Hadoop Distributed File System

Thomas Schwarz, SJ

HDFS Design

- HDFS is designed for:
  - Very large files
  - Streaming data access:
    - write-once-read-often
    - Latency is not important, but time to read the whole dataset is
  - Commodity hardware

HDFS Design

- HDFS is not good for:
    - Lots of small files
        - Uses a namenode, which has limited memory
            - ~150B per entry: 1GB gives 6.7 Million entries
    - Low-latency data access
        - Use HBase instead
    - Multiple writers with consistency
        - Writes are made only in append mode by a ***single*** writer

HDFS Design

- Blocks
  - Not storage blocks (~4KB), but much larger: > 128MB
  - Breaking files into blocks means HDFS is not limited by storage device sizes

HDFS Design

- Namenode and datanodes:
  - Namenode manages the file system:
    - Stored persistently are:
      - namespace image
      - edit log
  - Datanodes store and retrieve blocks
    - Periodically report to the namenode the blocks that they are storing

HDFS Design

- Client interface:
    - Portable Operating System Interface (POSIX)

HDFS Design

- Failure resilience:
  - Backup namenode image and entry log
  - Second namenode periodically merges the namenode image with the edit log
    - In case of primary namenode failure, some data is likely lost because of lag between primary and secondary namenode
  - Run a “hot standby namenode”

HDFS Design

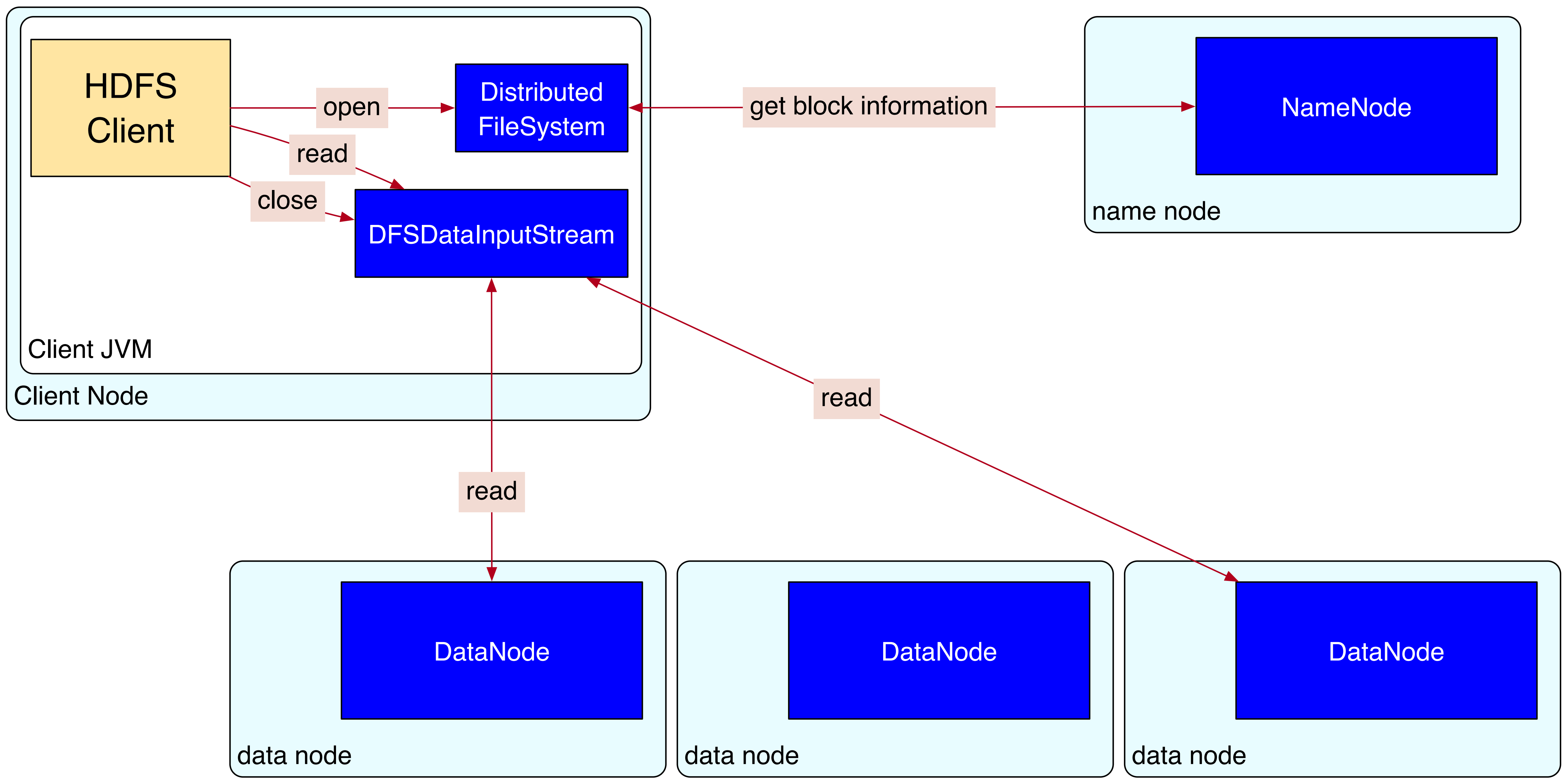
- HDFS Federation:
    - A collection of namespaces (**namenode volume)** with their own namenode

HDFS Interface

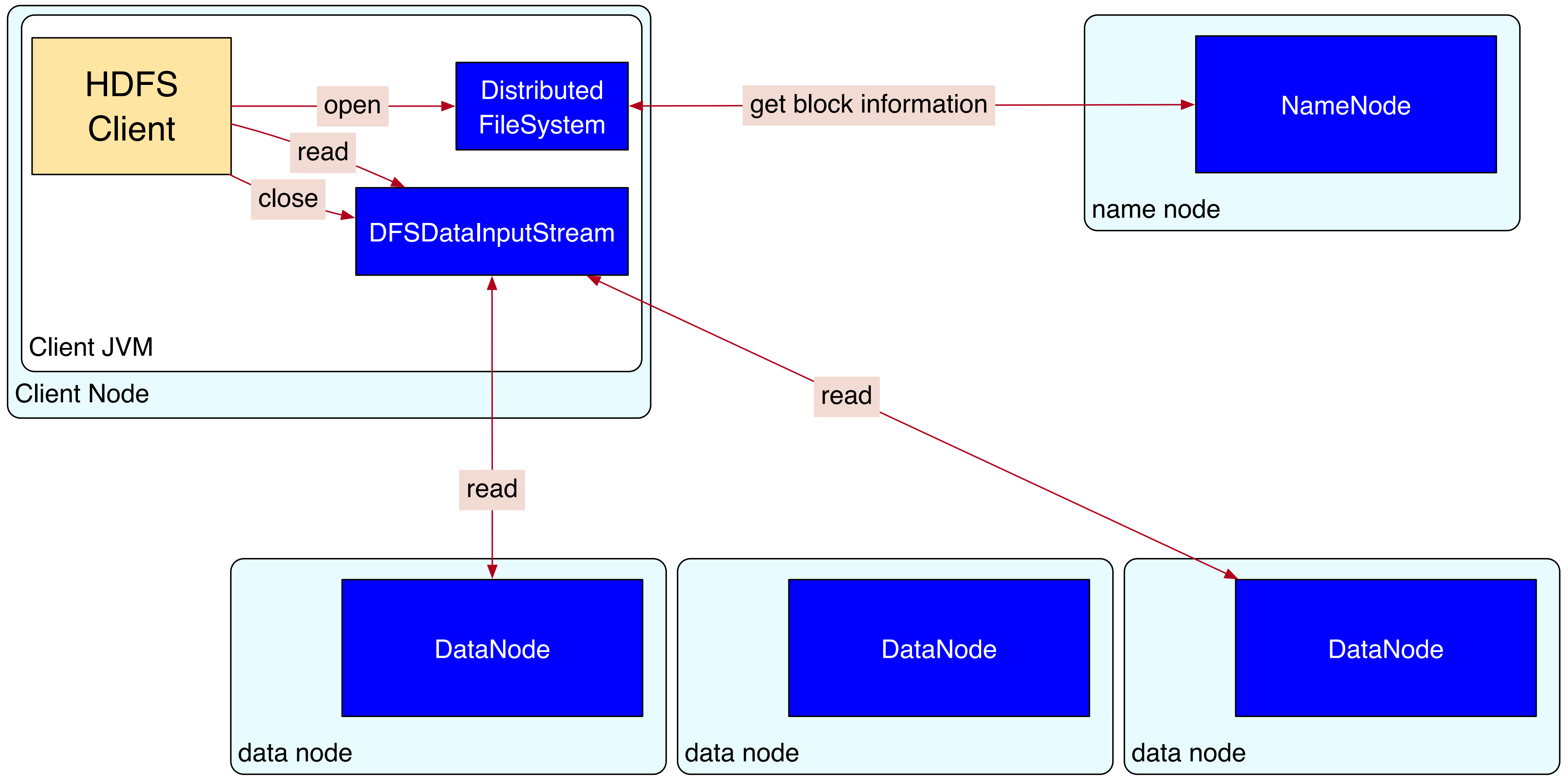
- HDFS implements an abstract file system defined by
  - `org.apache.hadoop.fs.FileSystem`
- Hadoop accesses HDFS mostly through a Java API
- Alternative: HTTP REST API

