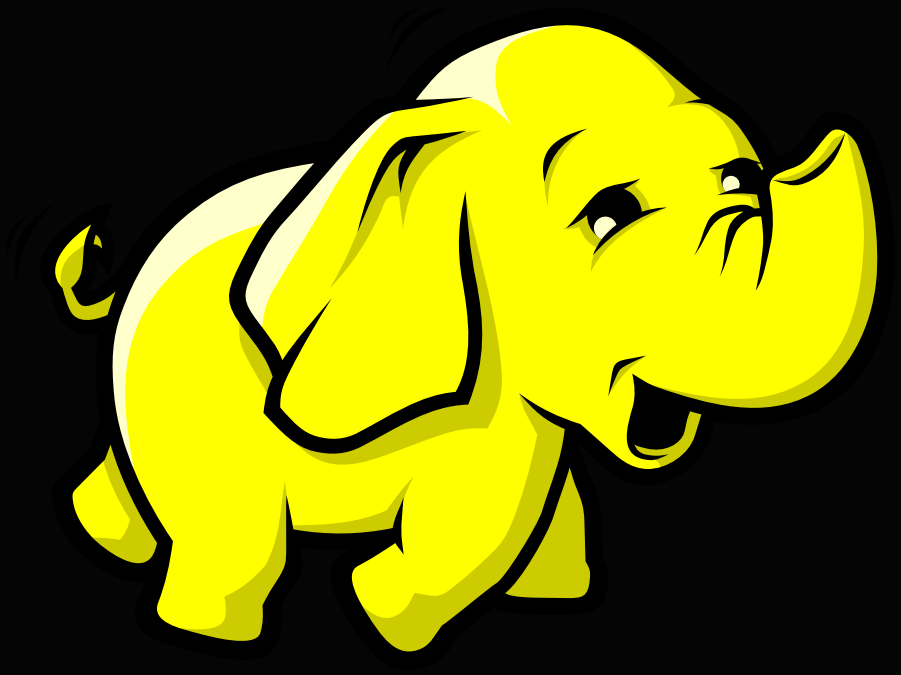
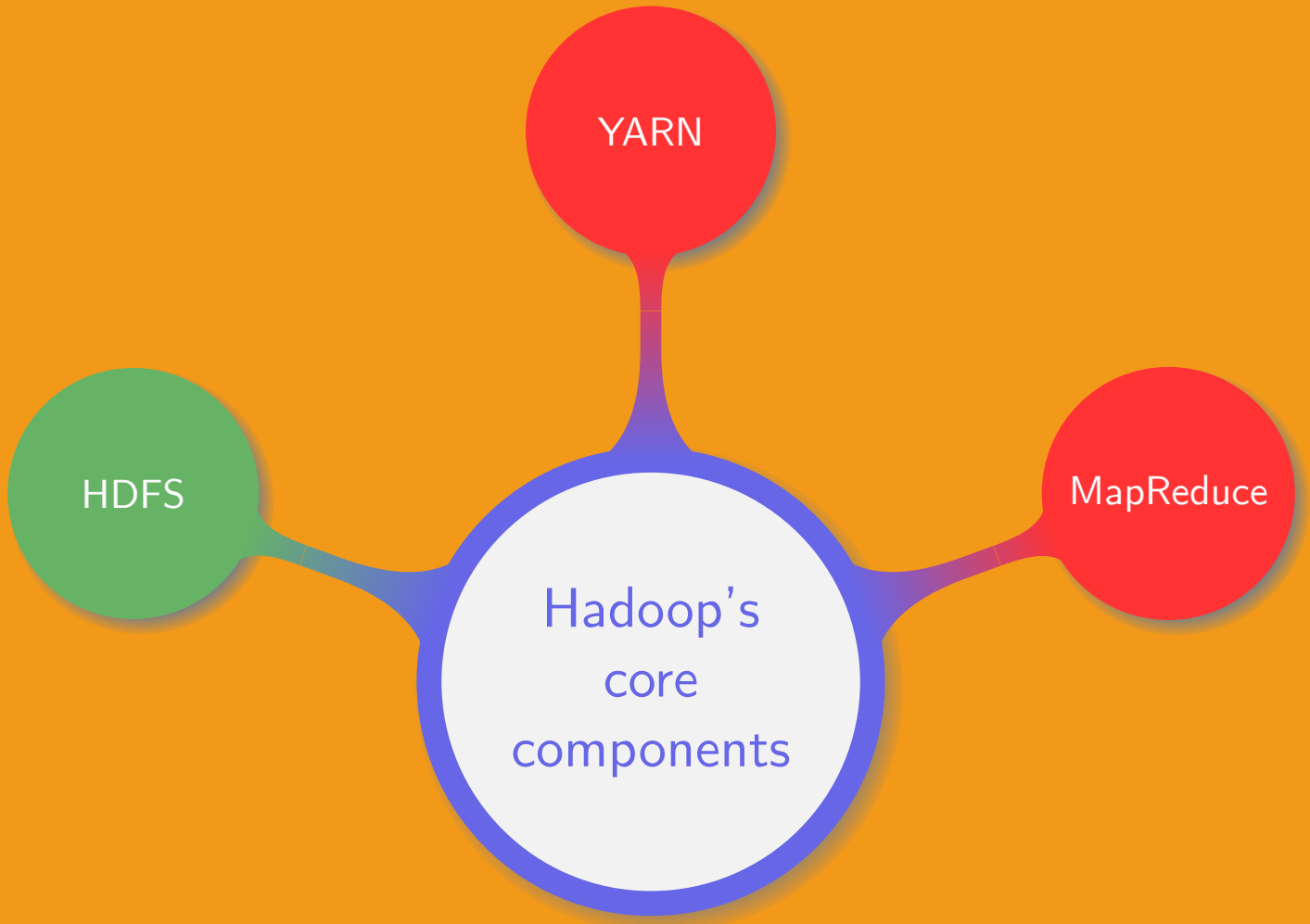


