunk version
- Location of chunk replicas, master obtains it at startup from each chunkserver, does not keep a persistent record of what chunkserver has what chunks
- Metadata maintained in memory to speed things up

Why Location is not Persistent

- If master server keeps location of chunks persistent then the master and chunkservers must be kept in sync as chunkservers join and leave
  - Difficult when there is to much churn
- Chunkserver ultimately knows what chunks it does or does not have on its own disks.

Operation Log

- Persistent log of historical changes of metadata
- Defines also the order of concurrent operations
- It is replicated on several remote machines and response goes back to the client only after the corresponding log record was flushed to disk locally and remotely
- Master recovers its state by replaying the log
- Log must be kept small for fast startup, period checkpoint

Design Overview: Consistency Model

- File namespace modifications: atomic & handled only by master server, done in memory and logged on disk
- Concurrent writes, atomic appends
- Chunk state can be:
  - **consistent** if all clients see same data, no matter which replica they ask
  - **defined** if it is consistent and known, i.e. some modification done w/o interruption
  - **undefined** if concurrent modifications are successful, it's consistent at all replicas but bits mixed from the different writes
  - **inconsistent** on any failed modification, inconsistent regions may be padded or contain duplicates

System-Wide Activities

- Master server also in charge of
  - chunk lease management
  - garbage collection of orphaned chunks
  - chunk migration between chunkservers

Leases

- Lease: Permissions for modifications, valid 60 seconds, granted by the master server to a chunkserver to modify the chunk.
- Primary chunkserver:
  - Primary chooses the serial number for all changes on a chunk
  - It propagates the changes to the chunkservers with the backup copies. The changes are not saved until all chunkservers acknowledge.
- Clients access the chunks by first querying the master; if the chunk is not being operated on (e.g. no outstanding leases exist), the master replies with the location, and the client contacts the chunkserver directly
- Master can revoke a lease before it expires (for example master wants to rename a file)

Read

- Client translates the file name and offset into a chunk index within a file
- Sends master request containing filename and chunk index
- Master replies with chunk handle and locations of replica (this info is cached by client, further reads do not require interaction with the master)
- Client sends a replica chunkserver the chunk handle and a byte range within the chunk

Write

- Client asks master what chunkserver has the lease for the chunk it wants to write to, if no lease exists, the master will give one to a chunkserver it chooses
- Master replies with identity of primary and the other replicas for that chunkserver
- Client pushes data to all replicas
- Once all replicas ack, client sends the update to the primary
- Primary assigns the order and sends the update to the replicas.
- Replicas apply change in the same order as the primary and send ack back.
- Primary replies to client.

Write: what can go wrong

- If the client gets the reply back it means write can succeed at the primary and at some replicas, any error is reported to the client
- Could the errors have happened at the primary? – Inconsistent state
- If the data is larger than a chunk, multiple write operations. In this case operations may overlap with operations from other clients, different clients may overwrite each other's operations, order is the same by different fragments from different clients – undefined state
- How would you make writes serializable?

Record Append

- Instead of writing new chunk, append to existing one
  - Restricted to chunk-size / 4 to minimize chunk-boundary overflowage
  - if (boundary overflowage) new chunk must be created
  - else write into chunk @some_offset, and tell secondary replicas some_offset
- If append fails, client retries the operation, no guarantee replicas are byte-wise identical!
  - Guarantee: record was written at least once atomically (there can be duplicates)
  - Failures at subset of chunkservers… inconsistent regions
  - If a secondary server was offline and missed some updates… may have holes after record append

Snapshot

- Makes a copy of file or directory tree
- Used to quickly create branch copies of huge data sets, or to checkpoint current state before experimenting to allow rollback
- Uses copy-on-write (like AFS)
  - Master revokes chunk leases, then snapshots
  - Any time a lease is requested, chunk is copied first on same chunkserver
  - After lease expires, commit to log + copy metadata

Snapshot Details

- Master receives snapshot request
- Master revokes existing leases, any other modification on those file have to go to the master first
- Master logs operation on disk
- Master makes change in metadata in memory, newly snapshot files point to same chunks as original file
- Client wants to write chunk C after snapshot, master sees count for C > 1 picks new handle id C’ ask chunkserver that has C to create a copy of C, with the new id C’.