HDFS Interface

Reads

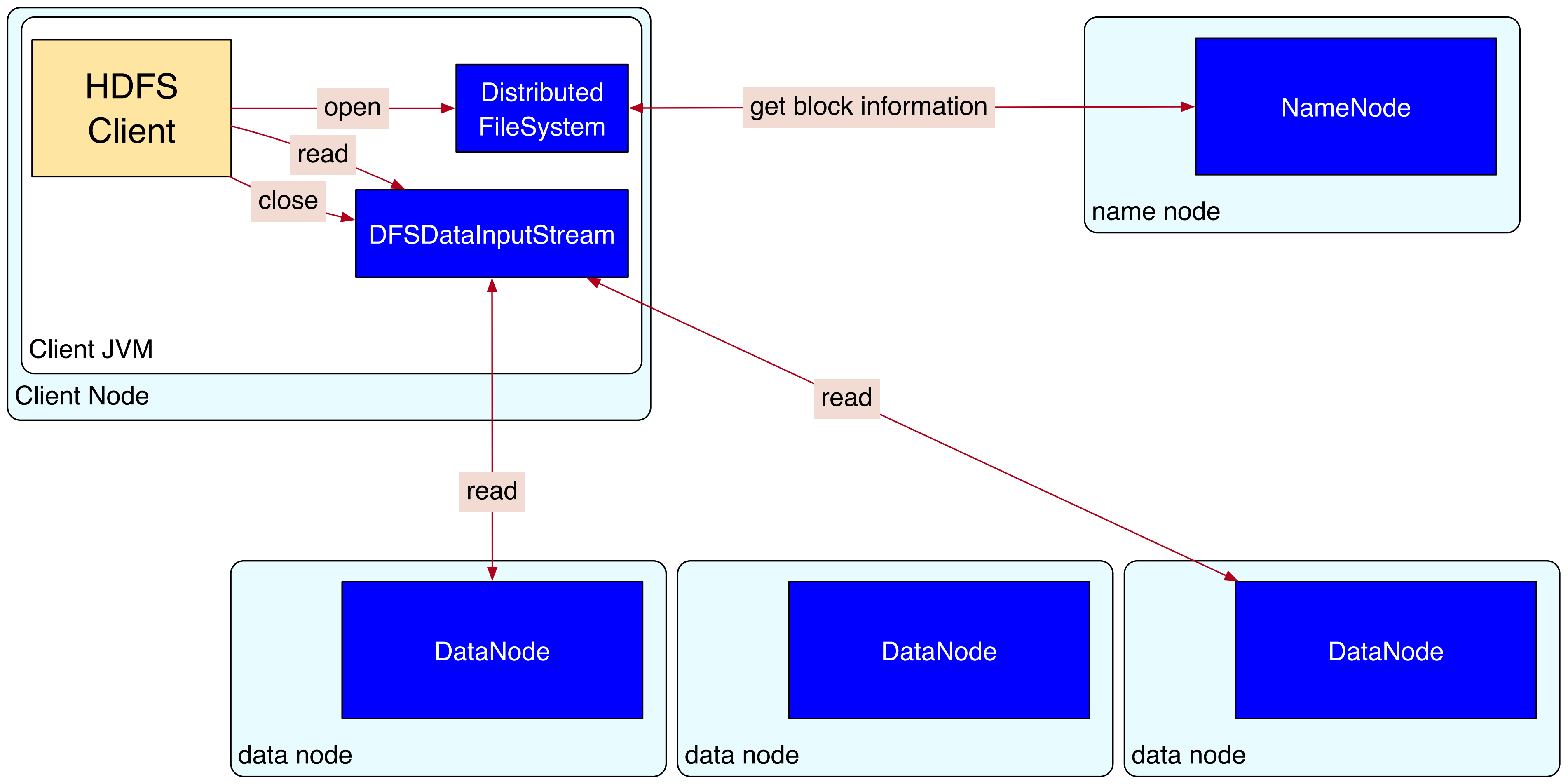


HDFS Interface



1. HDFS client makes an open call to HDFS, an instance of DistributedFileSystem.

HDFS Interface

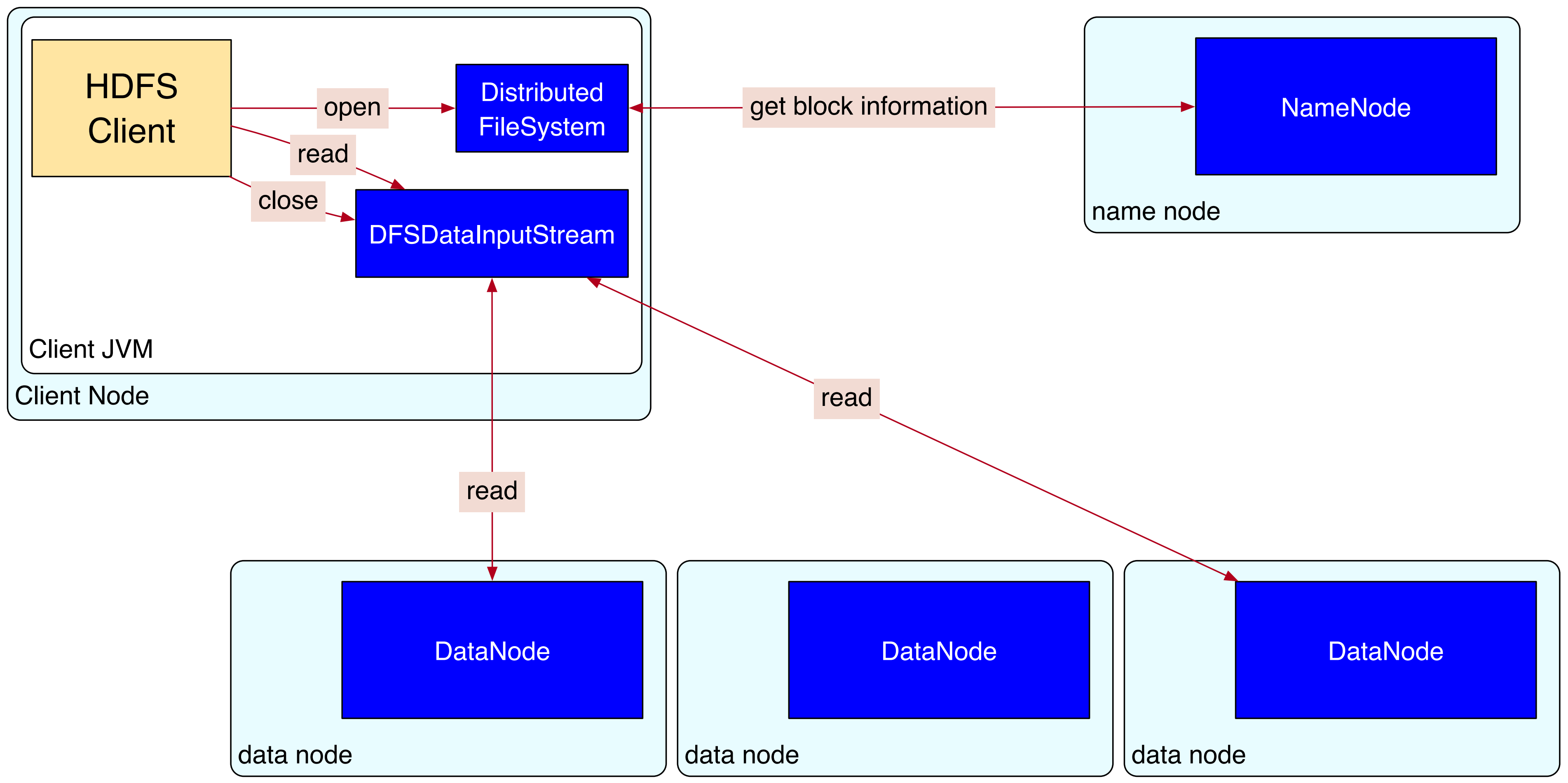


2. API translates into RPC calls to the NameNode DistributedFileSystem.

Determines the addresses of the first blocks.

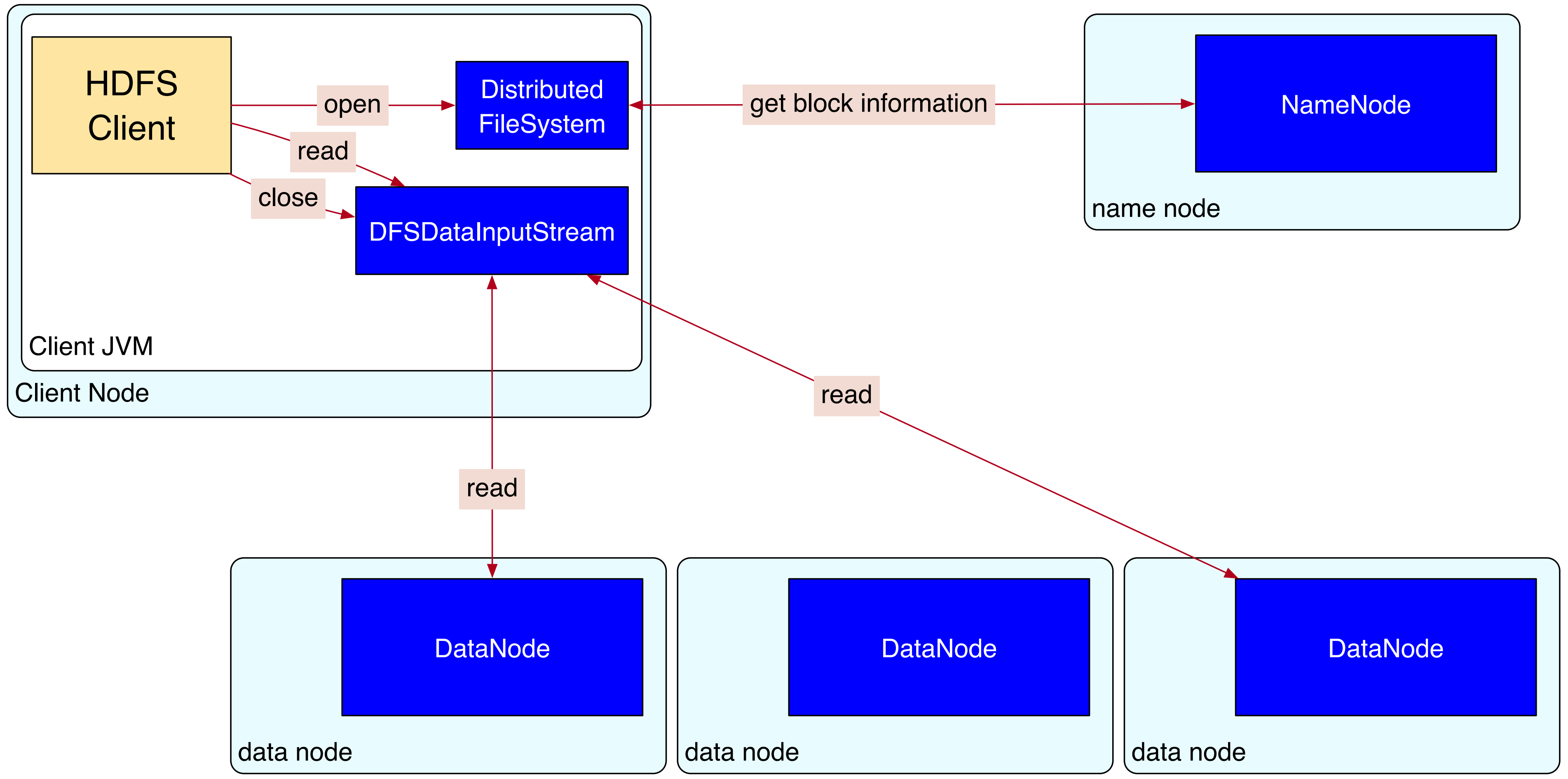
Prefers access to locally cached blocks

HDFS Interface



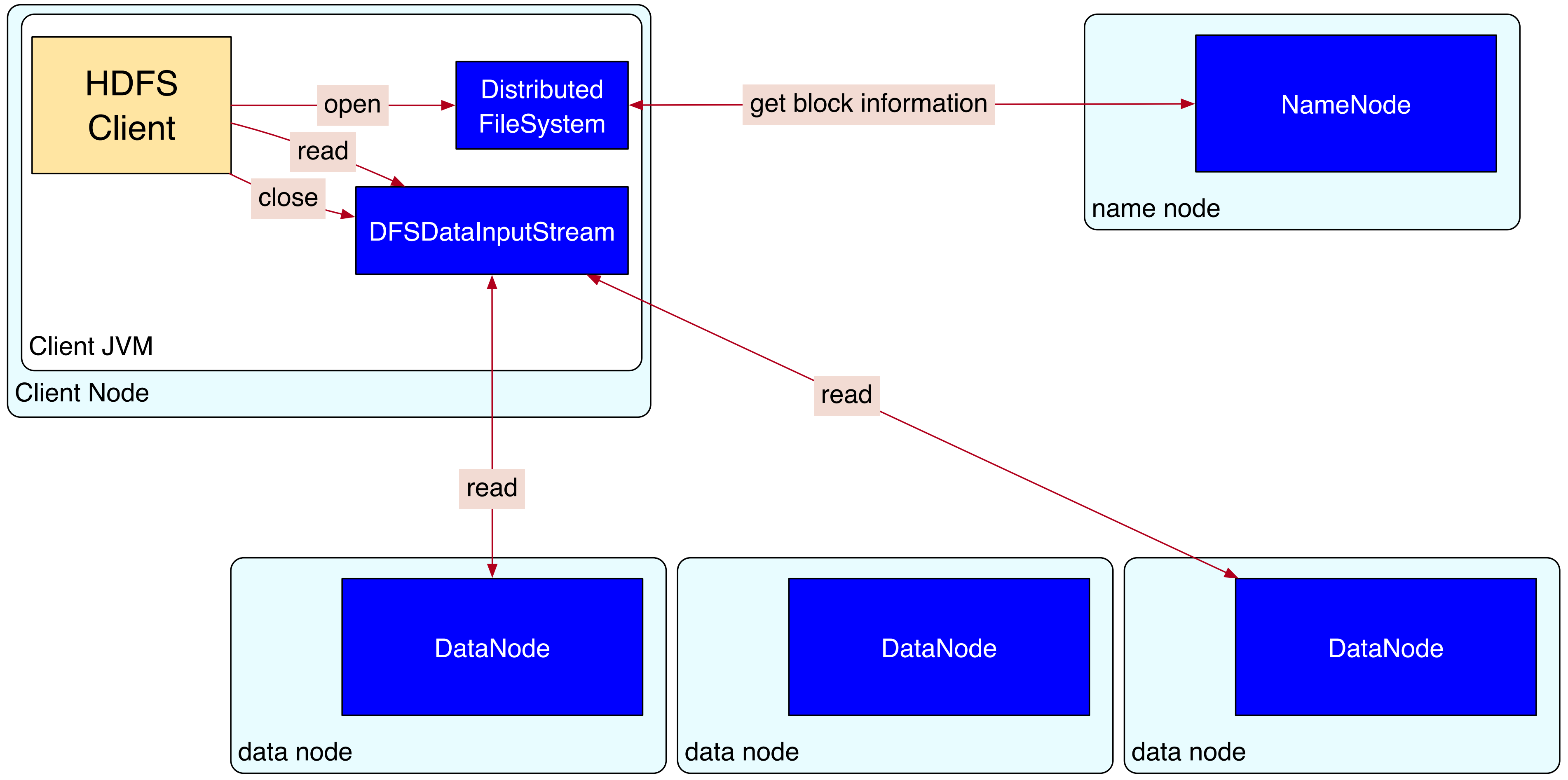
3. DistributedFileSystem returns a DFSDataInputStream, which wraps a DFSInputStream to manage I/O from NameNode and DataNode

HDFS Interface



4. HDFS Client calls “read” to various DataNodes repeatedly. After stream is ended, DFSDataInputStream will close the connection to the DataNode and access the DataNode with the next block.