Methods and tools for big data

10. Hadoop's core components

Jing & Manuel – Summer 2020

Chapter organisation



Distributed filesystems

Regular filesystems on a computer:

- Partition
- Hard drive
- LVM

Distributed filesystems

Regular filesystems on a computer:

- Partition
- Hard drive
- LVM

Distributed filesystems:

- Spans several computers
- Has to deal with potential network issues

Distributed filesystems

Regular filesystems on a computer:

- Partition
- Hard drive
- LVM logical volume manager

Distributed filesystems:

- Spans several computers
- Has to deal with potential network issues

Idea behind HDFS summarised on slide 9.17

File blocks

reduce seek time

Blocks in HDFS:

- Default size of 128 MB

very large. usually 4kB

- Files smaller than a block size do not occupy the whole block
- A file can be larger than a whole disk
- Data and metadata handled separately
- Easy to implement fault tolerance and availability

network issue, crash, etc.

backup
by defult: 3 copies in total

Nodes

Two types of node in a cluster:

- Namenode:
  - Maintains FS tree and metadata for all files and directories
  - Locally stores information in namespace image and edit log
  - Knows on which datanodes the blocks of a file are located
- Datanode:
  - Stores and retrieve blocks
  - Regularly reports the list of stored blocks to namenode
  - Can store certain blocks in cache

each task -> correspongding log file

avoid race condition

problem: name node 的blocks of file丢失，就无法找到data

Finding a file

A namenode has no persistent copy of where blocks are:

- Each datanode announces the blocks it has
- All the information is kept in memory by the namenode
- When a write occurs an entry is added to the edit log

Finding a file

A namenode has no persistent copy of where blocks are:

- Each datanode announces the blocks it has
- All the information is kept in memory by the namenode
- When a write occurs an entry is added to the edit log

What to do if the namenode fails?

Memory limitations

A namenode stores all the blocks of all the files in its memory:

- Assume 1 GB of memory for 1 million blocks
- 200 nodes cluster, 24 TB each: ∼12 GB of memory
- Cluster at Yahoo!: 25 PB → ∼64 GB of memory
- Cluster at Facebook: 60 PB → ∼156 GB of memory

Huge memory

Memory limitations

A namenode stores all the blocks of all the files in its memory:

- Assume 1 GB of memory for 1 million blocks
- 200 nodes cluster, 24 TB each: $\sim$12 GB of memory
- Cluster at Yahoo!: 25 PB $\rightarrow$ $\sim$64 GB of memory
- Cluster at Facebook: 60 PB $\rightarrow$ $\sim$156 GB of memory

How about having more namenodes?

HDFS federation

rsync -> backup

Allowing more namenodes:

- Split the filesystem over several independent namenodes
- Each namenode has a namespace
- Each namespace has its own pool of blocks
- A namespace with a block pool is called namespace volume
- A datanode is not attached to a specific namespace volume

一个datanode可以对应很多namenode（存不同namenode的block）
如果有namenode fail不至于整个datanode都没用了（维持computation power)

High availability

Active在运作，passive是backup

Two namenodes in an active-passive mode:

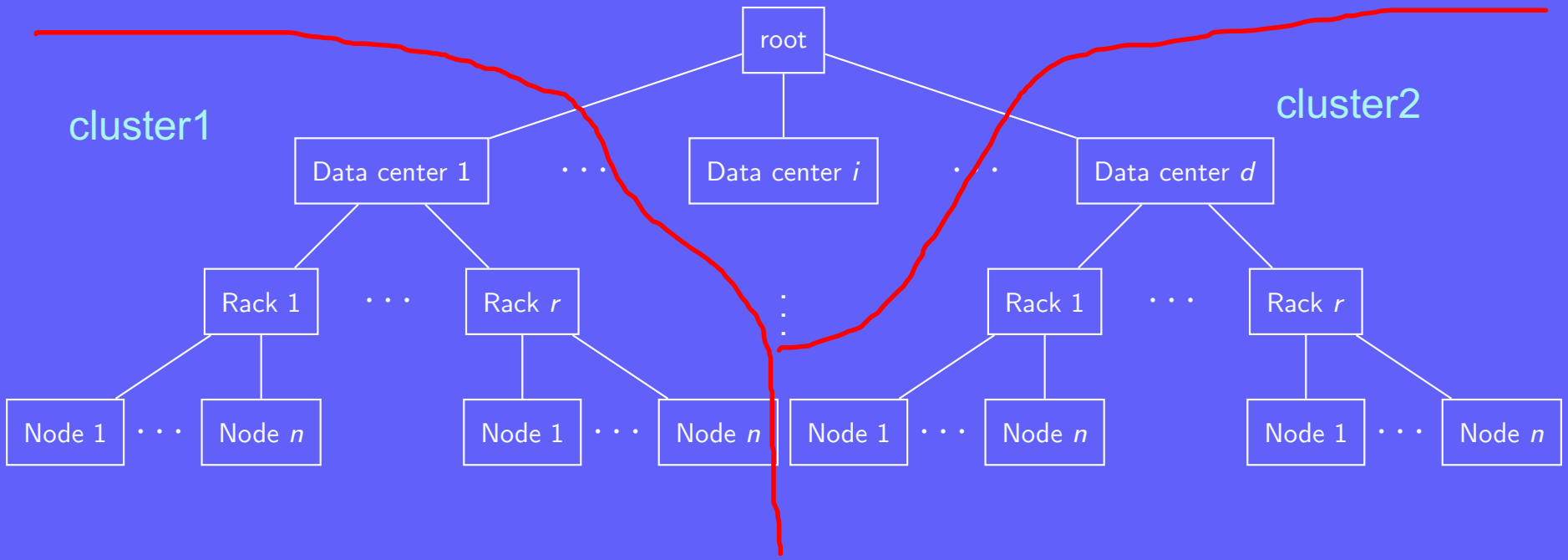
- Passive node takes over in case of failure of the active one
- The two namenodes share the same edit log
- Only the active namenode can write to the edit log
- Passive namenode reads entries when written in edit log
- Datanodes send block reports to both namenodes
- Passive namenode also works as secondary namenode
- Clients must be configured to handle namenode failures

trouble: 如果active node down，用了passive node，然后active突然好了 -> race condition

解决方式STONITH (kill the other one)

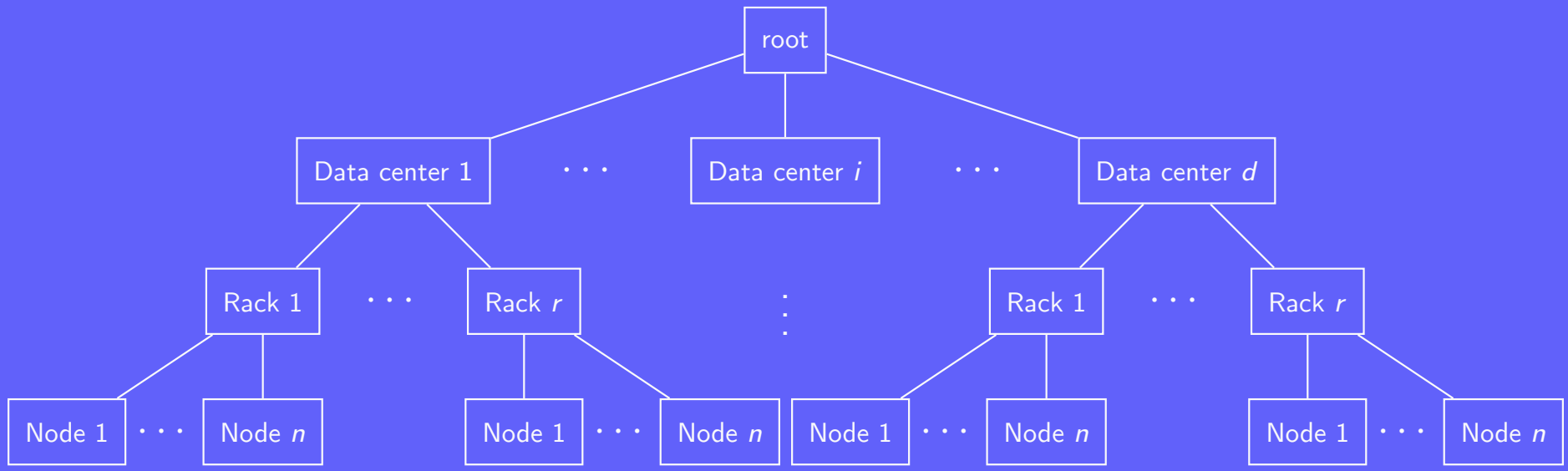
Cluster balancing

The goal is to evenly spread blocks across the whole cluster:



Cluster balancing

The goal is to evenly spread blocks across the whole cluster:



Replicas' location:

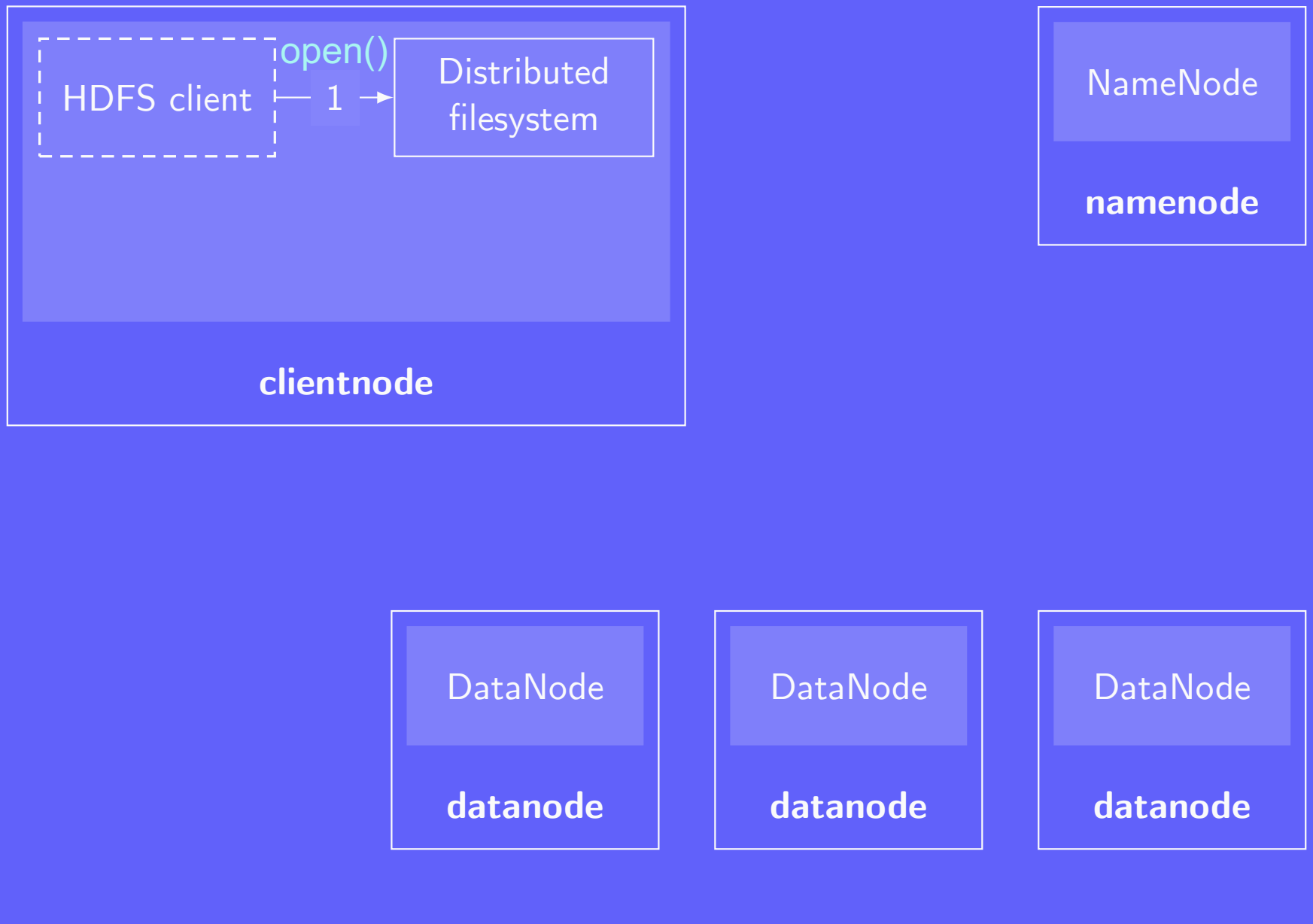
- First: same node as the client
- Second: random, different rack from the first
- Third: same rack as the second but different node
- Others: random nodes in the cluster