Namespace Management & Locking

- GFS: not your typical file system
  - Lacks typical per-directory data structure to list each file in the directory
  - Does not support aliases (i.e. hard or sym links)
  - Namespace: lookup table that maps full pathnames to metadata
  - Lookup table fits in memory (prefix compression)
- Master ops acquire set of locks before running
  - Ex: read locks: /d1, /d1/d2, /d1/d2/…/dn, /d1/d2/../dn/leaf to lock "leaf"
  - File creation does not lock parent dir: no directory or inode-esque structure to modify
  - Allows concurrent modifications in same directory
  - Locks always acquired in same order, prevents deadlock

Replica Replacement

- Hundreds of chunkservers across many hardware racks, each accessed by hundreds of clients on same or different racks
- Want to maximize reliability, availability but also network utilization
- Must spread replicas across machines and racks
  - Replicas survive switch failure or power outage of entire rack, etc
  - Read traffic for a chunk can exploit bandwidth of multiple racks
  - Write traffic must flow through multiple rack

cReation, Re-replication, Rebalancing

- Creation:
  - Place new replicas on servers with <avg disk use
    - Equalizes disk utilization across chunkservers
  - Limit number of recently created chunks per server
    - Creation implies chunk will be heavily written soon
  - Place new replicas on different racks
    - Requires IP addressing scheme to identify physical location of chunkservers

Garbage Collection

- Files that are "deleted"… aren' t
  - "Deleting" a file renames it… now it' s "hidden"
  - Hidden files >= 3 days old are removed
    - 3 day limit is configurable
    - Prevents accidental-and-irreversible data loss
  - Undelete by renaming hidden file
  - Lazy garbage collection, i.e. during idle time
    - In-memory metadata erased
    - Orphaned chunks gradually erased by chunkservers
    - Also removes stale replicas (not up to current chunk version #)
  - 3-day delay caused some trouble for some users
    - Temp files stick around, wasting space
    - Replication level can be user specified per part of file namespace

Fault Tolerance and Diagnosis: Availability

- Fast recovery
  - Master, chunk servers restore state quickly
  - No distinction between normal/abnormal termination
- Chunk replication
- Master replication
  - State of master server is replicated
  - External watchdog can change DNS over to replica if master fails
  - Additional “shadow” masters provide RO access during outage
    - Shadows may lag the primary master by fractions of 1s
    - Only thing that could lag is metadata
    - Depends on primary master for replica location updates

Fault Tolerance and Diagnosis: Integrity

- Chunkservers checksum to detect corruption
  - Corruption caused by disk failures, interruptions in r/w paths
  - Each server must checksum because chunks not byte-wise equal
  - Chunks are broken into 64 KB blocks
  - Each block has a 32 bit checksum
  - Checksums kept in memory and logged with metadata
  - Can overlap with IO since checksums all in memory
- Client code attempts to align reads to checksum block boundaries
- During idle periods, chunkservers can checksum inactive chunks to detect corrupted chunks that are rarely read

2: HDFS

HDFS Basics

- An open-source implementation of Google File System, created in 2005
- Assume that node failure rate is high
- Large files, some several GB large
- Write-once-ready-many pattern
- Reads are performed in a large streaming fashion
- Large throughput instead of low latency
- Moving computation is easier than moving data

HDFS Files

- User data divided into 64MB blocks and replicated across local disks of cluster node to address:
  - Cluster network bottleneck
  - Cluster node crashes
- Master/Slave Architecture
  - Master (Namenode) maintains a name space and metadata
  - Slaves (Datanodes): maintain three copies of each data block

Namenode

- Arbitrator and repository for all HDFS metadata
- Data never flows through Namenode
- Executes file system namespace operations
  - open, close, rename files and directories
- Determines mapping of blocks to Datanodes

EditLog & FsImage

- Managed by Namenode
  - Stored in files on the local OS file system
- EditLog
  - Transaction log
  - Records all changes to file system metadata
- FsImage
  - Image of entire file system namespace
  - Mappings of blocks to files
  - File system properties
  - Stored in a file on the local OS file system