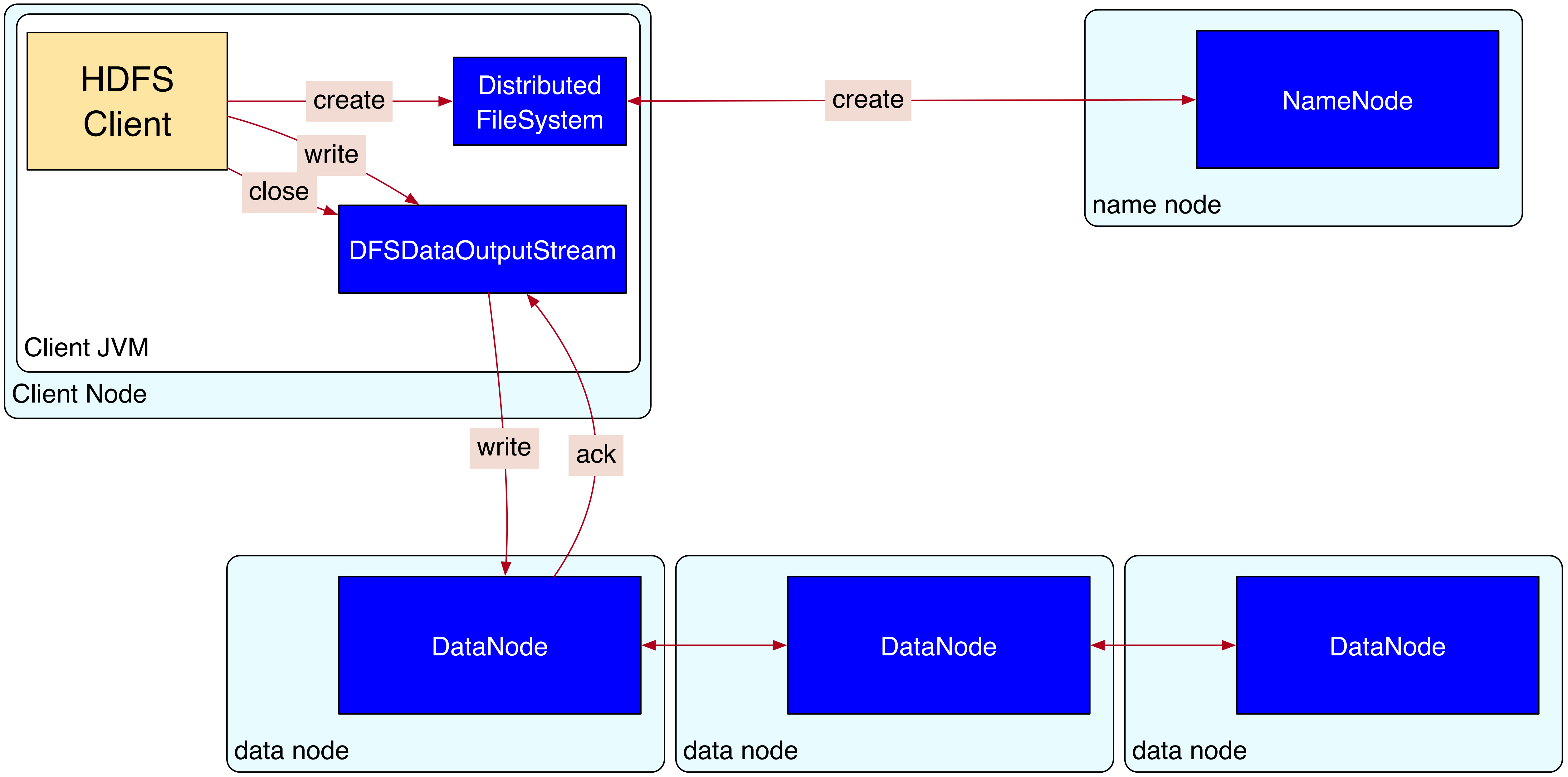
HDFS Interface



5. HDFS client will eventually close the DFSInputStream

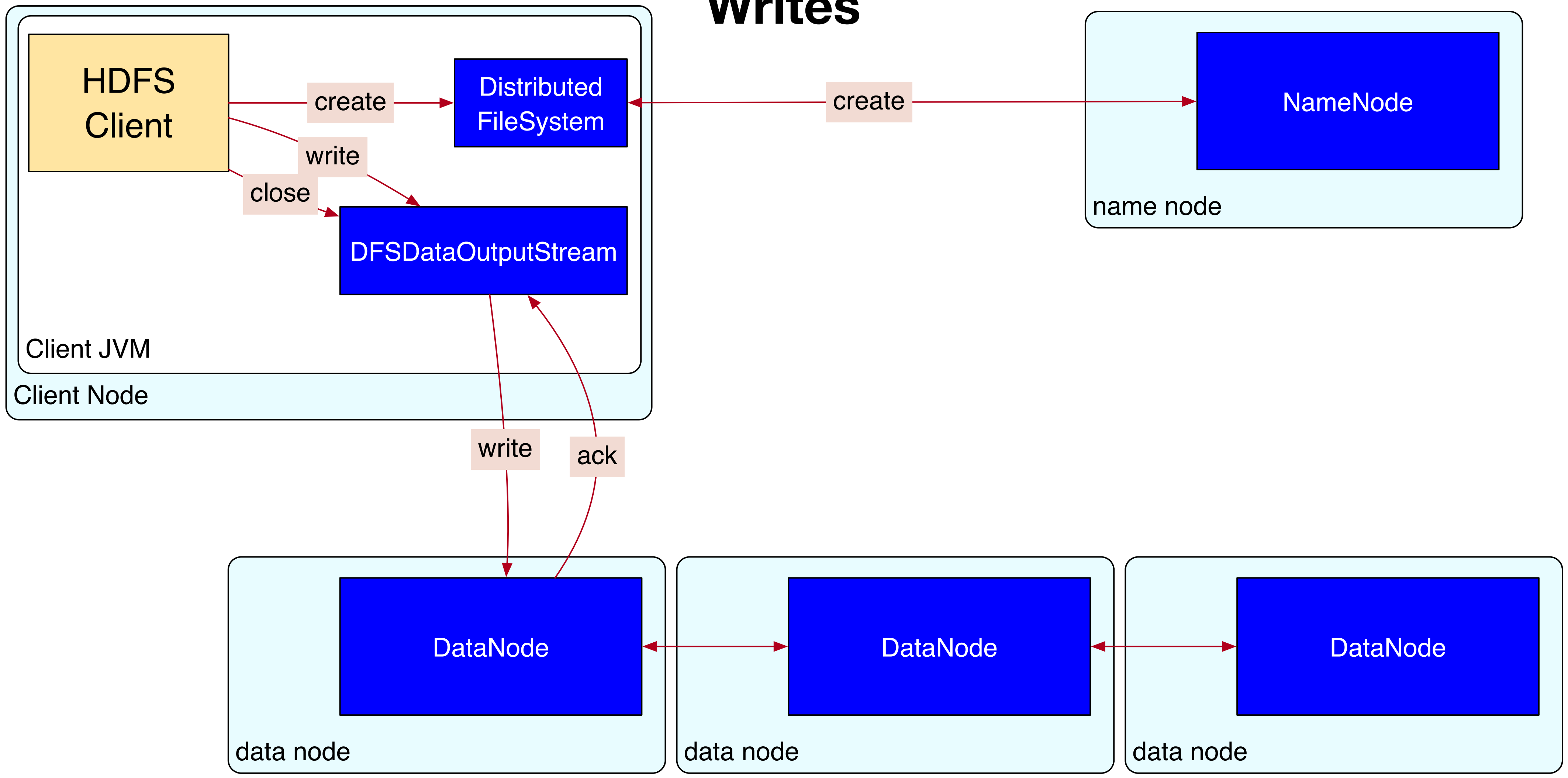
HDFS Interface

Writes



HDFS Interface

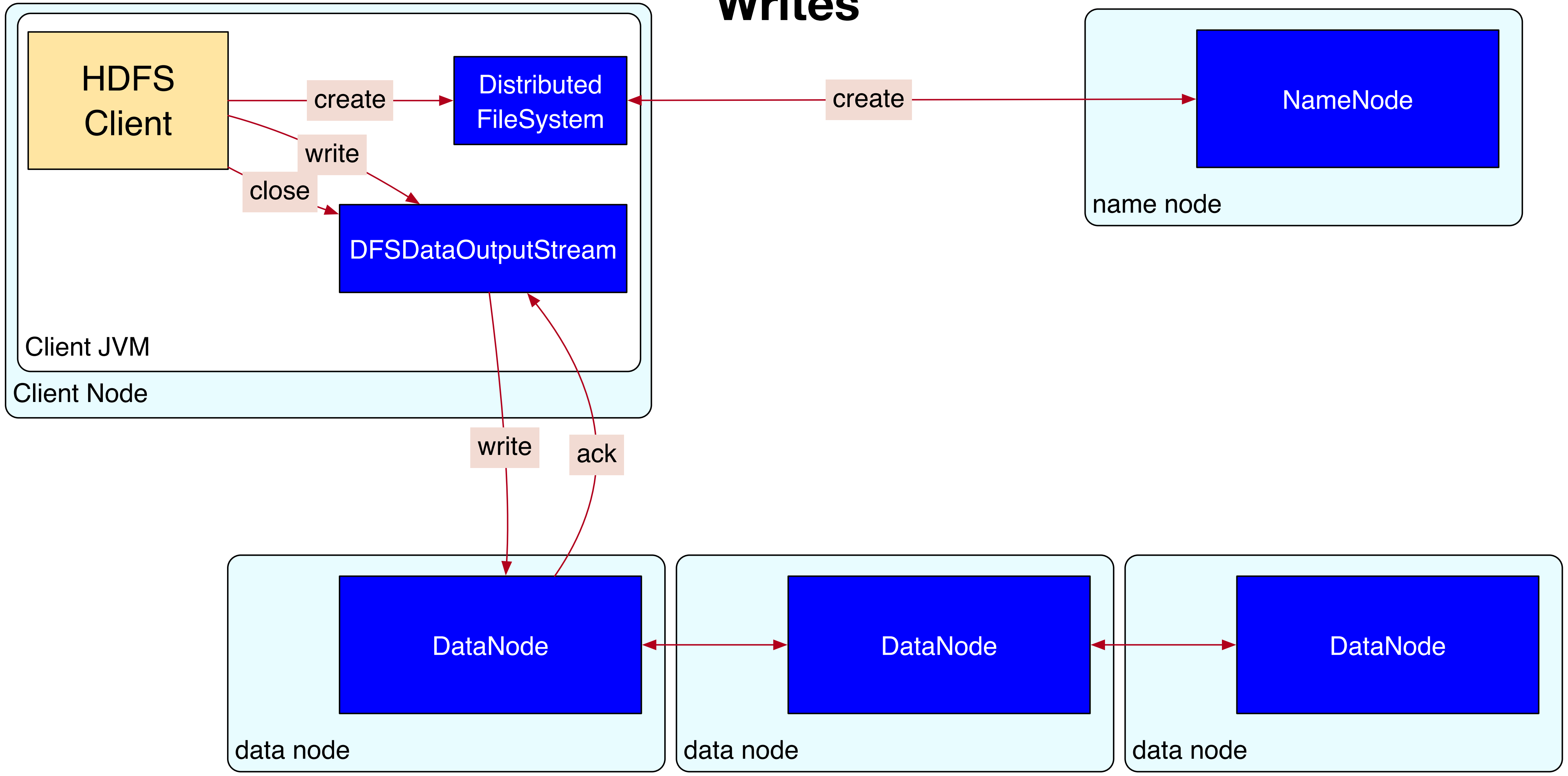
Writes



1. Client creates a file in DistributedFileSystem

HDFS Interface

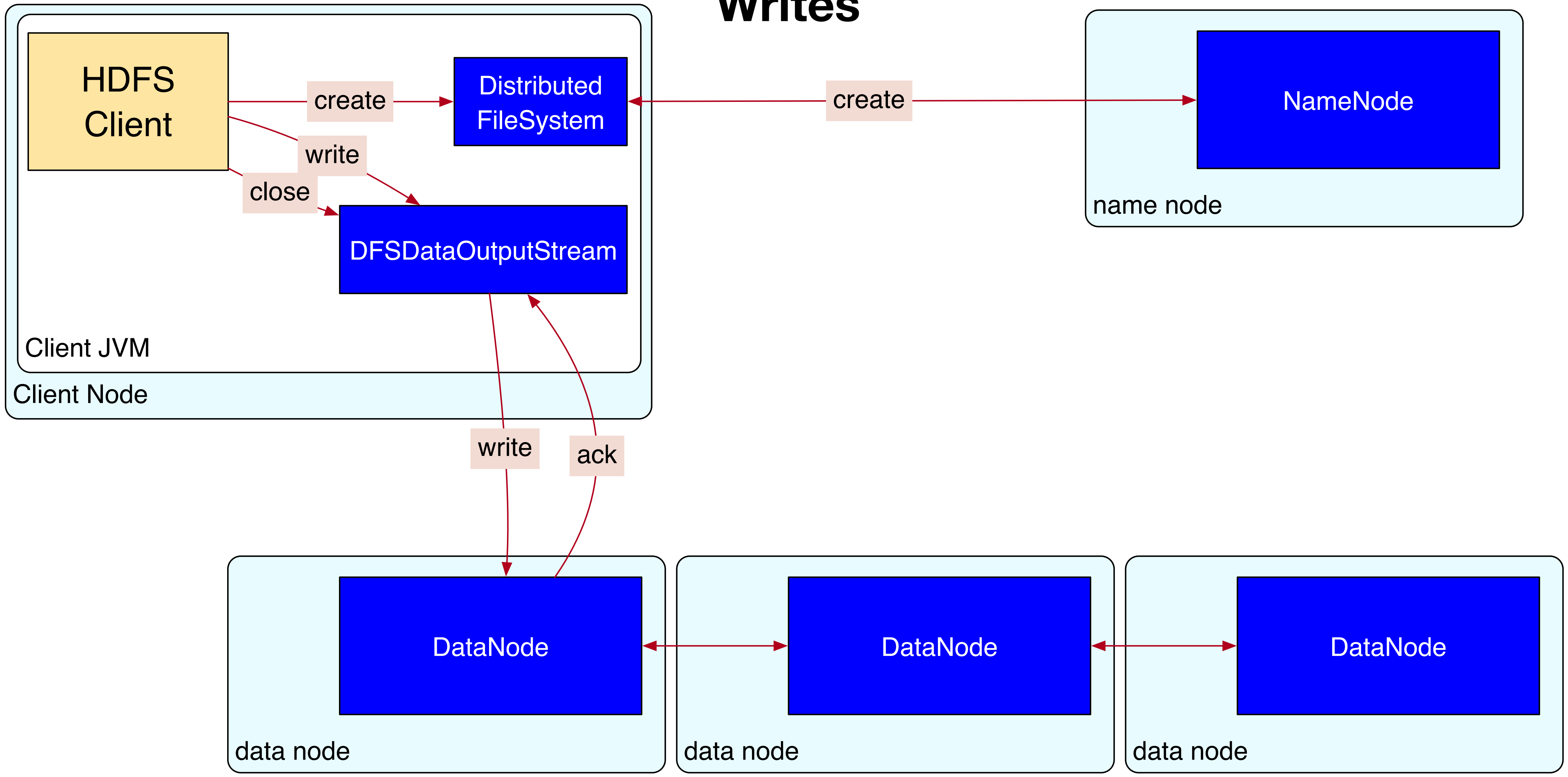
Writes



2. DistributedFileSystem makes an RPC call to the NameNode. NameNode checks whether the file can be created. If not, the nack results in DistributedFileSystem throwing an exception.