不用同一个rack：防止一个rack全crash了

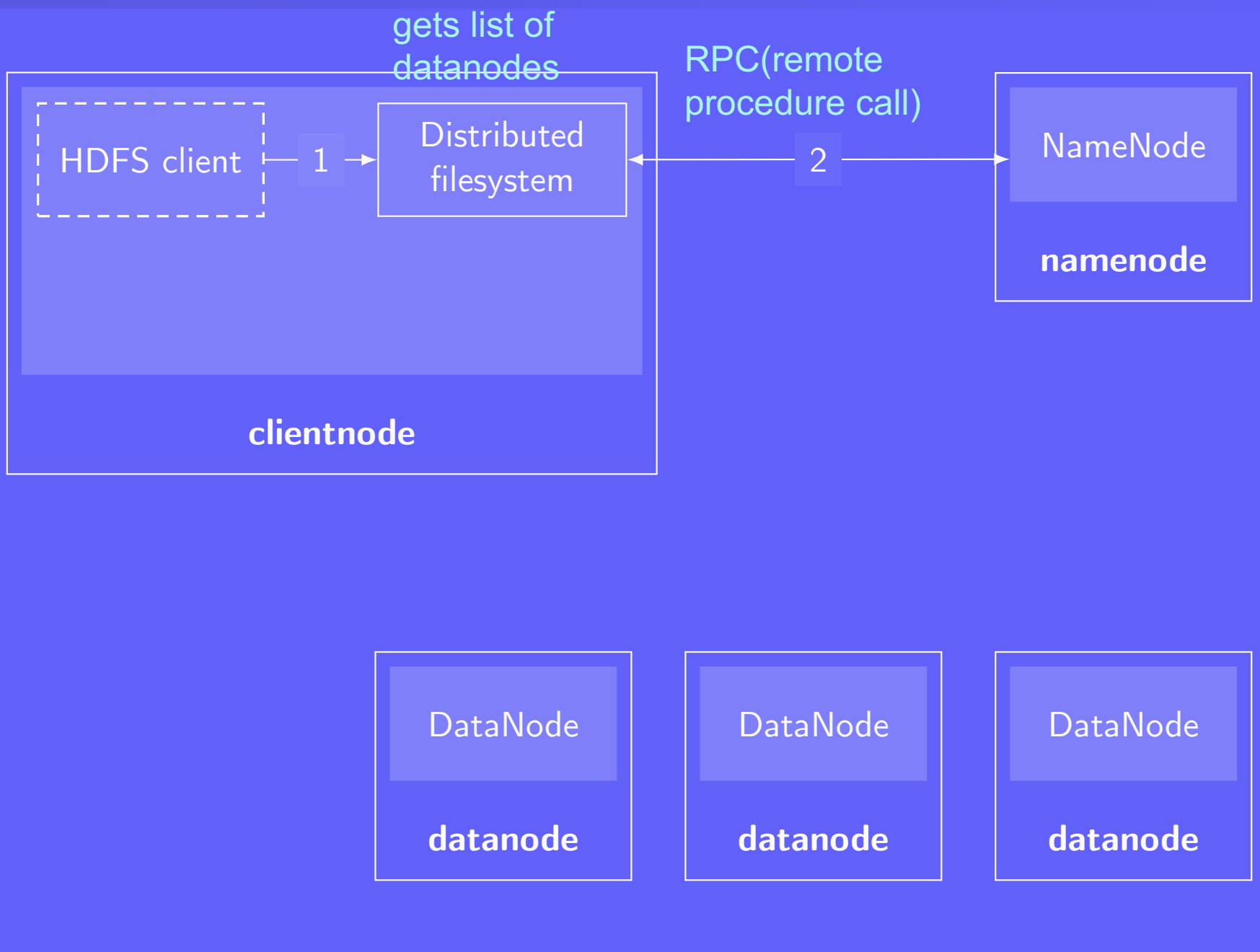
转换到同一个rack的快

avoid using almost full nodes

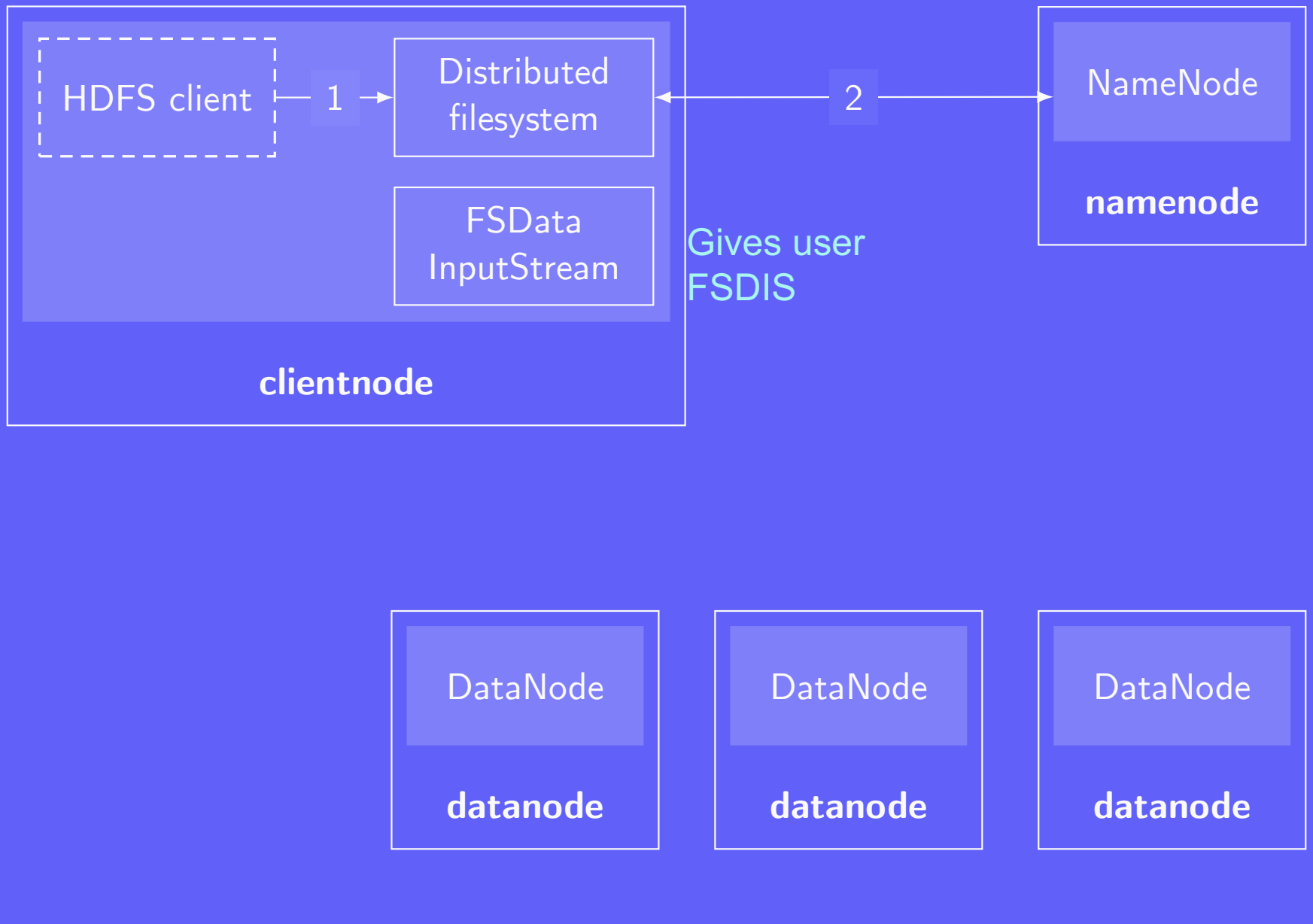
File read


HDFS client
open()
1
Distributed filesystem
clientnode
NameNode
namenode
DataNode
datanode
DataNode
datanode
DataNode
datanode

