LINKÖPING UNIVERSITY

Introduction to Parallel I/O by Distributed File Systems

Christoph Kessler

IDA, Linköping University

2003

Cluster: How to speed up the slow I/O?

The default file system in a compute cluster
is an ordinary (**sequential**) **shared file system**

- **Remote shared file server** (containing, e.g., users' home directories, bin)
  - Connected to cluster nodes by the interconnection network
  - File system "mounted" for remote access from each cluster node
  - Files exist (in principle) only in one place – easy to find, consistent
  - Cluster nodes access the file server concurrently with remote file-read / file-write commands over the interconnection network

  ☹ Performance (throughput) bottleneck for I/O intensive programs!

  ☹ Performance (latency) bottleneck for accessing LARGE files!

  ☹ Single point of failure

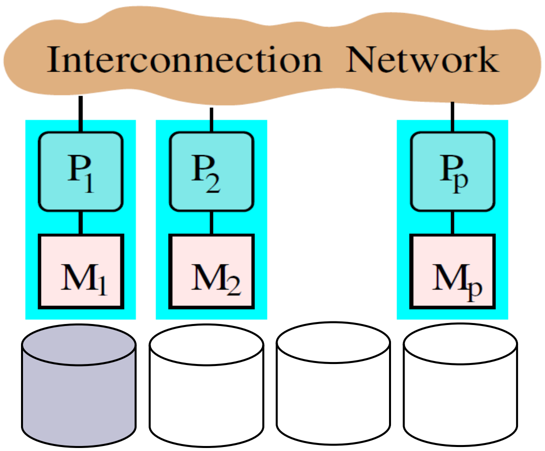
- Each cluster node also has its own **node-local secondary storage**
  - usually a node-local hard disk with local file system controlled by the node's operating system

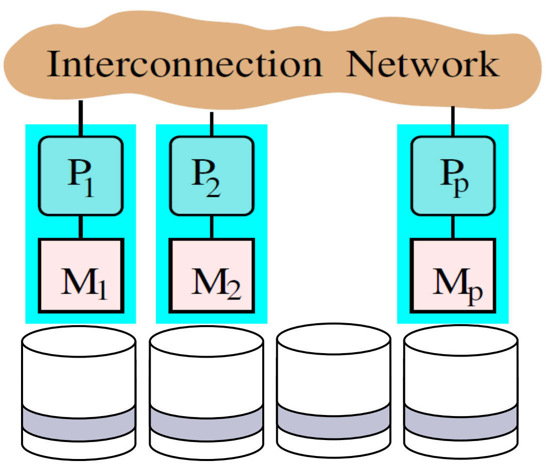
  ☹ Directly accessible to that node only

Towards Parallel I/O Processing of Big-Data

Big Data …

- too large to be read+processed in reasonable time by 1 server only
- too large to fit in main memory at a time
  - Usually residing on **secondary storage** (local or remote)
- Storage on a single hard disk (sequential access) would prevent parallel I/O processing
- Solution: **Partition and distribute** the contents of the file system across nodes to allow for parallel access





→ For parallelizing the I/O work, need to use a **distributed file system**

More Cluster Issues: The Need for Fault Tolerance

Typical disk-based server failure rate *

- Assume 4 SATA disks per server (node)
- Assume 5% disk failure rate / year

→ 20% of servers fail from disk every year!

- Assume 5% server failures from other reasons (power supplies etc.)

→ 25% of the servers fail every year

→ 1 in 1360 servers fails each day

→ In a datacenter with 50,000 servers, 37 servers fail each day!

Solution:

- Redundancy by replication of data at file system level prevents data loss
- Combine with the distributed file system approach

* Estimation based on: P. Helland *et al.*: Too big NOT to fail. *Communications of the ACM* 60(6):46-50, June 2017.

Distributed File System

- Large files are **distributed** (”***sharded***”)

  = split into blocks of e.g. 64MB (**shards**)
  and spread out (e.g., hashed) across the cluster nodes

  - Each shard may be stored as an ordinary file on a node-local hard disk
  - Each node owns only some shards = a fraction of a distributed file
  - Need a **directory** to look up where (on which node) to find which shard
    - **Directory / name server** to look up distributed files’ metadata and shard locations
  - **Parallel access** to distributed files is possible
    - Can access multiple shards of the same file in parallel
    - ☺ Higher bandwidth, lower latency
- Also, **replicas** for fault tolerance
  - E.g. 3 copies of each shard on *different* servers
  - ☺ Can **read** from the *closest* copy
  - ☹ Need to keep copies *consistent* – **write**s (all copies) are expensive
- Examples of distributed file systems: Google GFS, Hadoop HDFS
- If starting from input data in an ordinary file, need to first copy the data from ordinary (host) file system to distributed FS