HDFS Interface

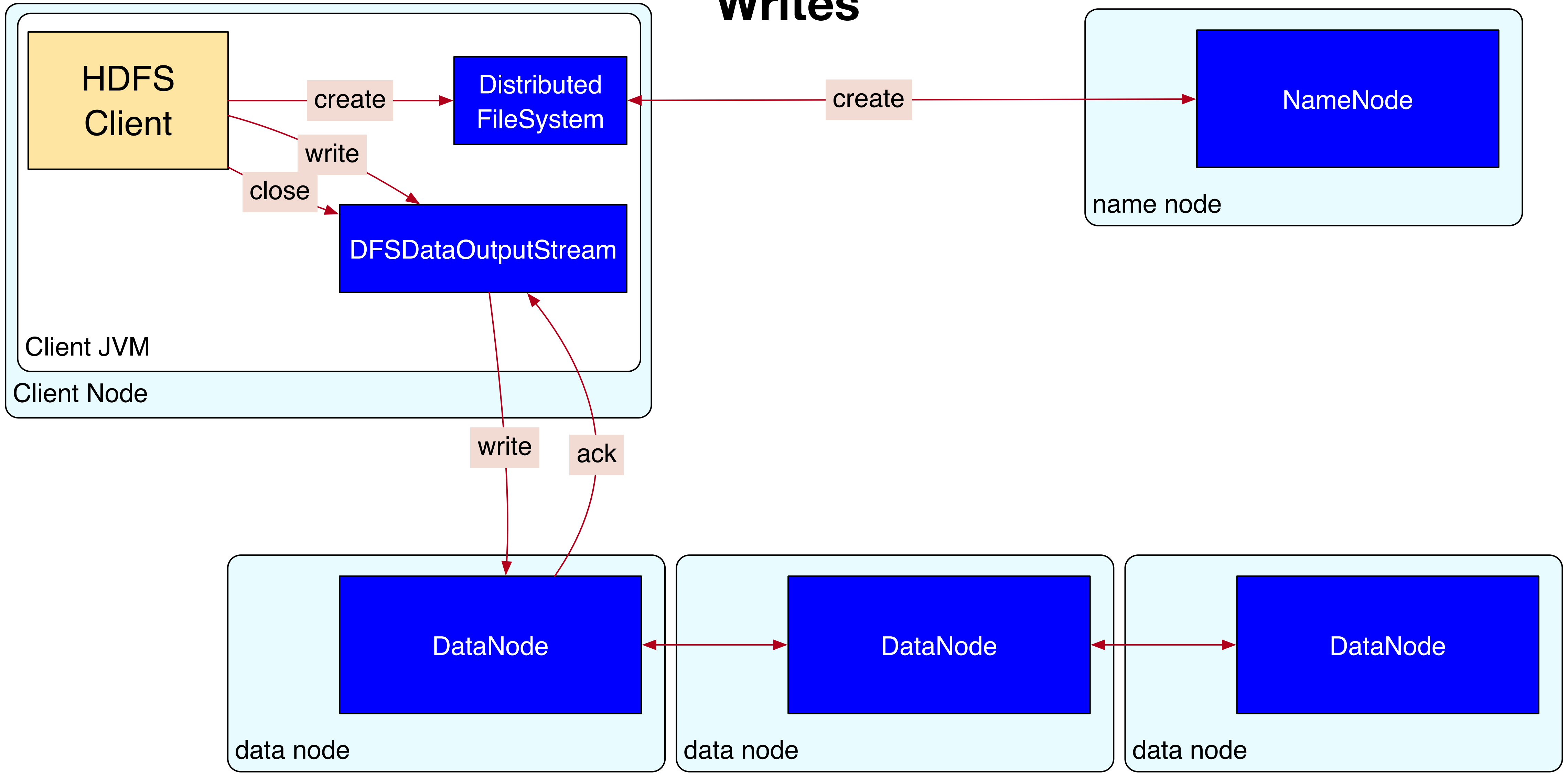
Writes



3. Client writes to DFSDataOutputStream. DFSDataOutputStream breaks output into packets, written to a *data queue*. The data queue is consumed by the DataStreamer, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas.

HDFS Interface

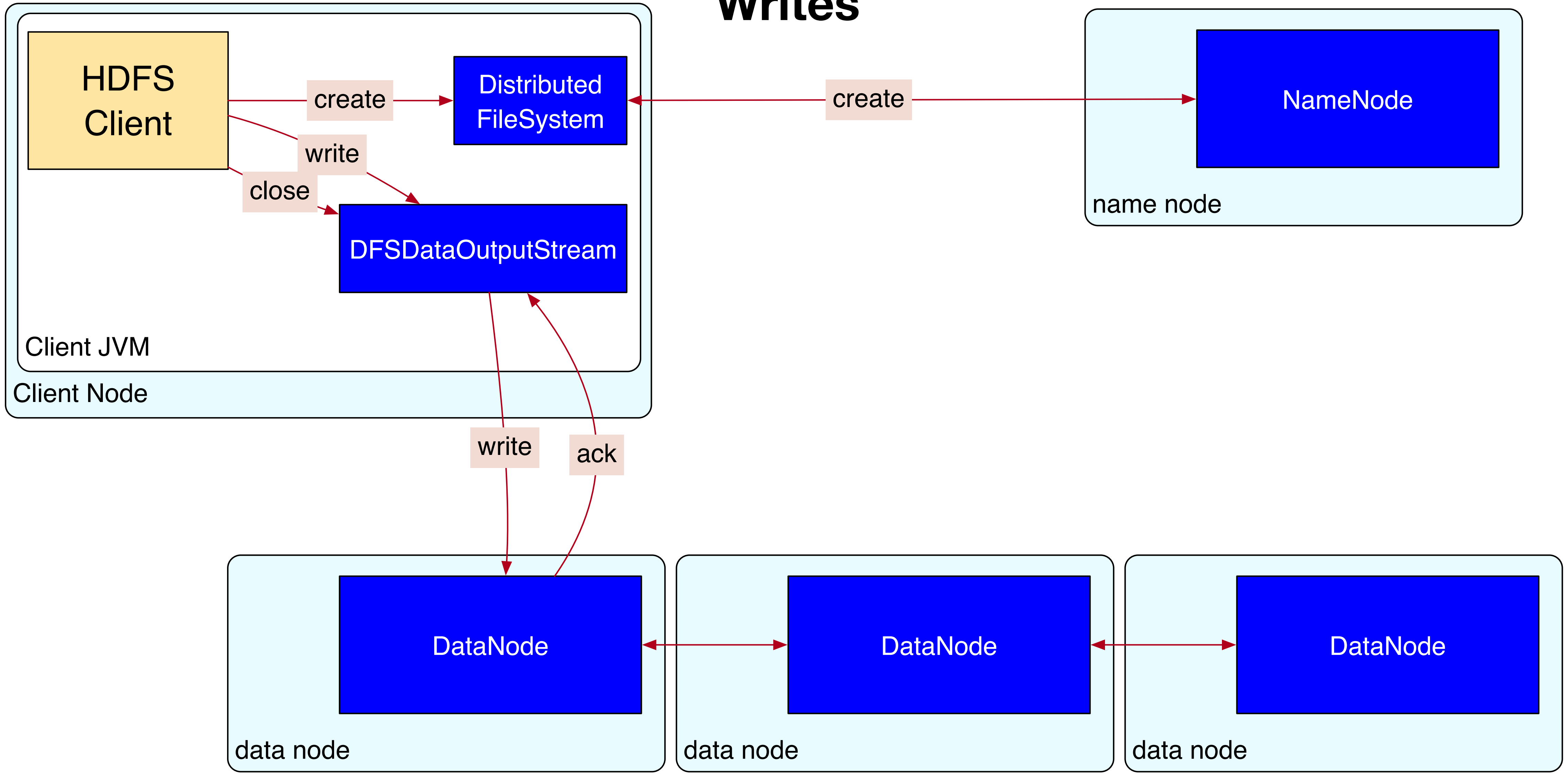
Writes



4. DataStreamer streams data to the first DataNode, which stores each packet and forwards them to the second DataNode, which forwards them to the third DataNode, etc.

HDFS Interface

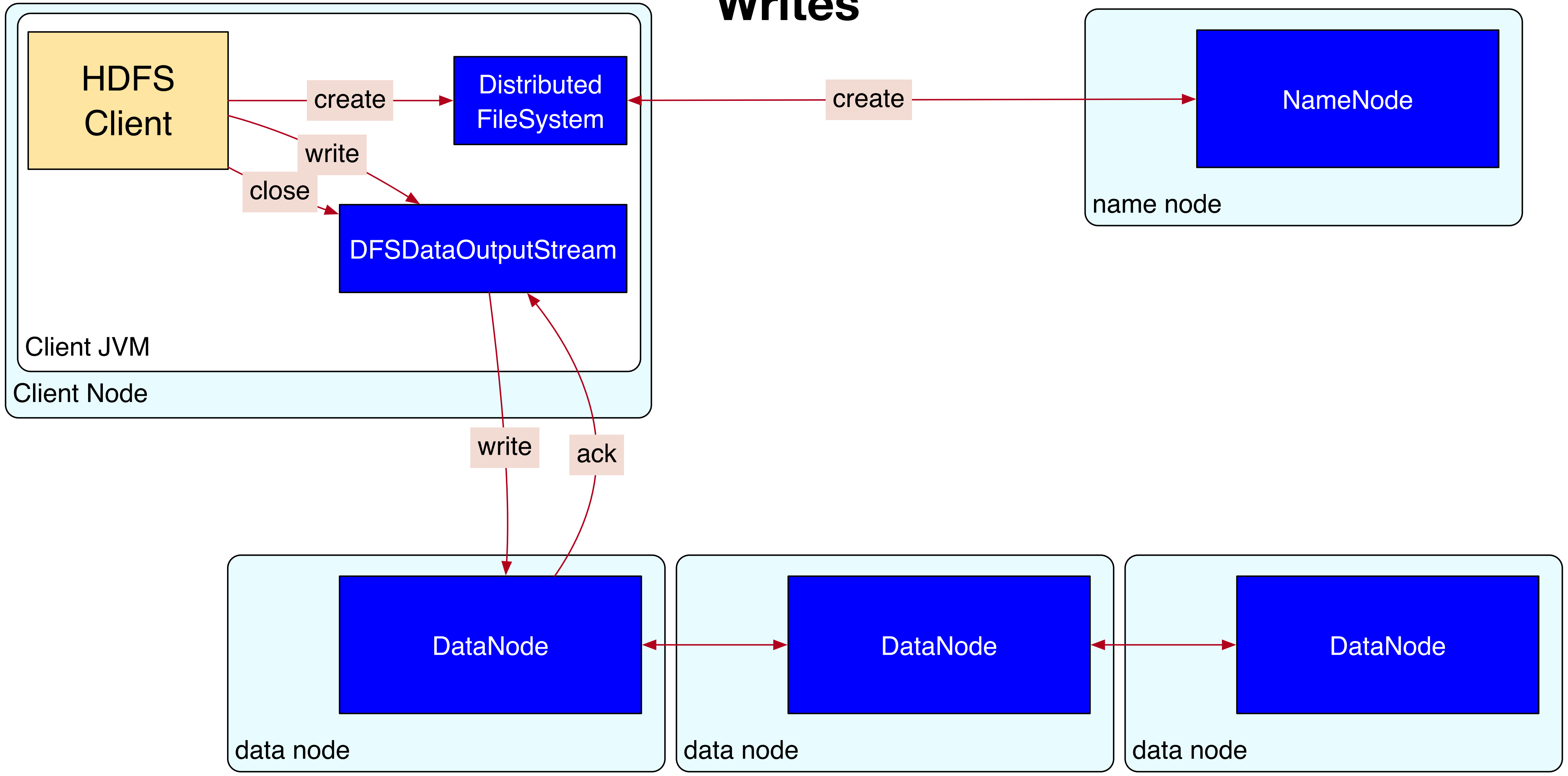
Writes



5. DFSDataOutputStream maintains an internal queue of packets waiting for acks, the *ack queue*.

HDFS Interface

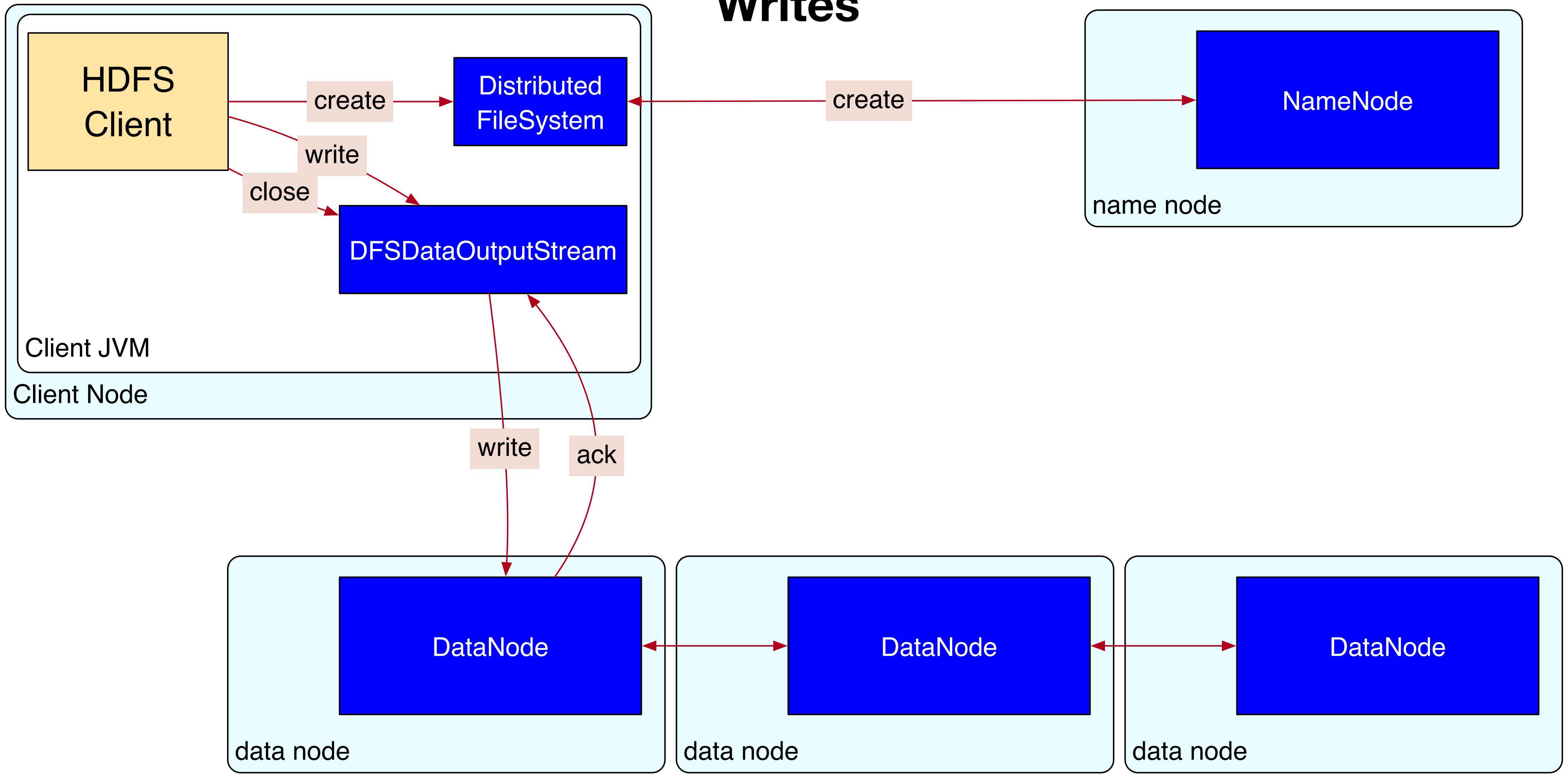
Writes



6. Eventually, the HDFS Client closes the file. It receives an ack if the ack queue is empty.

HDFS Interface

Writes



7. The NameNode is informed by the DataStreamer of all locations

HDFS Interface

- If there is a failure of a DataNode:
  - Pipeline is closed.
  - Packets in the ack-queue are added to the front of a new pipeline containing surviving DataNodes
    - So, no packets are missed
  - Current block is given a new identifier, communicated to the NameNode so that a partially written block can be deleted when the DataNode recovers