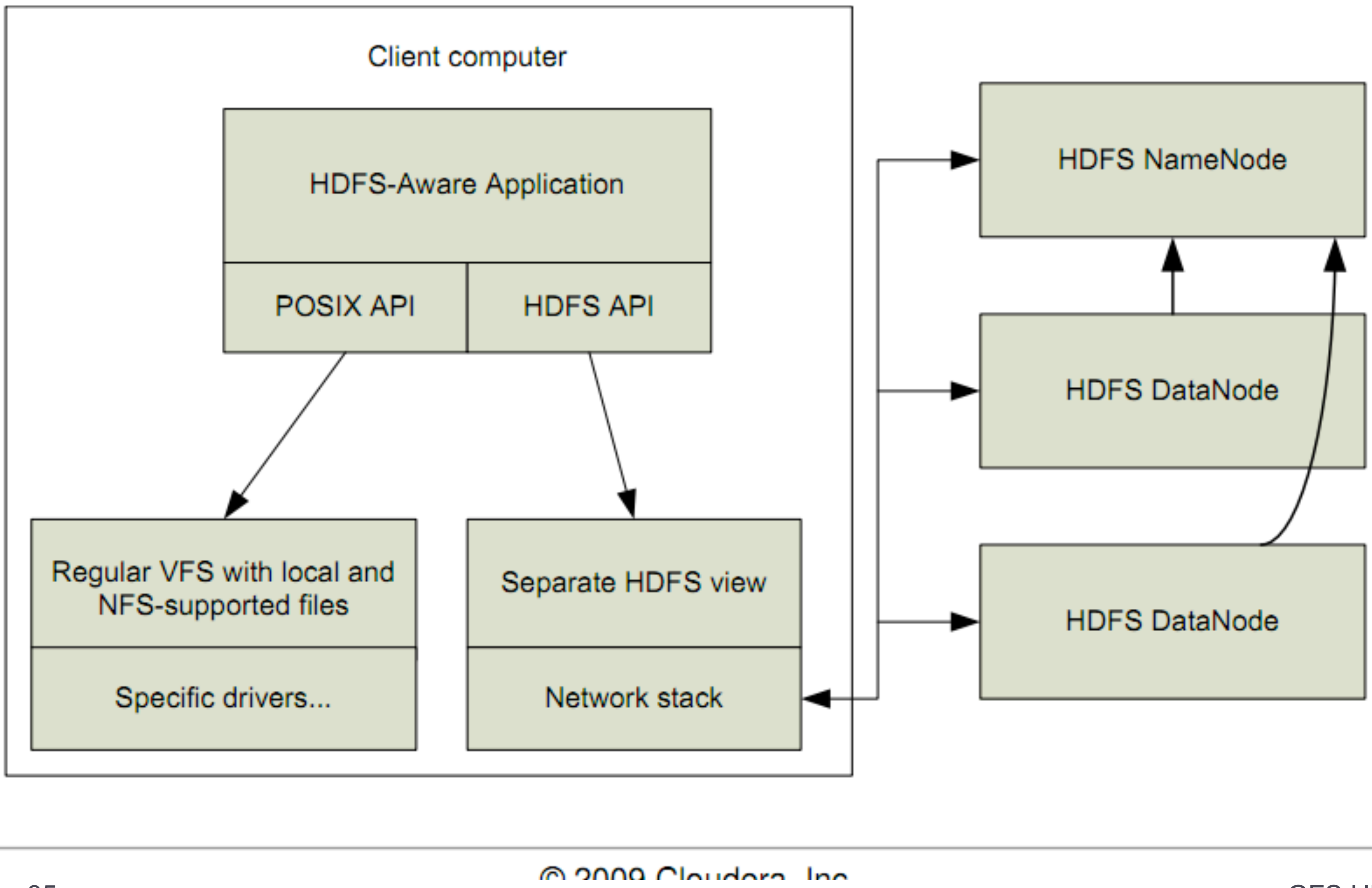
Datanodes

- Serve read / write requests from client
- Block creation, deletion and replication upon instruction from Namenode
- No knowledge of HDFS files
- Stores HDFS data in files on local file system
  - Determines optimal file count per directory
  - Creates subdirectories automatically