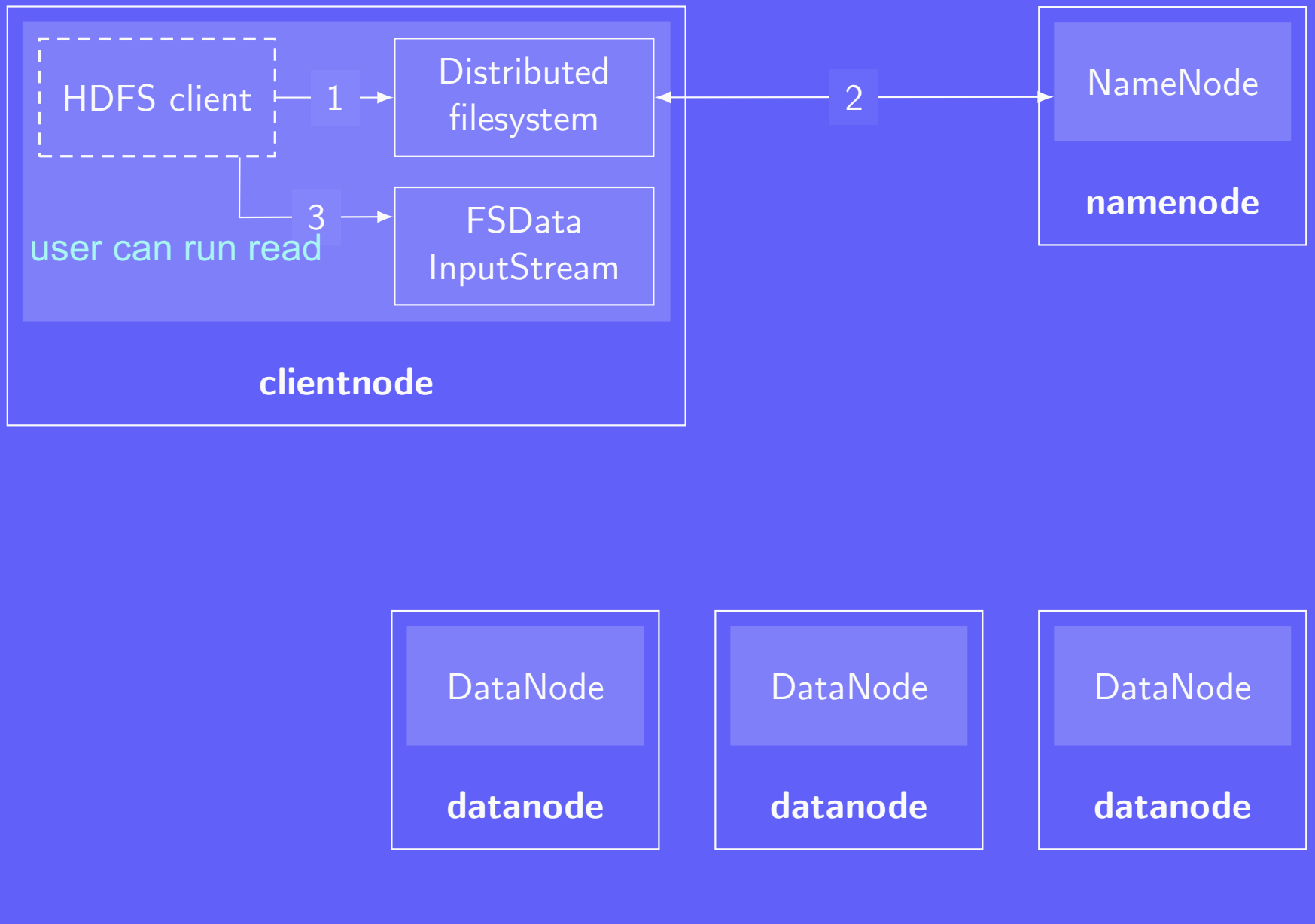
File read


gets list of datanodes
RPC(remote procedure call)
HDFS client
1
Distributed filesystem
2
NameNode
namenode
clientnode
DataNode
datanode
DataNode
datanode
DataNode
datanode