Example: Hadoop Distributed File System (HDFS)

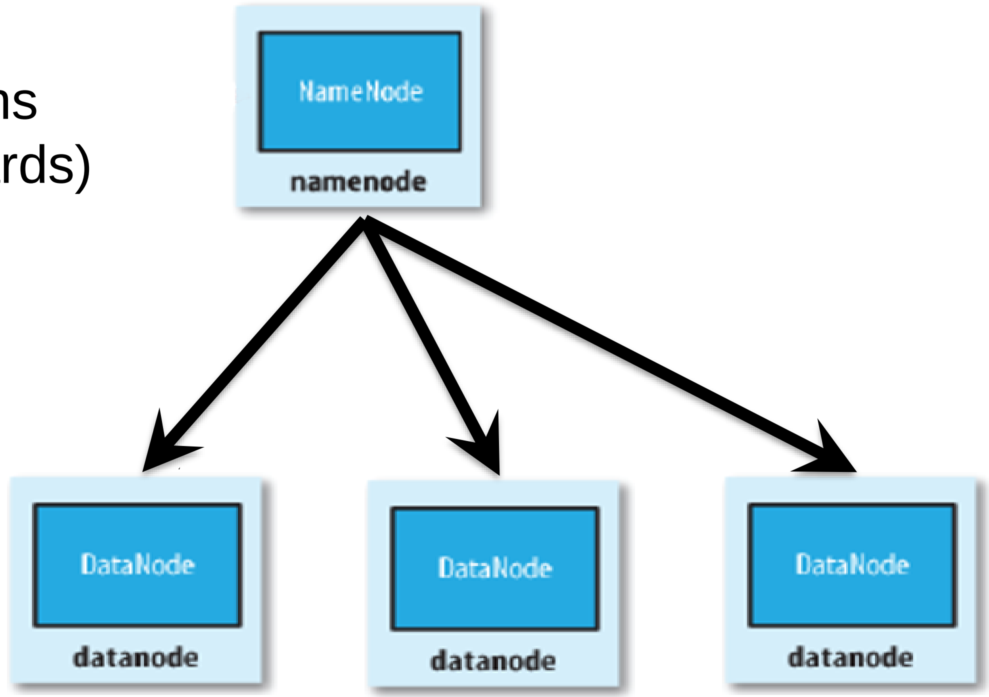
Christoph Kessler, IDA,
Linköpings universitet.

Example: HDFS

- **Hadoop Distributed File System**
  - For very large files on clusters
- Runs on top of the native file systems
  - Files divided into 64MB or 128 MB blocks (shards)
    - Block size is a configuration parameter
  - Usually, 3 copies per block for fault tolerance
    - Stored on different nodes, preferably one on a different rack
- HDFS file: Write once, read multiple times
  - Caching blocks is possible
  - Exposes the locations of file blocks via API
- Handles failures – disk/node/rack failures