Communication Protocols

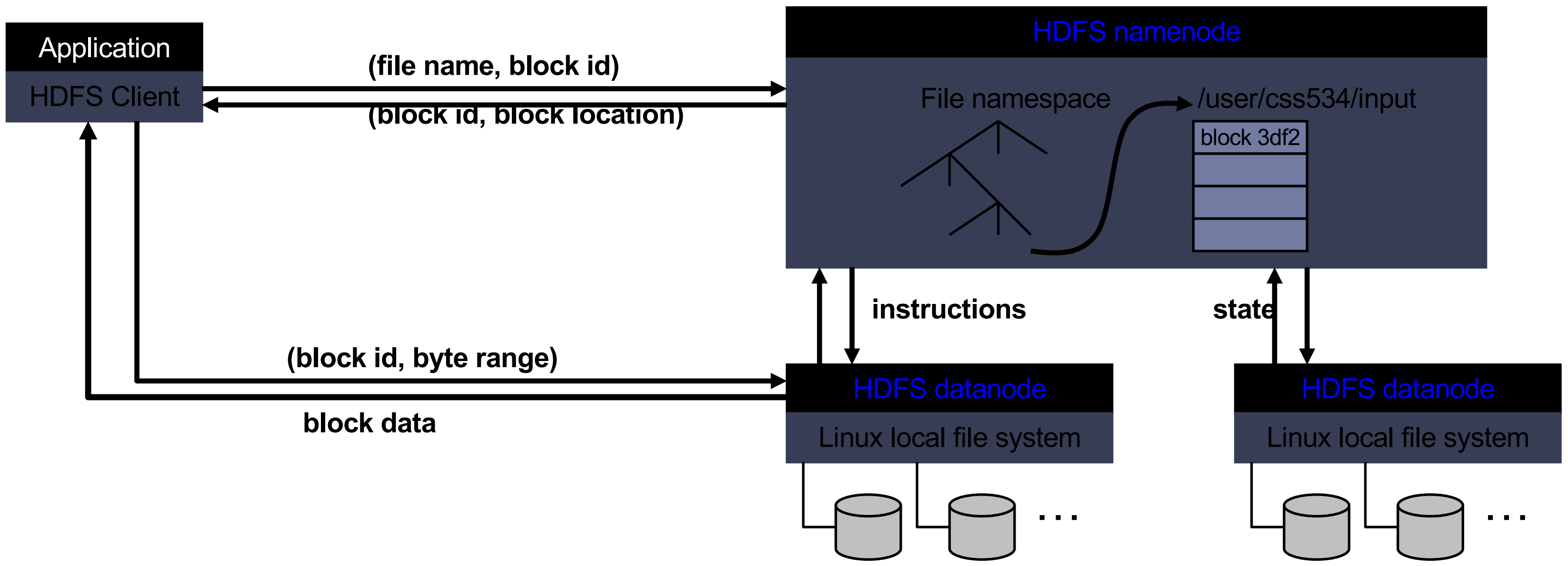
- Layered on top of TCP/IP
- RPC abstraction wraps protocols
- ClientProtocol
  - Client talks to Namenode
- Datanode Protocol
  - Datanodes talk to Namenode
- Namenode never initiates any RPCs
  - It only responds to RPC requests