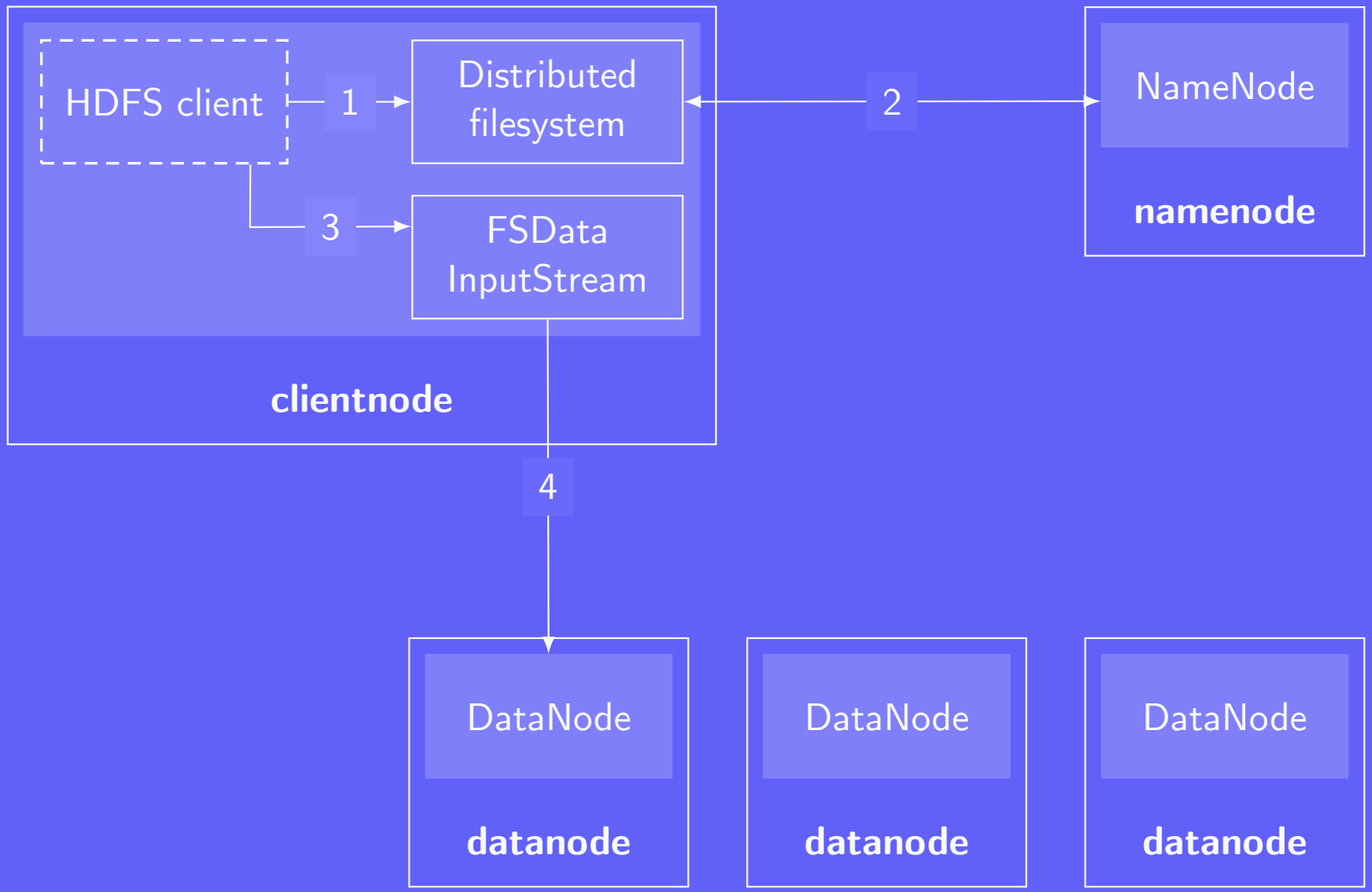
File read


HDFS client
1
Distributed filesystem
FSData InputStream
clientnode
2
Gives user FSDIS
NameNode
namenode
DataNode
datanode
DataNode
datanode
DataNode
datanode