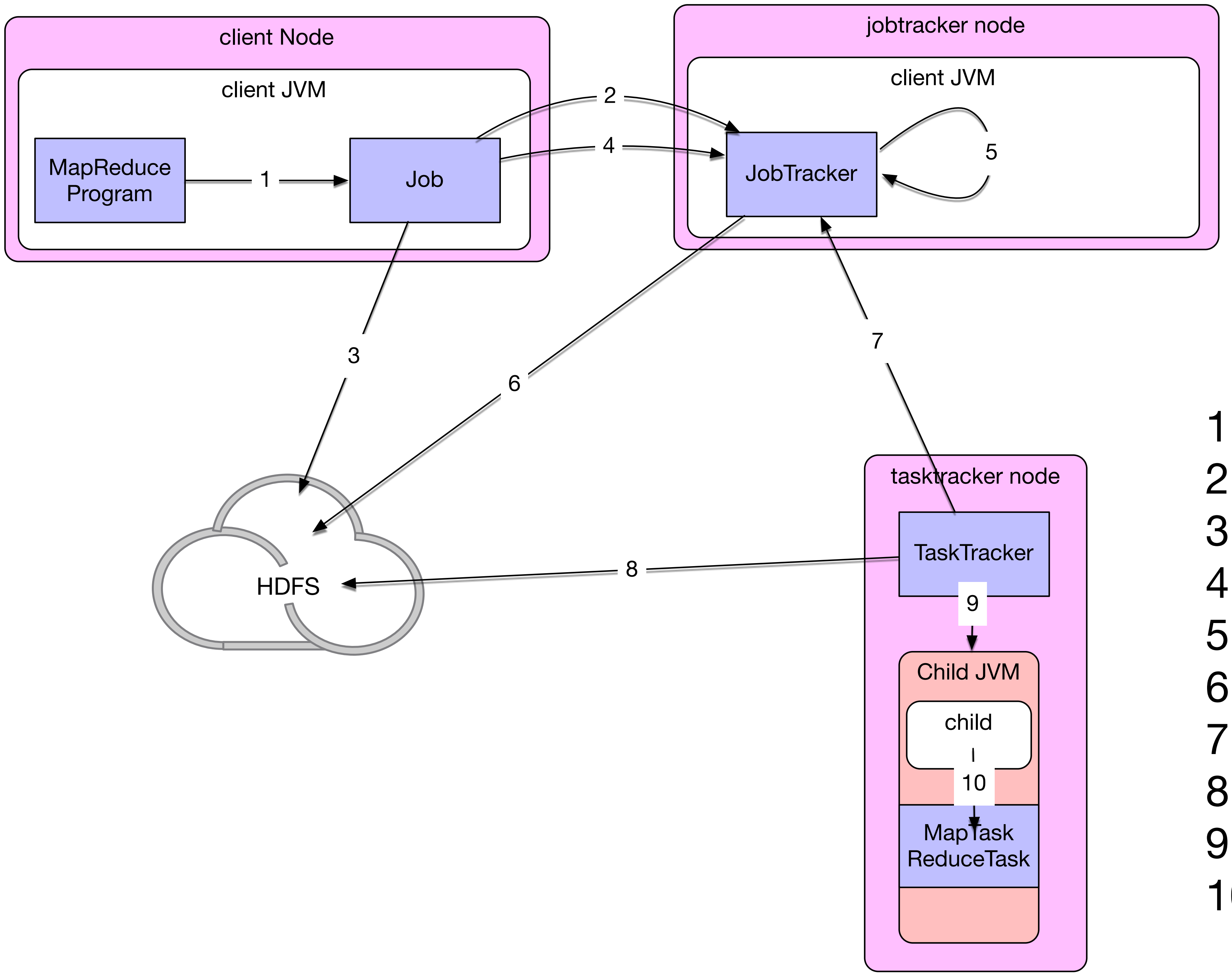
HDFS Interface

- Coherency:
  - After creating a file, it takes time for the NameNode to know about the blocks that have been written.
    - This is visible to users
  - Flushing only means that the NameNode knows about the file

Hadoop

- Classic Map-Reduce
  - Client that submits the map-reduce job
  - Job trackers which coordinate the job run
  - Task trackers that run the tasks that the job has been split into
  - Distributed file system (HDFS — Hadoop file system) for file sharing between entities

Hadoop Classic

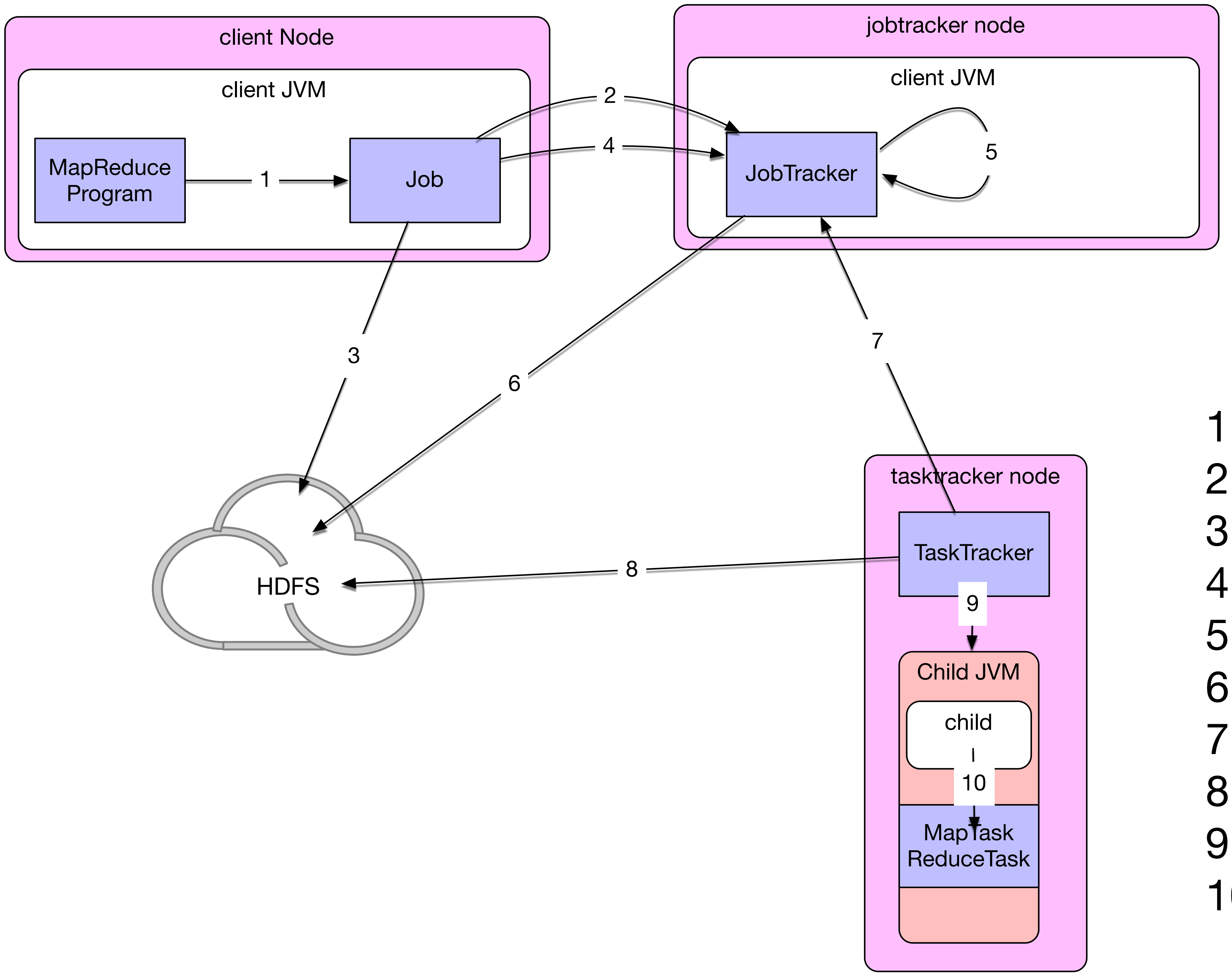


1: run job
2: get new job id
3: copy job resources
4: submit job
5: initialize job
6: retrieve input splits
7: heartbeat
8: retrieve job resources
9: launch
10: run

Hadoop

- Job submission
    - Creates an internal JobSummitter instance
        - JobSubmitter
            - asks jobtracker for a new jobid
            - check the output specifications of the job
            - computes the input split for the job
            - copies the resources needed for the job
            - tells the jobtracker that the job is ready for submission

Hadoop Classic



1: run job
2: get new job id
3: copy job resources
4: submit job
5: initialize job
6: retrieve input splits
7: heartbeat
8: retrieve job resources
9: launch
10: run

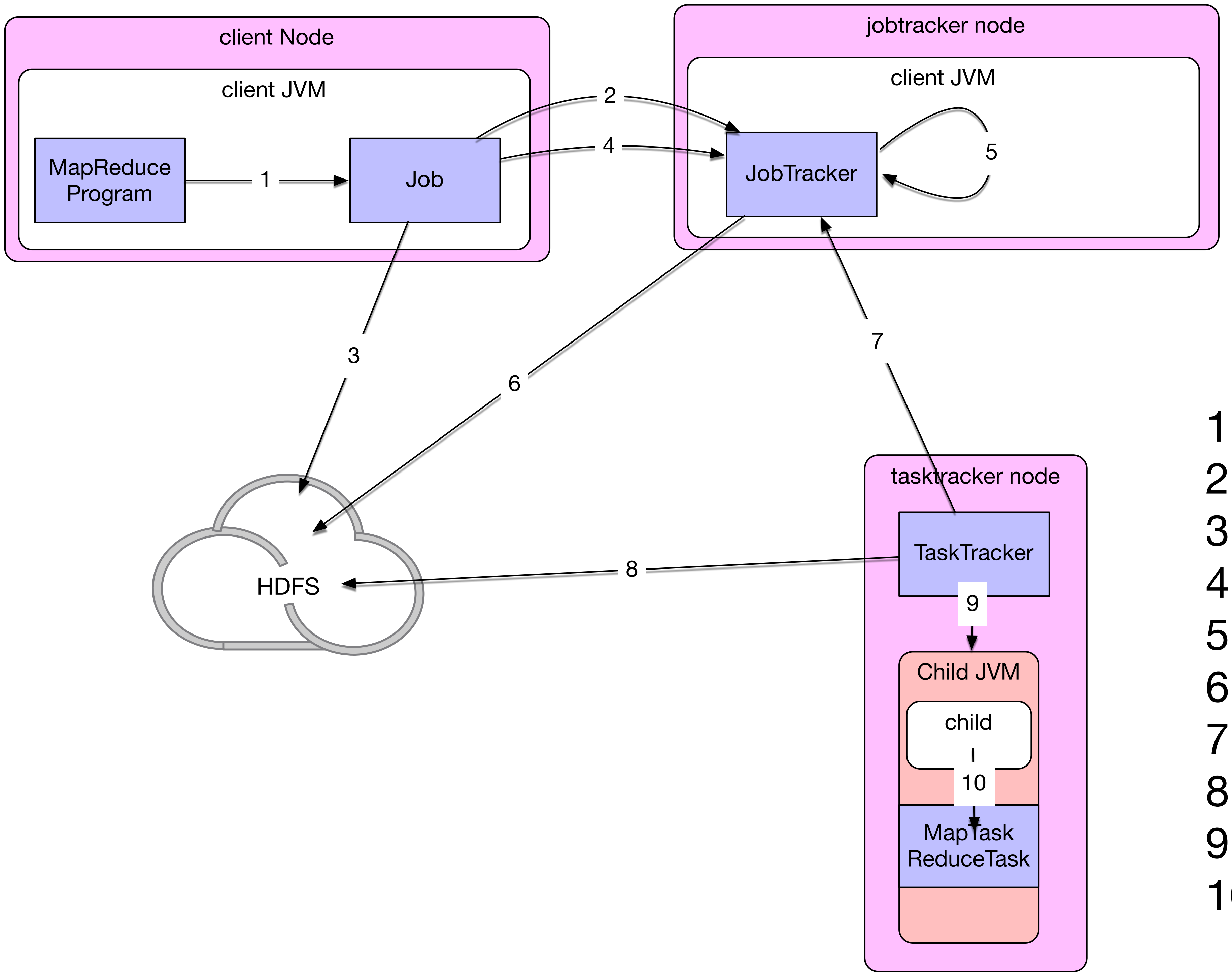
Hadoop

- Jobtracker receives call from submitJob( )
  - places it in internal queue
  - retrieves the input splits computed by the client
  - creates a map task for each split
    - number of mappers is set by the mapred.reduce.tasks
  - runs job setup task
  - runs job cleanup task

Hadoop

- Task assignment
  - Tasktrackers periodically send heartbeat to jobtracker
    - Includes message if task is done so that node can get a new job
  - Tasktrackers have a set number of map and reduce jobs that they can handle
  - To create a reduce task, the jobtracker simply goes through the list of reduce tasks and assigns one
  - Preference is given to data-local (reduce on the same node) or rack-local

Hadoop Classic



1: run job
2: get new job id
3: copy job resources
4: submit job
5: initialize job
6: retrieve input splits
7: heartbeat
8: retrieve job resources
9: launch
10: run