HDFS Client Block Diagram



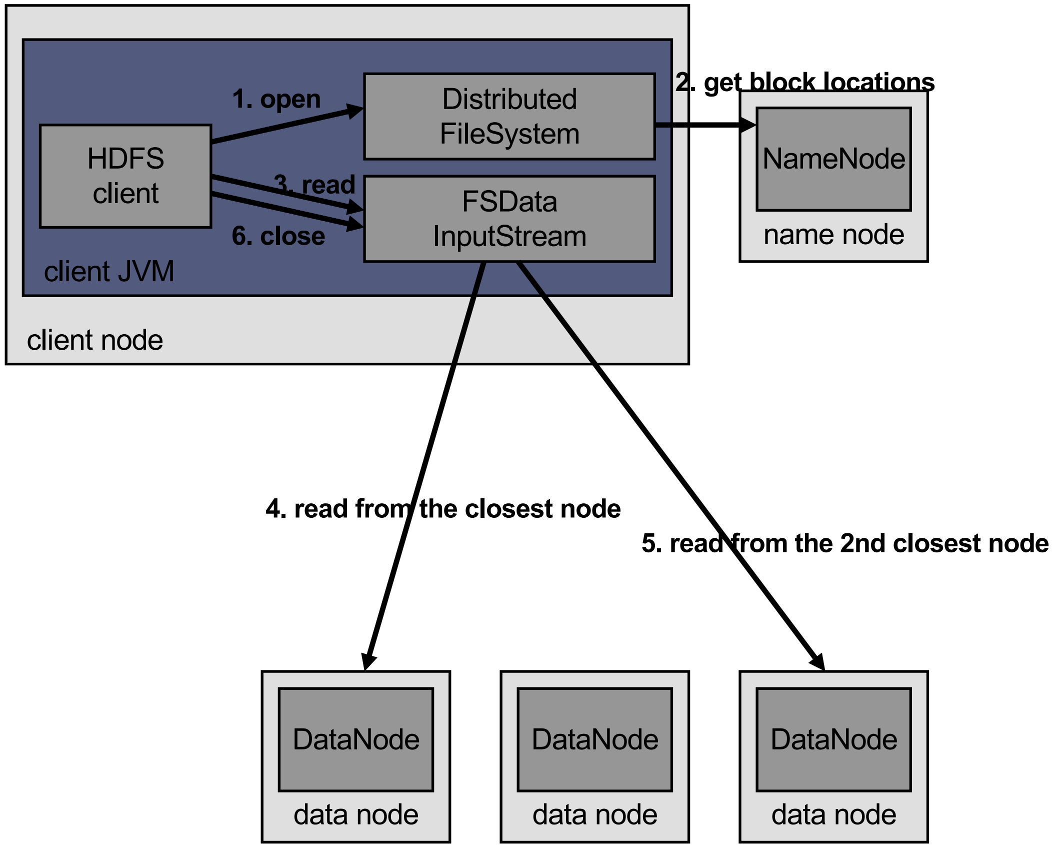
HDFS: Hadoop Distributed File Systems

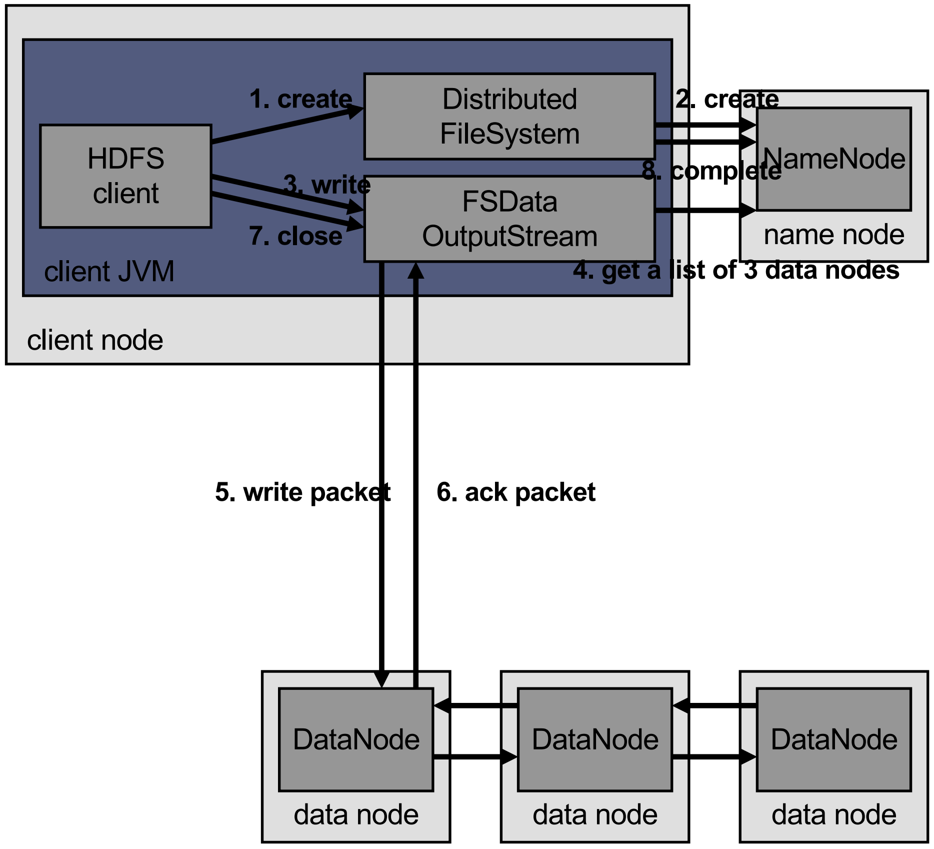
- Client requests meta data about a file from namenode
- Data is served directly from datanode



File Read/Write in HDFS

- File Read
- File Write





If a data node crashed, the crashed node is removed, current block receives a newer id so as to delete the partial data from the crashed node later, and Namenode allocates an another node.