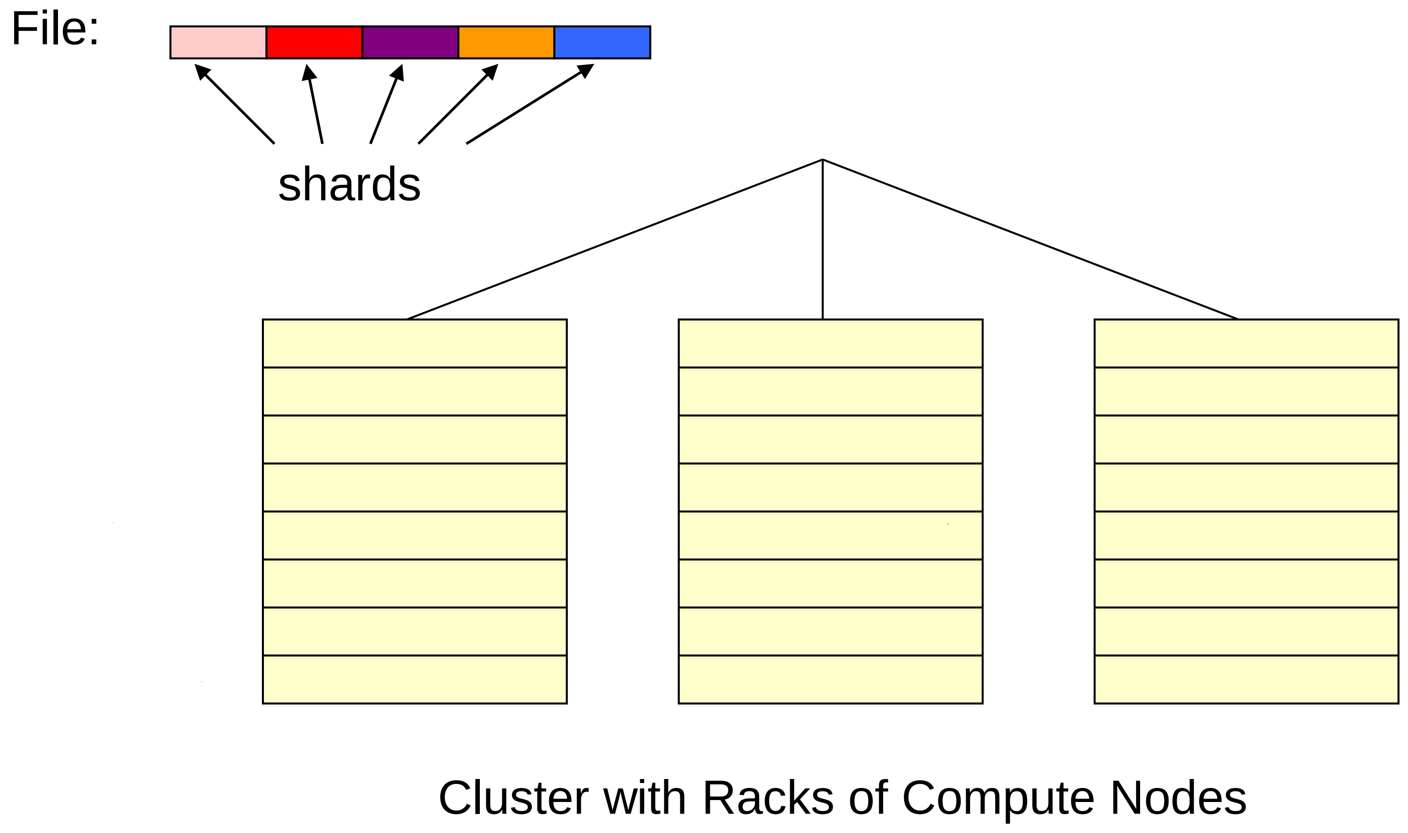
HDFS Organization

- **Name-node** (master)
  - Process that manages the file system namespace and **metadata**
  - Stores in memory the locations of all copies of all blocks (shards) for each HDFS file
  - Lookup of block locations
- **Data-nodes** (workers)
  - Process, one on each node
  - Performs writing and reading of blocks
  - Send heartbeat to the name-node for failure detection



HDFS Example

- How to distribute the blocks (shards) with replicas?

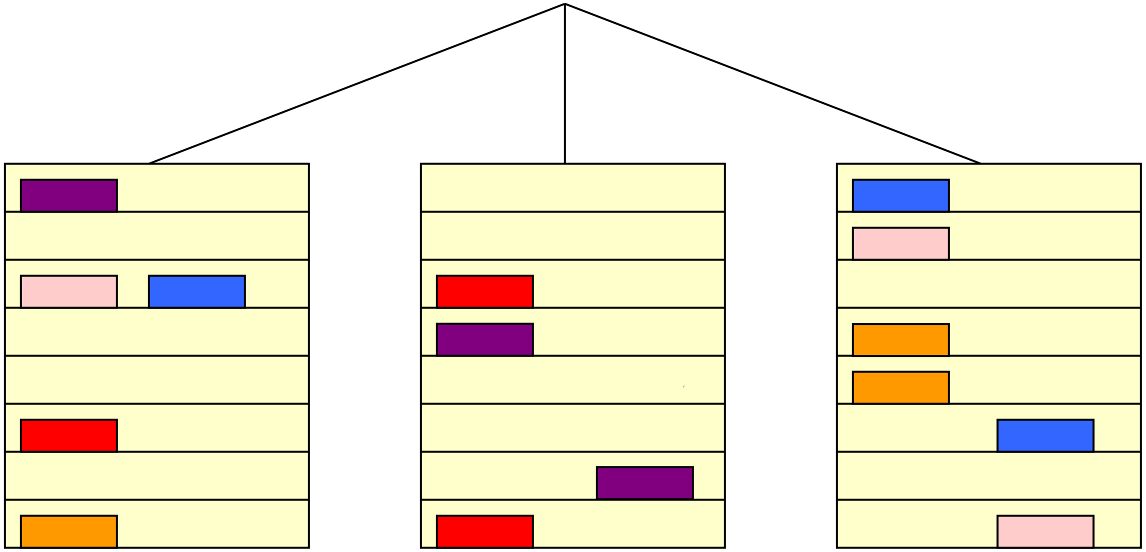


Source: J. D. Ullman invited talk EDBT 2011

HDFS Block Placement and Replication

- Aim: improve data reliability, availability, and network bandwidth utilization
- Default replica placement policy
  - No data-node contains more than one replica
  - No rack contains more than two replicas of the same block
- Name-node ensures that the number of replicas is reached
- Balancer tool – balances the disk space usage
- Block scanner – periodically verifies checksums