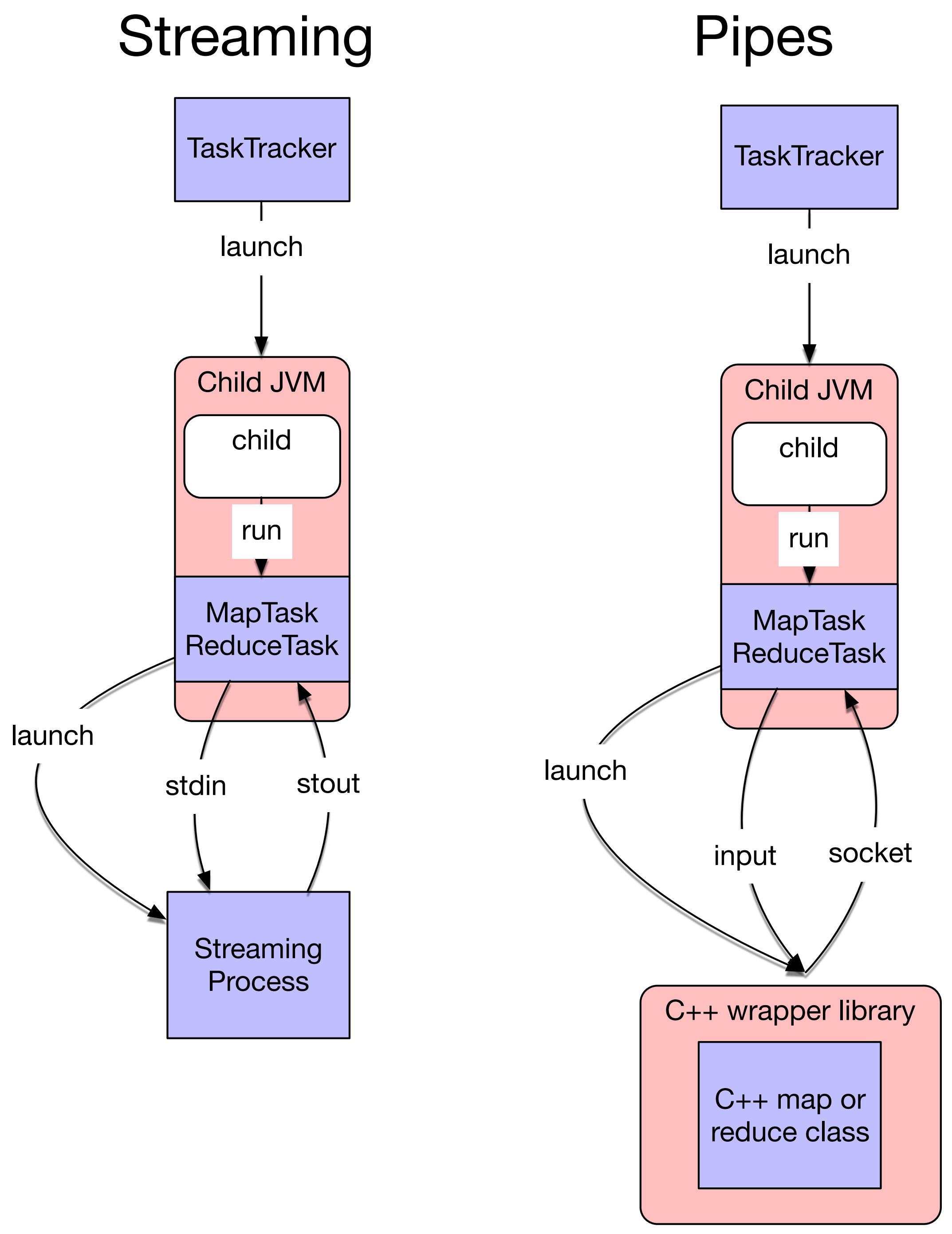
Classical Hadoop

- Task execution
  - Tasktracker localizes the job JAR from the file system
  - It copies any files needed from the distributed cache
  - Creates instances of TaskRunner to run the task
    - TaskRunners launch a new Java Virtual Machine
    - Child informs parent of progress

Classical Hadoop

- Streaming and pipes run special map and reduce tasks
  - Streaming:
    - communicates with process using standard input and output streams
  - Pipes:
    - Pipes task listens on socket
    - passes C++ process a port number
  - In both cases
    - Java process passes input key-value pairs to the external process

Classic Hadoop Streaming and Piping



Classical Hadoop

- Progress and status updates
  - Map-Reduce jobs can take hours
  - Each job has a status
  - System estimates progress for each task
    - Mappers: per cent input dealt with
    - Reducers: More complicated, but estimates are possible
  - Tasks use set of counters for various events
    - Can be user defined — see below

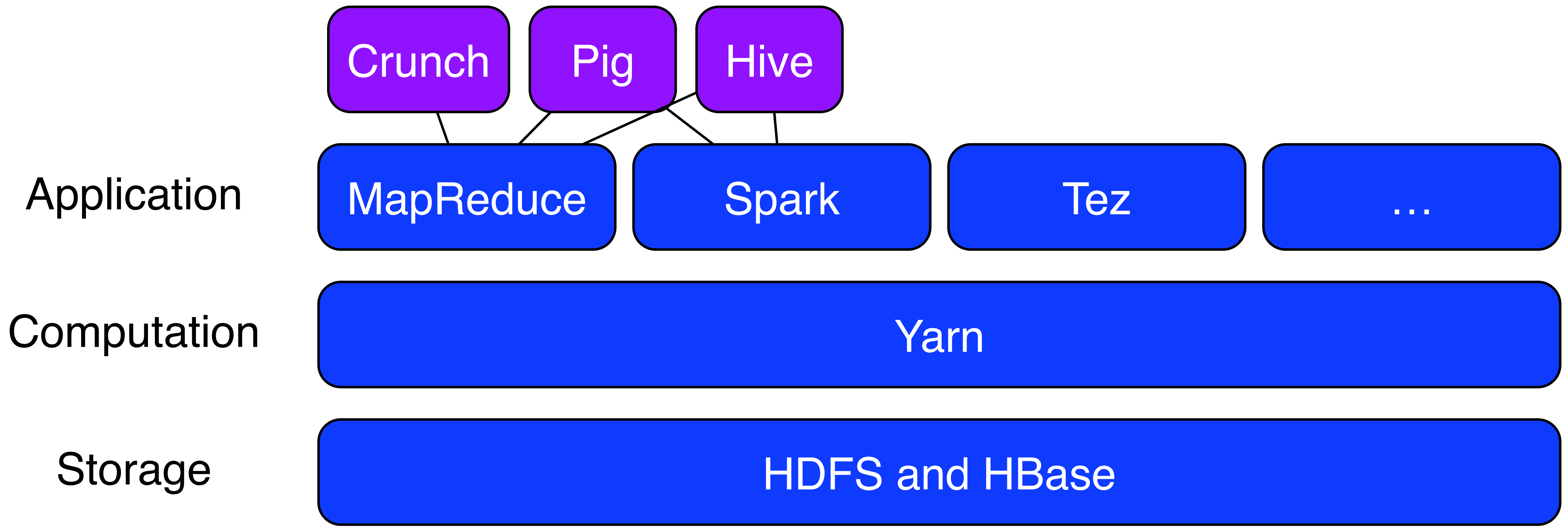
Classical Hadoop

- Job completion
  - If job tracker receives notification that last task has completed
    - Job status changes to “successful”
    - Job statistics are sent to console

Apache Yarn

Yet Another Resource Negotiator

Yarn Design


Crunch
Pig
Hive
Application
MapReduce
Spark
Tez
…
Computation
Yarn
Storage
HDFS and HBase

Yarn

- Yarn provides services via two daemons
  - Resource manager (one per cluster)
  - Node managers (running on all nodes in a cluster)
- Node managers run *containers*
  - Container = application-specific process with resources
    - Typically a Unix process or a Linux cgroup

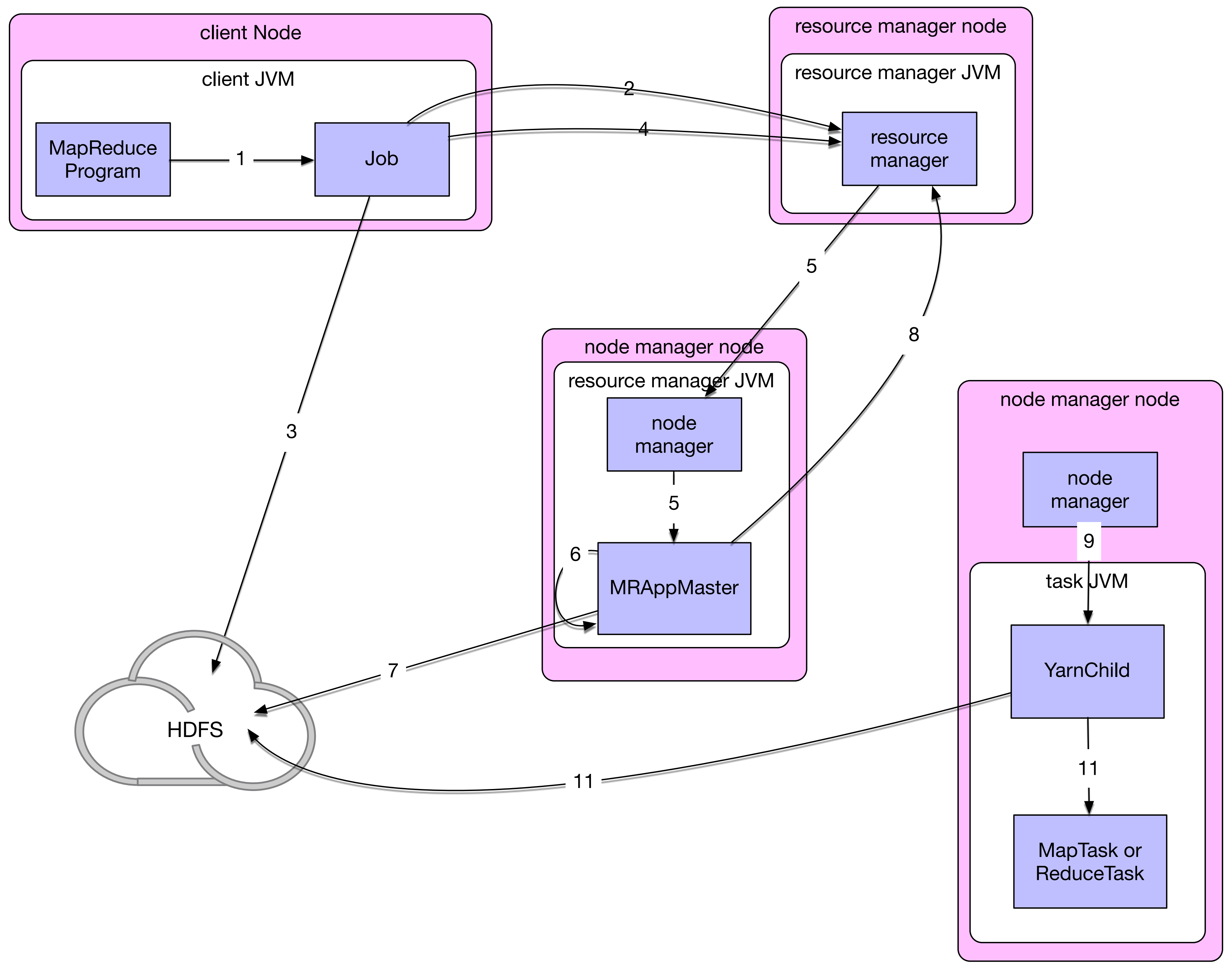
Yarn: Hadoop 2

- Classic structure runs into bottlenecks at about 4000 nodes
- YARN: Yet Another Resource Negotiator
  - YARN splits jobtrackers into various entities:
    - Resource manager daemon
    - Application master

Yarn: Hadoop 2

- Each application instance has a dedicated application master

Yarn: Hadoop 2

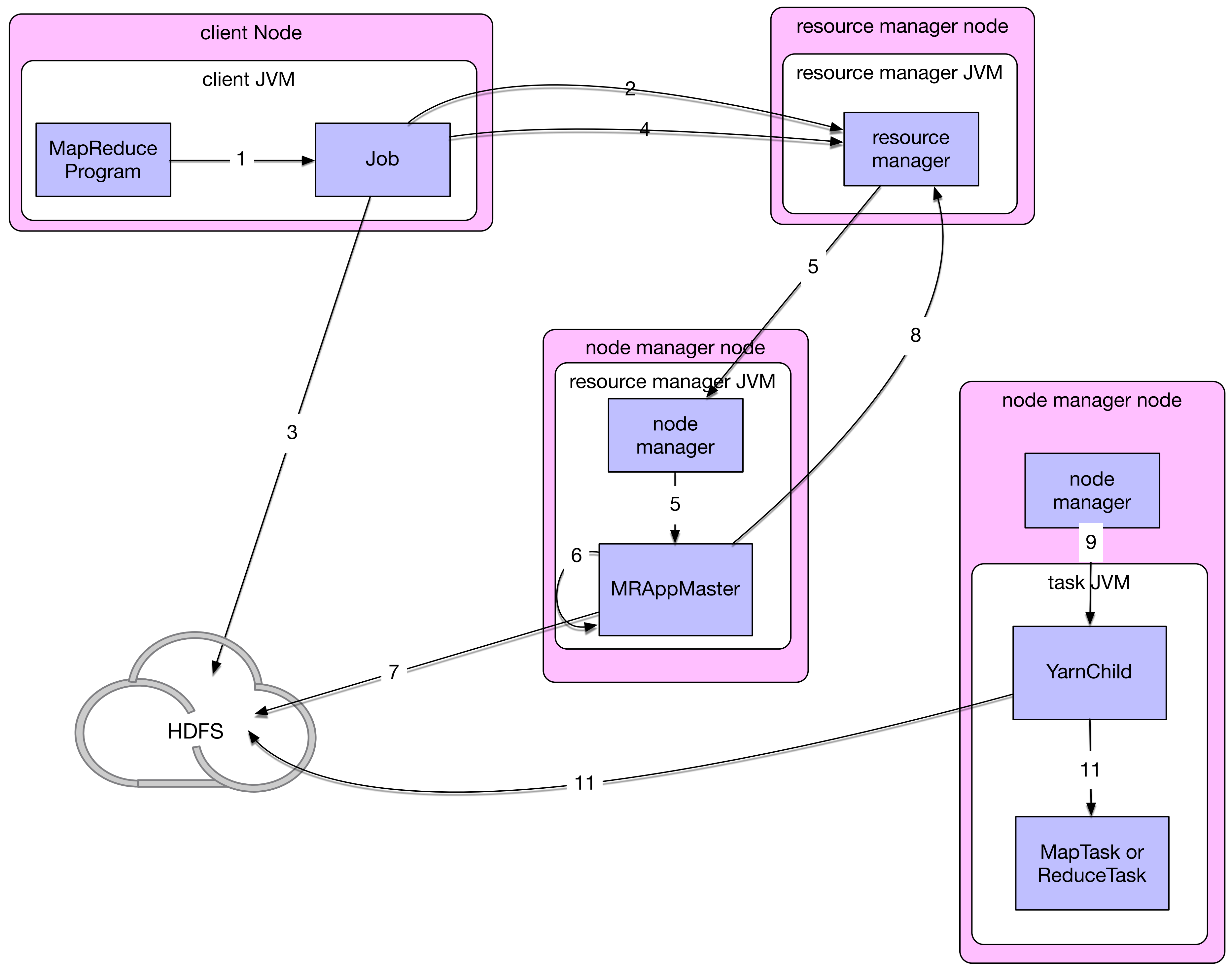


1: run job
2: get new application
3: copy job resources
4: submit application
5: start container and launch
6: initialize job
7: retrieve input splits
8: allocate resources
9: start container and launch
10: retrieve job resources
11: run