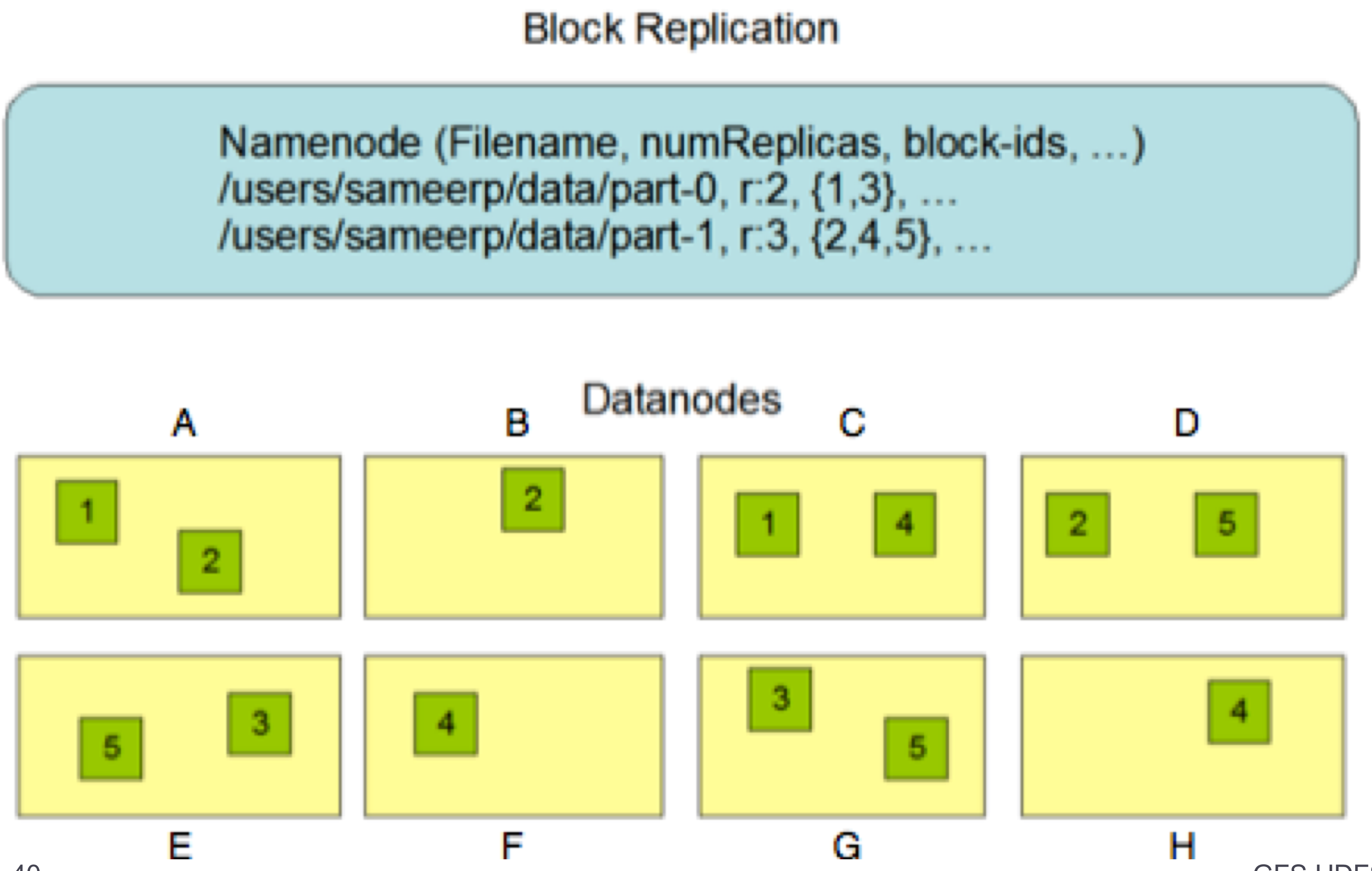
Replica Placement

- Distinguishes HDFS from most other DFS
- When replication factor == 3
  - Put one replica on local rack
  - Put one replica on different node on local rack
  - Put one replica on different node on different rack
- Replicas do not evenly distribute across racks