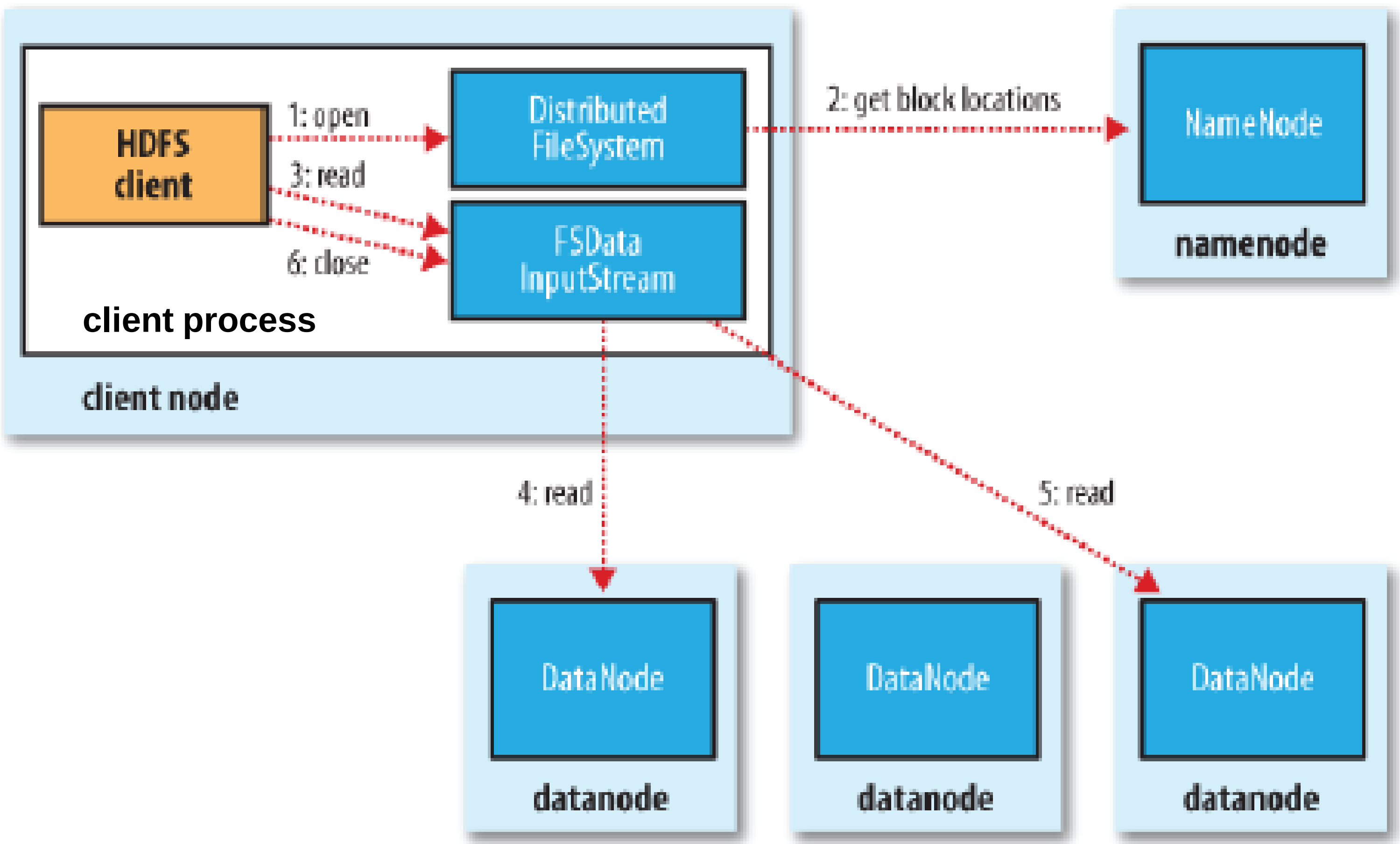
Default HDFS block placement policy



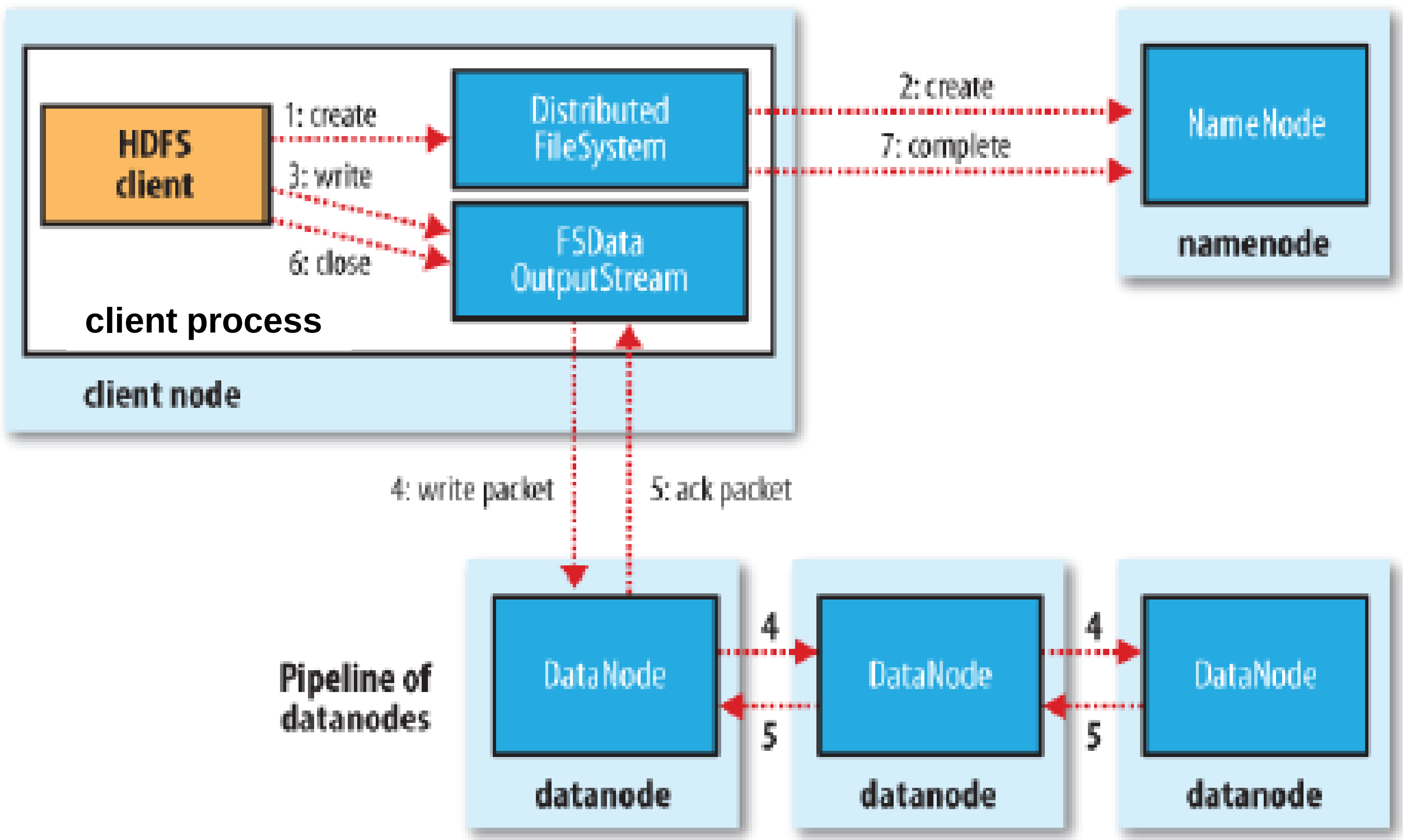
Write:

- 1st replica located on the writer node
- 2nd and 3rd replicas on two different nodes in a different rack
- Any other replicas (if any) are located on random nodes

HDFS – File Reads



HDFS – File Writes



HDFS is Good for …

- Storing very large files – GBs and TBs
- High-throughput parallel I/O
  - Time to read the entire dataset is more important than the latency in reading the first record.
- Commodity hardware
  - Clusters are built from commonly available hardware
  - Designed to continue working without a noticeable interruption in case of failure

HDFS is currently Not Good for …

- Low-latency data access
  - HDFS is optimized for delivering high throughput of data
- Lots of small files
- Re-writing the same HDFS file, and arbitrary file modifications
  - HDFS files are append-only
    – write is only allowed at the end of the file