Yarn: Hadoop 2

- Job submission as before
- When resource manager receives a call to submitApplication() hands off to scheduler
- Scheduler allocates a container
- Resource manager launches application master process there (5)
-

Yarn: Hadoop 2



1: run job
2: get new application
3: copy job resources
4: submit application
5: start container and launch
6: initialize job
7: retrieve input splits
8: allocate resources
9: start container and launch
10: retrieve job resources
11: run

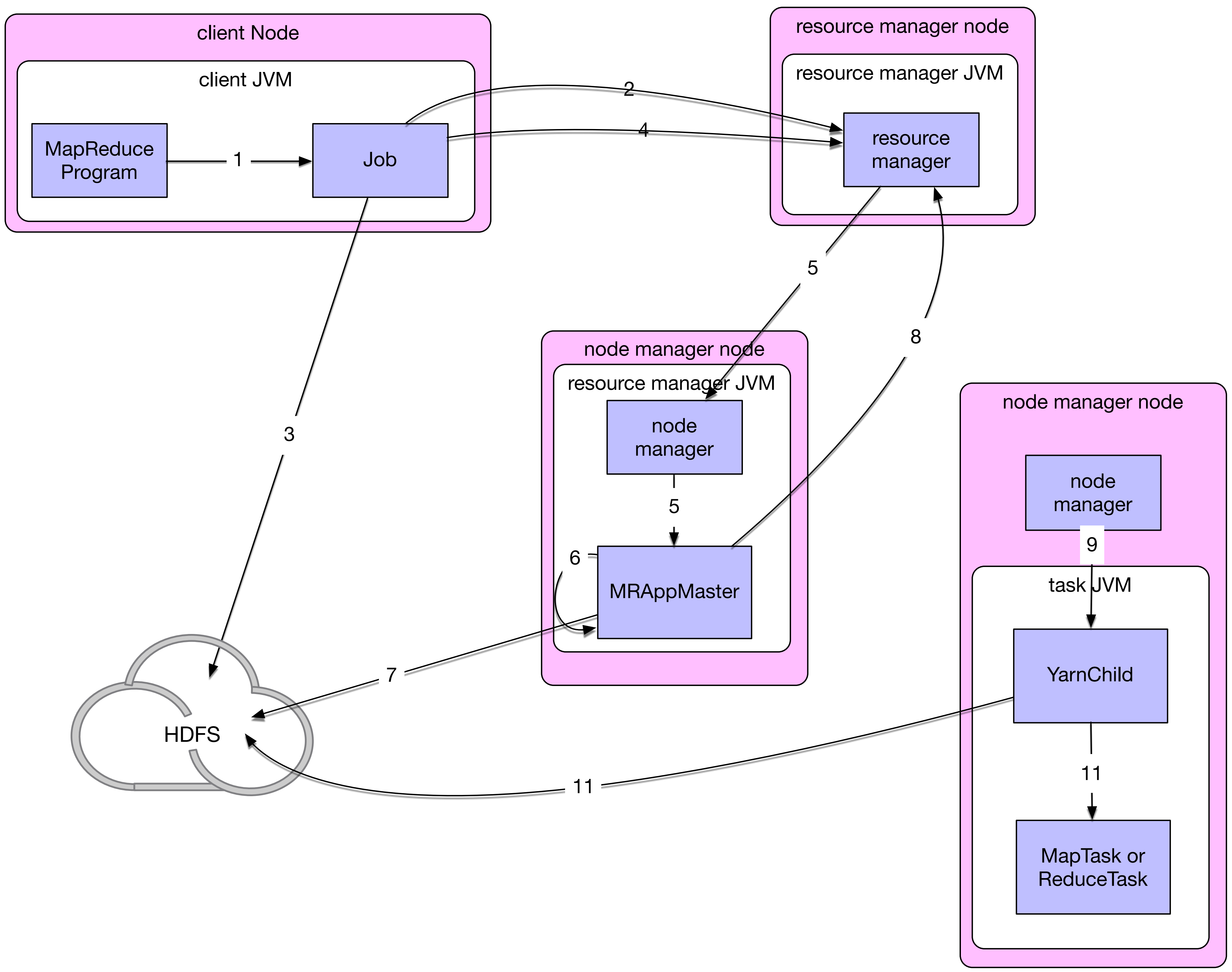
Yarn: Hadoop 2

- Application master
  - initializes the job by creating book-keeping objects
    - to receive and report on progress by individual tasks
  - Retrieves input splits
  - Creates a map task object for each split
  - Creates a number of reduce tasks (mapreduce.job.reduces)
  - Decides how to run job
    - Small jobs might be run in the same node
    - *Uber* tasks

Yarn: Hadoop 2

- Application master requests containers for all map and reduce tasks from resource manager
  - Scheduler gets enough information to make smart decisions
  - One of the selling points of Yarn: Resource allocation is much smarter

Yarn: Hadoop 2



1: run job
2: get new application
3: copy job resources
4: submit application
5: start container and launch
6: initialize job
7: retrieve input splits
8: allocate resources
9: start container and launch
10: retrieve job resources
11: run

Yarn: Hadoop 2

- Task execution:
    - Application master starts container by contacting the node manager
    - Streaming and pipes work in the same way as Classical MapReduce

Yarn: Hadoop 2

- Progress and Status reports
  - tasks report progress and status to application masters
  - (Classical: reports move from child through tasktracker to jobtracker for aggregation)
  - client polls application master every second

Failures in Classic Hadoop

- Child task is failing:
    - Throwing a runtime exception
        - JVM through task master informs client
        - Taskmaster can take on another task
        - Streaming tasks are marked as failed
    - Sudden exit of child JVM
        - Taskmaster notes exit and marks attempt as failed

Failures in Classic Hadoop

- Child task is failing
  - Hanging tasks
    - Tasktracker notices lack of progress updates and marks task as failed
      - Normal timeout period is 10 minutes

Failures in Classic Hadoop

- When jobtracker is notified that a task attempt has failed
    - jobtracker reschedules task elsewhere
    - jobtracker does try a maximum of four times
    - Client can specify the percentage of tasks that are allowed to fail

Failures in Classic Hadoop

- Users, jobtrackers can kill task attempts
  - E.g. speculative execution can kill duplicates
  - E.g. Tasktracker has failed

Failures in Classic Hadoop

- Tasktracker failure
  - Tasktracker then no longer sends heartbeats
  - Jobtracker removes tasktracker from its pool
  - Jobtracker arranges for map tasks to restart
    - Because there might be no access to the local results
  - Tasktrackers can be blacklisted if too many tasks there fail

Failures in Classic Hadoop

- Jobtracker failure
  - No mechanism in Hadoop to deal with this type of failure

Failures in YARN

- Task failures like before:
  - Propagated back to application master
  - Application master marks them as failed
  - Hanging tasks are discovered by application master

Failures in YARN

- Application master failure
  - Several attempts for a task to succeed
  - Application master sends heartbeats to Resource manager
  - Resource manager can restart application master elsewhere

Failures in YARN

- Node manager failure
  - Application manager will know due to lack of hearbeats

Failures in YARN

- Resource manager failure
  - Resource manager is hardened by using checkpointing to save state

Job Scheduling

- First implementations just used FIFO
- Later, priorities were introduced
- Fair scheduler: every user gets equal access to the capacity of the cluster
- Capacity scheduler: made up of queues
  - Each queues is run like a fair scheduler
  - Gives administrator more control over how different users are treated

Hadoop Data Integrity

- Use hashes a.k.a. checksum to verify integrity
  - E.g. CRC32, CRC32C
  - Datanodes check integrity by verifying checksums when data is accessed
  - Periodically scans all data

Hadoop Compression

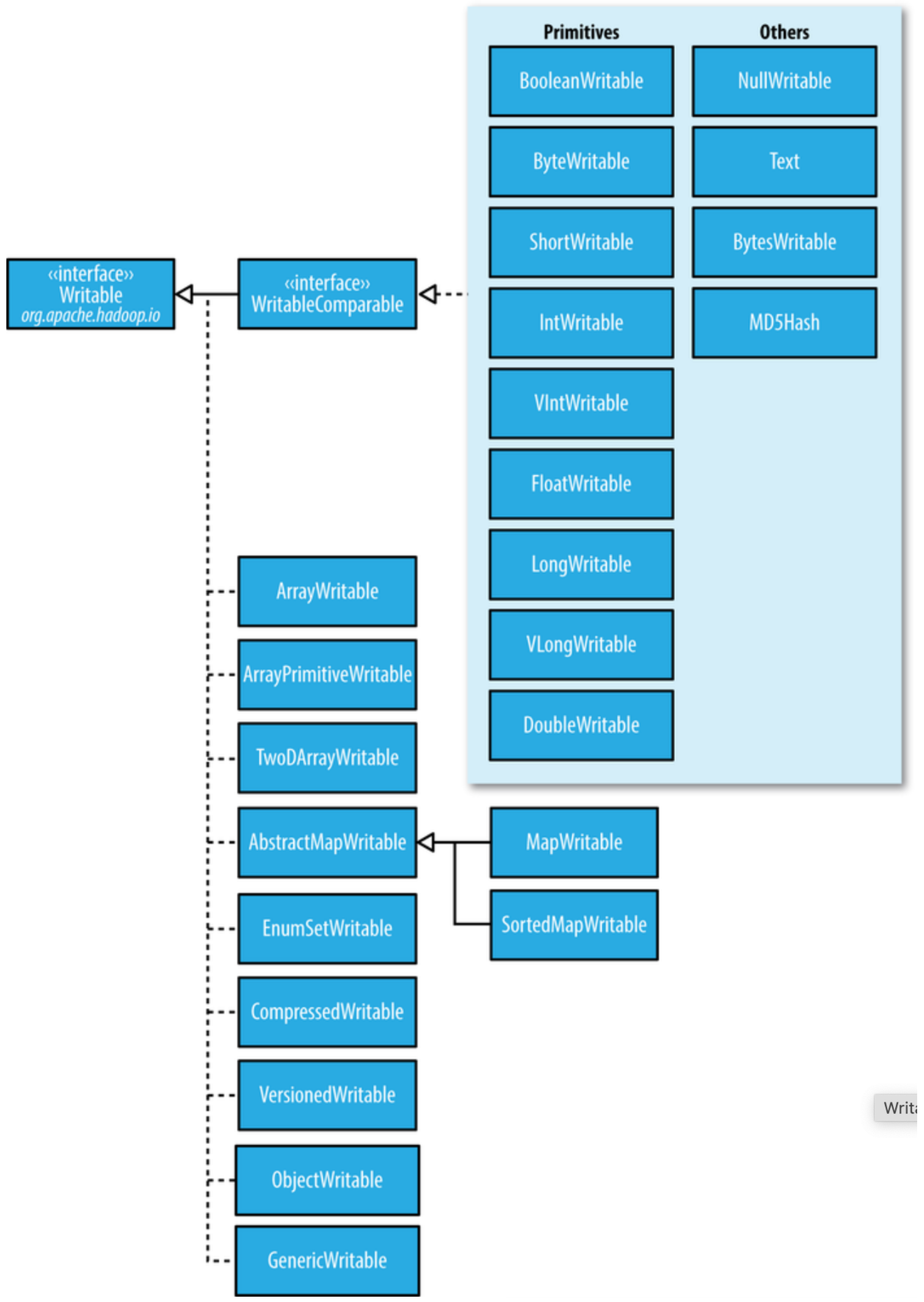
- Hadoop uses a variety of compression algorithms
- Splittable compression:
    - Can Hadoop blocks be uncompressed independently?