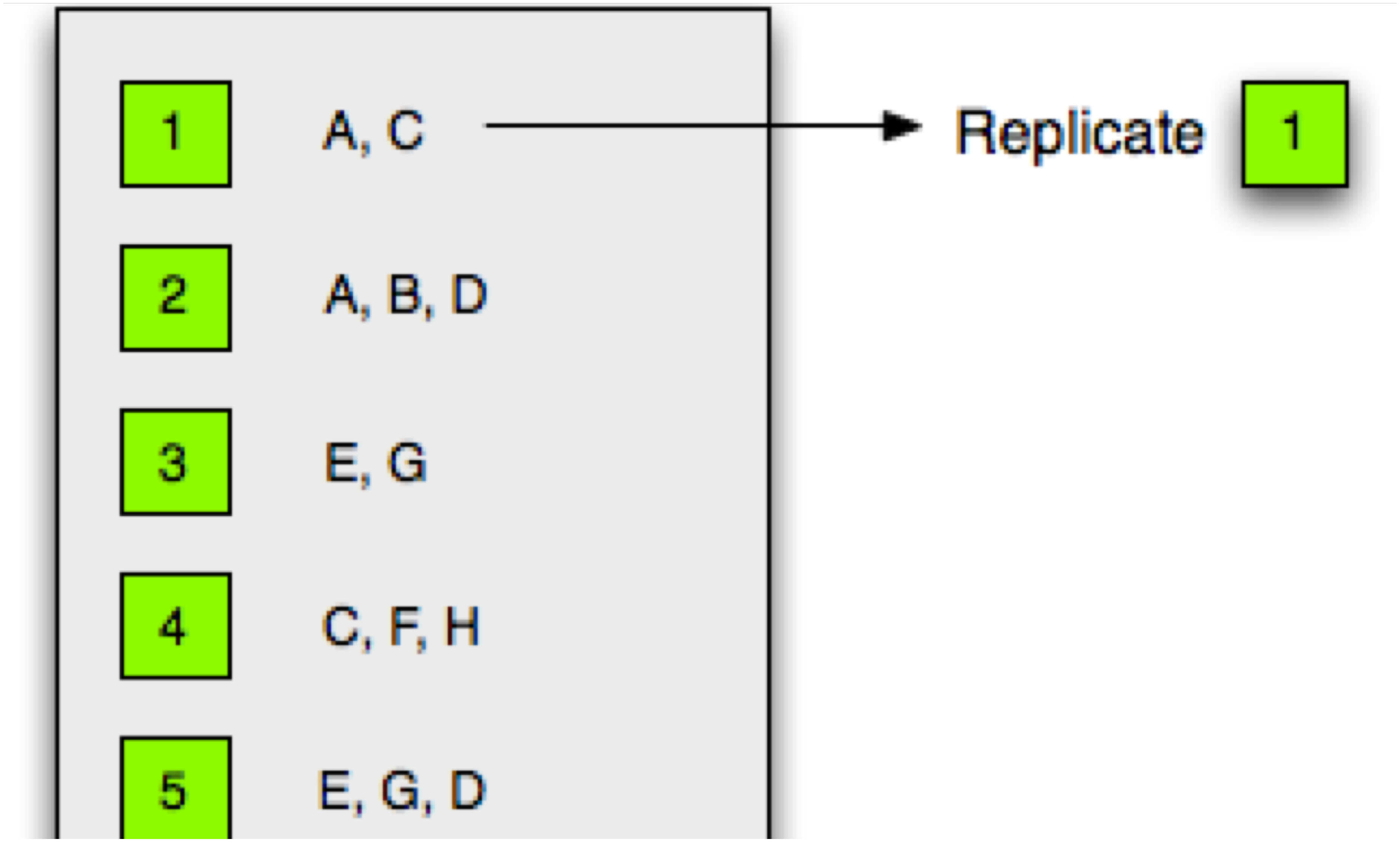
Start-Up Process

- Namenode enters Safemode
  - Replication does not occur in Safemode
- Each Datanode sends Heartbeat
- Each Datanode sends Blockreport
  - Lists all HDFS data blocks
- Namenode creates Blockmap from Blockreports
- Namenode exits Safemode
- Replicate any under-replicated blocks

Datanode Blockreports


Block Replication
Namenode (Filename, numReplicas, block-ids, ...)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...
Datanodes
A
1
2
B
2
C
1
4
D
2
5
E
5
3
F
4
G
3
5
H
4

BlockMap and Replication

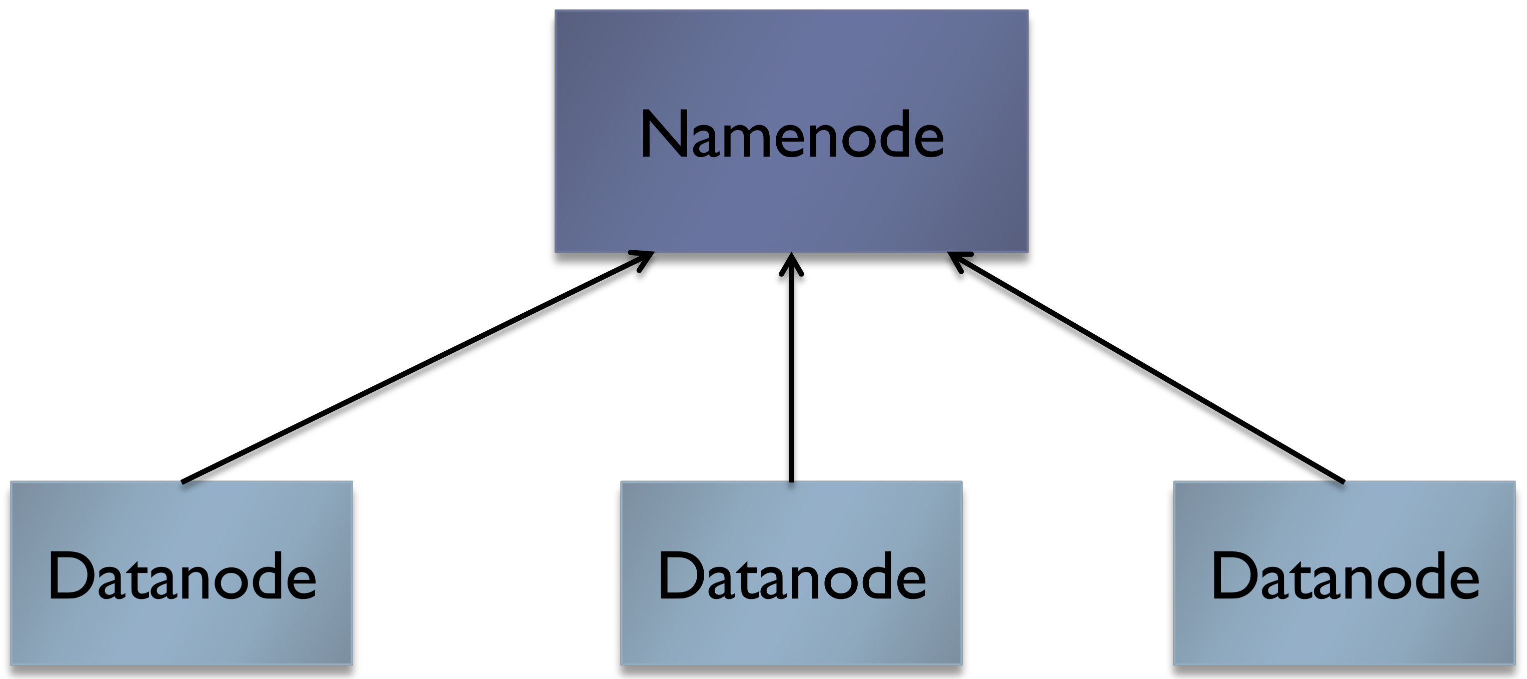


Checkpoint Process

- Performed by Namenode
- Two versions of FsImage
  - One stored on disk
  - One in memory
- Applies all transactions in EditLog to in-memory FsImage
- Flushes FsImage to disk
- Truncates EditLog
- Currently only occurs on start-up

Datanode Failure

- Datanode sends periodic Heartbeats



Datanode Failure

- Namenode marks Datanodes without recent heartbeat as dead
- Does not forward any new I/O requests
- Constantly tracks which blocks must be replicated with BlockMap
- Initiates replication when necessary

Namenode Failure

- Single Point of Failure for HDFS cluster
- FsImage and EditLog are central data structures for HDFS
  - Corruption / loss of these files causes HDFS to become non-functional
  - Manual intervention is necessary
- Automatic restart and failover of Namenode not yet supported (but planned)

Limitations

- Write-once model
  - Plan to support appending-writes
- A namespace with an extremely large number of files exceeds Namenode’s capacity to maintain
- Cannot be mounted by exisiting OS
  - Getting data in and out is tedious
  - Virtual File System can solve problem
- Java API
  - Thrift API is available to use other languages

Limitations

- HDFS does not implement / support
  - User quotas
  - Access permissions
  - Hard or soft links
  - Data balancing schemes
- No periodic checkpoints
- Namenode is single point of failure
  - Automatic restart and failover to another machine not yet supported

Recent Additions

- Allow for a fail-over backup for the namenode
- Allow multiple namenodes (serving multiple namespaces) organized in federations, no synchronization/consistency across them