Hadoop Compression

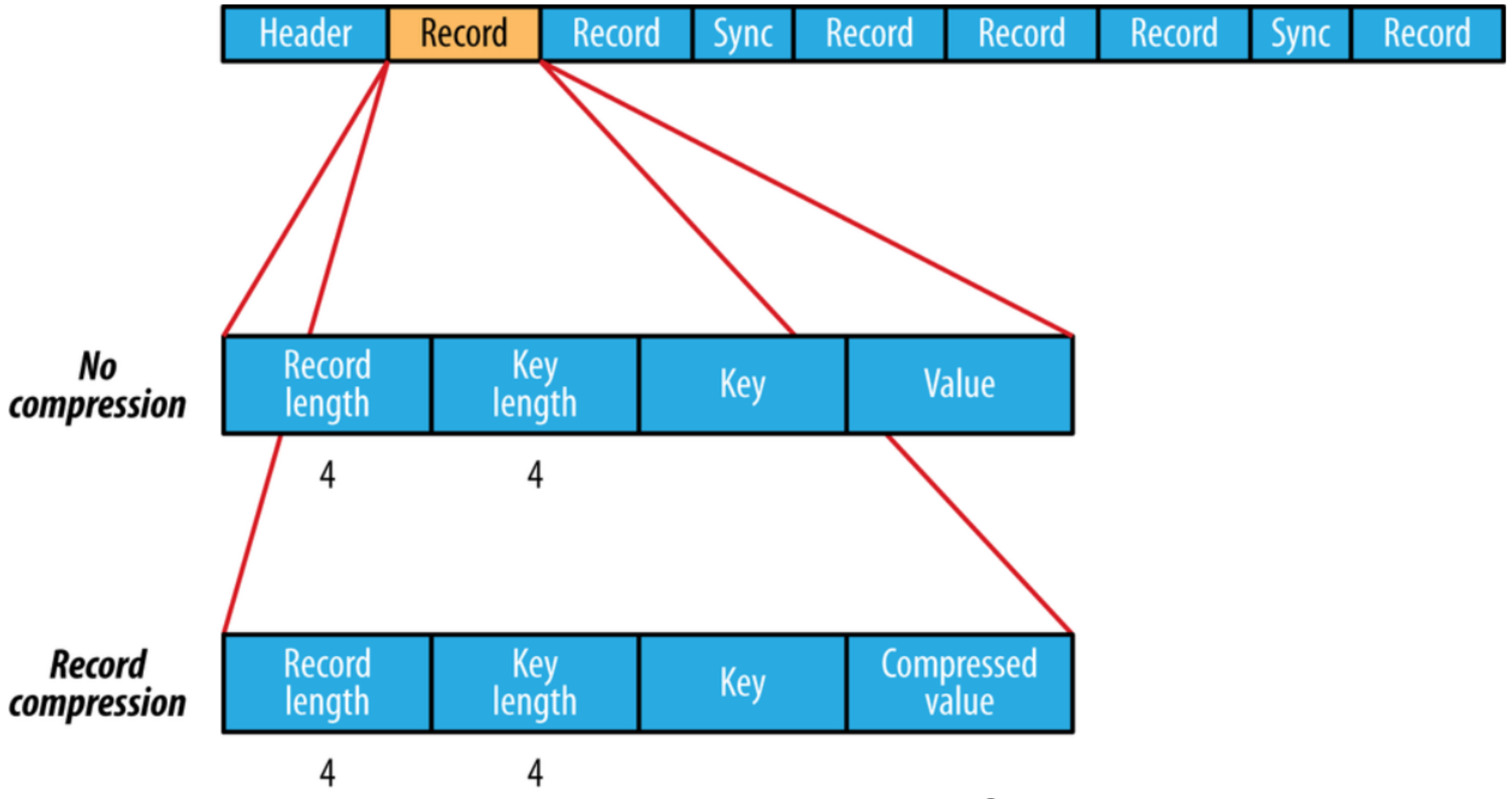
- While Hadoop hides the movement of data, data still needs to be moved
- Compressing intermediate results can give large performance gains

Hadoop Serialization

- Serialization:
  - Turning structured data into a byte stream, e.g. for storage
- Writable:
  - Hadoop supports a variety of encodings, encompassed in the Writable class hierarchy



Example: The SequenceFile



Using Record Compression

Avro: Language-Neutral Data Serialization

- Apache Avro
  - Data format which interfaces for many languages
  - Avro data is described in a language-independent schema
    - The schema itself is usually written in JSON
    - Data is usually encoded in a binary format

Avro: Language-Neutral Data Serialization

- Avro has schema-resolution capabilities:
  - Schema used to read data does not need to be the one used to write data
    - Allows adding fields

Parquet

- Apache Parquet
    - Columnar storage format for nested data
        - Columnar:
            - Values for one attribute are stored next to another
            - Can use compression
    - Parquet can store ***nested structures*** in a columnar way

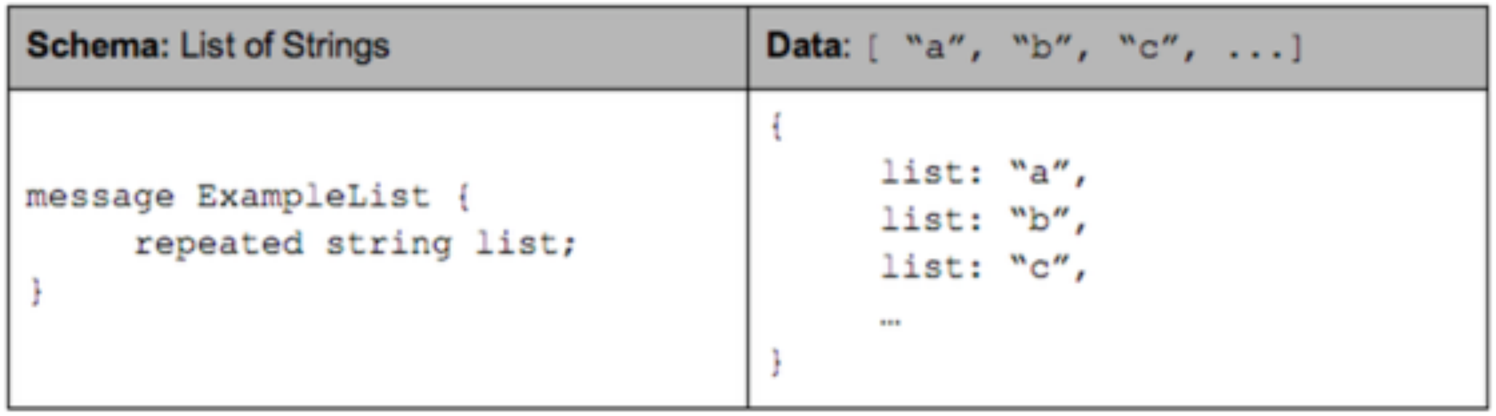
Parquet

- Data is described by a schema
  - Simple schema:

    -
      ```
      message WeatherRecord {
        required int32 year;
        required int32 temperature;
        required binary stationId (UTF8);
      }
      ```

Parquet

- Nesting is created with the *group* types

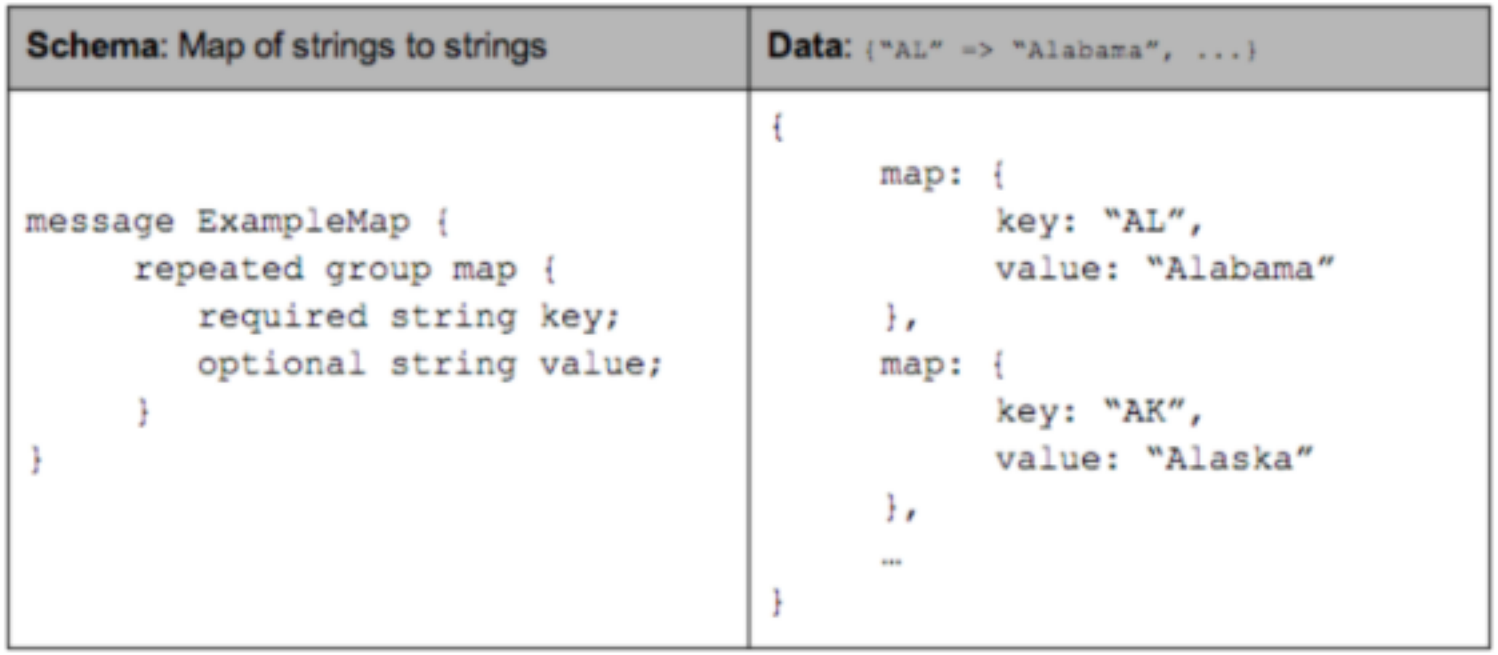


Schema: List of Strings

```
message ExampleList {
     repeated string list;
}
```

Data: `[ "a", "b", "c", ...]`

```
{
     list: "a",
     list: "b",
     list: "c",
     …
}
```



Schema: Map of strings to strings

```
message ExampleMap {
     repeated group map {
        required string key;
        optional string value;
     }
}
```

Data: `{"AL" => "Alabama", ...}`

```
{
     map: {
          key: "AL",
          value: "Alabama"
     },
     map: {
          key: "AK",
          value: "Alaska"
     },
     …
}
```

Hadoop Operations

- Hadoop is flexible with cluster size
  - But need to describe network topology
  - Configuration is not global:
    - Each Hadoop node has its own configuration files
      - Can run old and new machines in parallel
  - Primary, secondary namenode and yarn resource manager need a lot of hardware support

Hadoop Operations

- Security:
  - Uses Kerberos

Hadoop 3

- Use of erasure coding instead of replication for infrequently read files
- Higher availability for NameNode
- Yarn TimeLine server:
  - Retrieve information on current and past application as well as the framework
  - No longer limited by using a single disk
  - Can now be replicated

Erasure Coding

- n-out-of-m encoding
  - Data stored in m chunks
  - All data is available as long as n chunks can be read

Erasure Coding

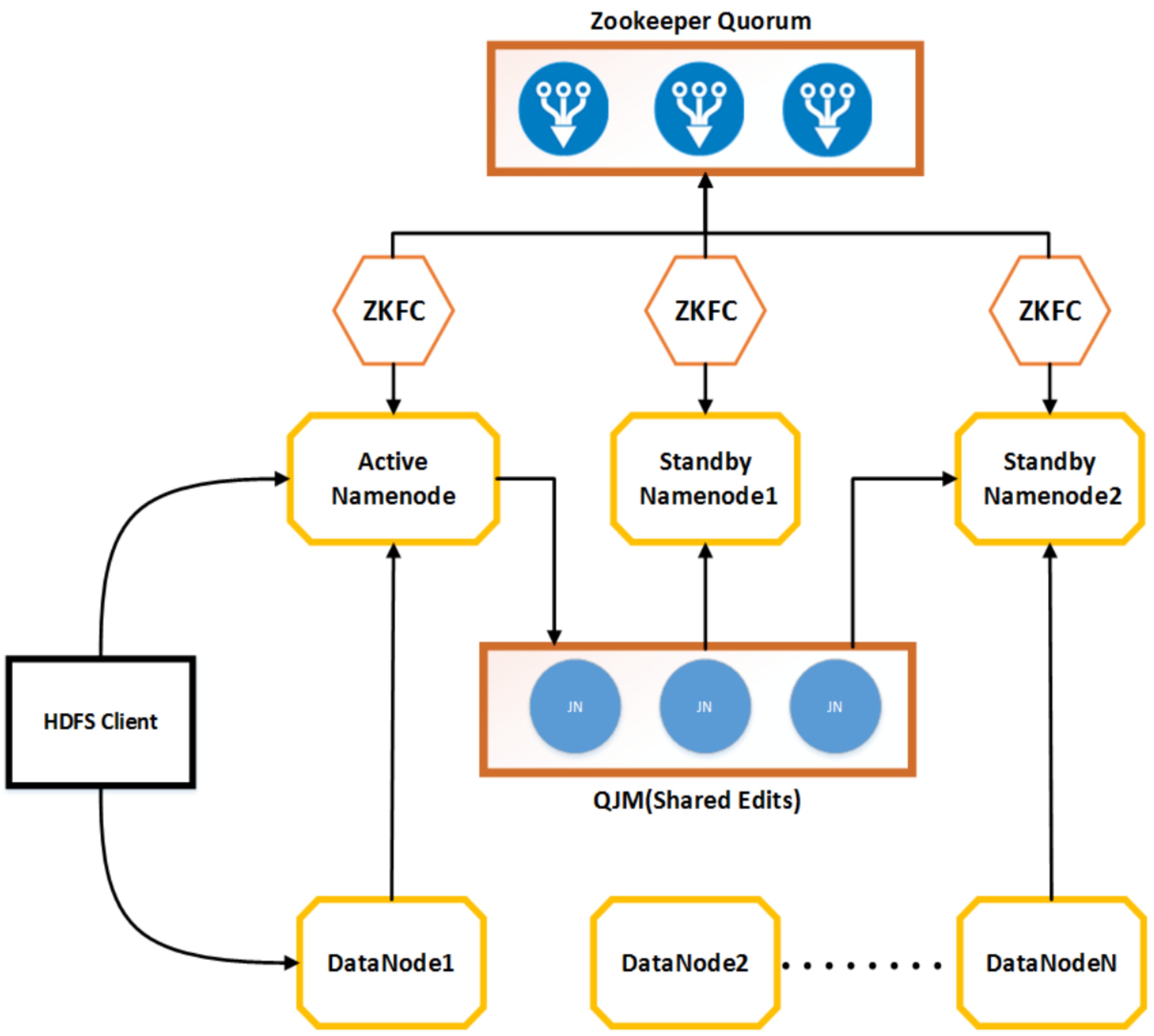
- Example: Generalized linear codes
  - Data is seen as a sequence of symbols in $\mathcal{F}(2^n)$, the finite field with 2^n elements
  - Store data in k arrays $D_i =^t (d_{i,0}, d_{i,1}, d_{i,2}, \dots)$
  - Calculate $n-k$ parity arrays via

$$(D_1, D_2, \dots, D_k, D_{k+1}, \dots, D_n) = \begin{pmatrix} 1 & 0 & \dots & 0 & p_{11} & \dots & p_{1,n-k} & \\ 0 & 1 & \dots & 0 & p_{21} & \dots & p_{2,n-k} & \\ \vdots & \vdots & \ddots & \vdots & & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & p_{n1} & \dots & p_{n,n-k} & \end{pmatrix}^t \cdot (D_1, D_2, \dots, D_k)$$

Erasure Coding

- If some data arrays are lost:
  - Calculate an inverse from the “generator matrix” after column elimination and use matrix multiplication

Hadoop 3



HDFS overview in Hadoop 3

Hadoop 3

Hadoop 3

Hadoop 3

Hadoop 3