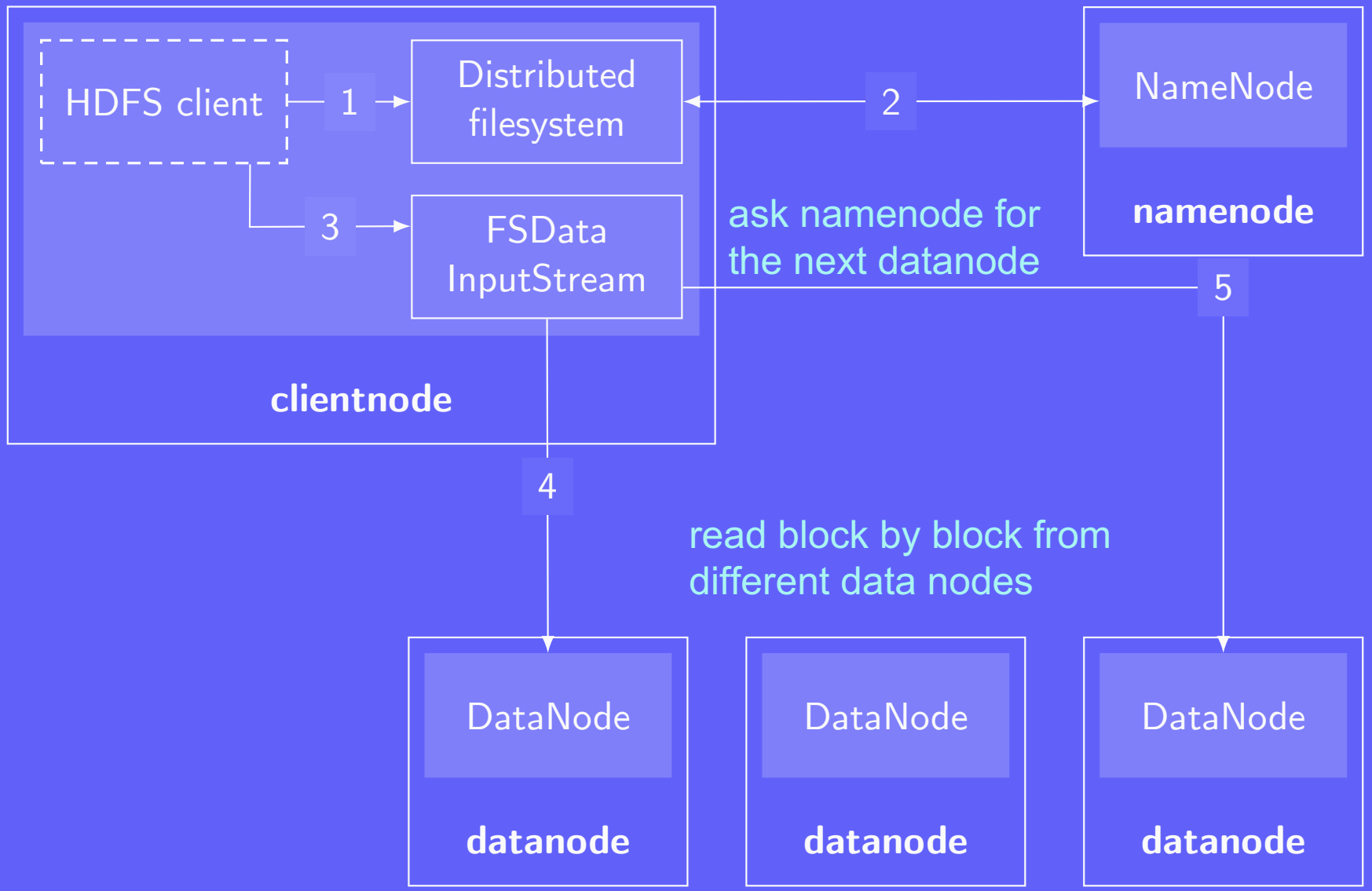
File read


HDFS client
1
Distributed filesystem
2
NameNode
namenode
3
user can run read
FSData InputStream
clientnode
DataNode
datanode
DataNode
datanode
DataNode
datanode

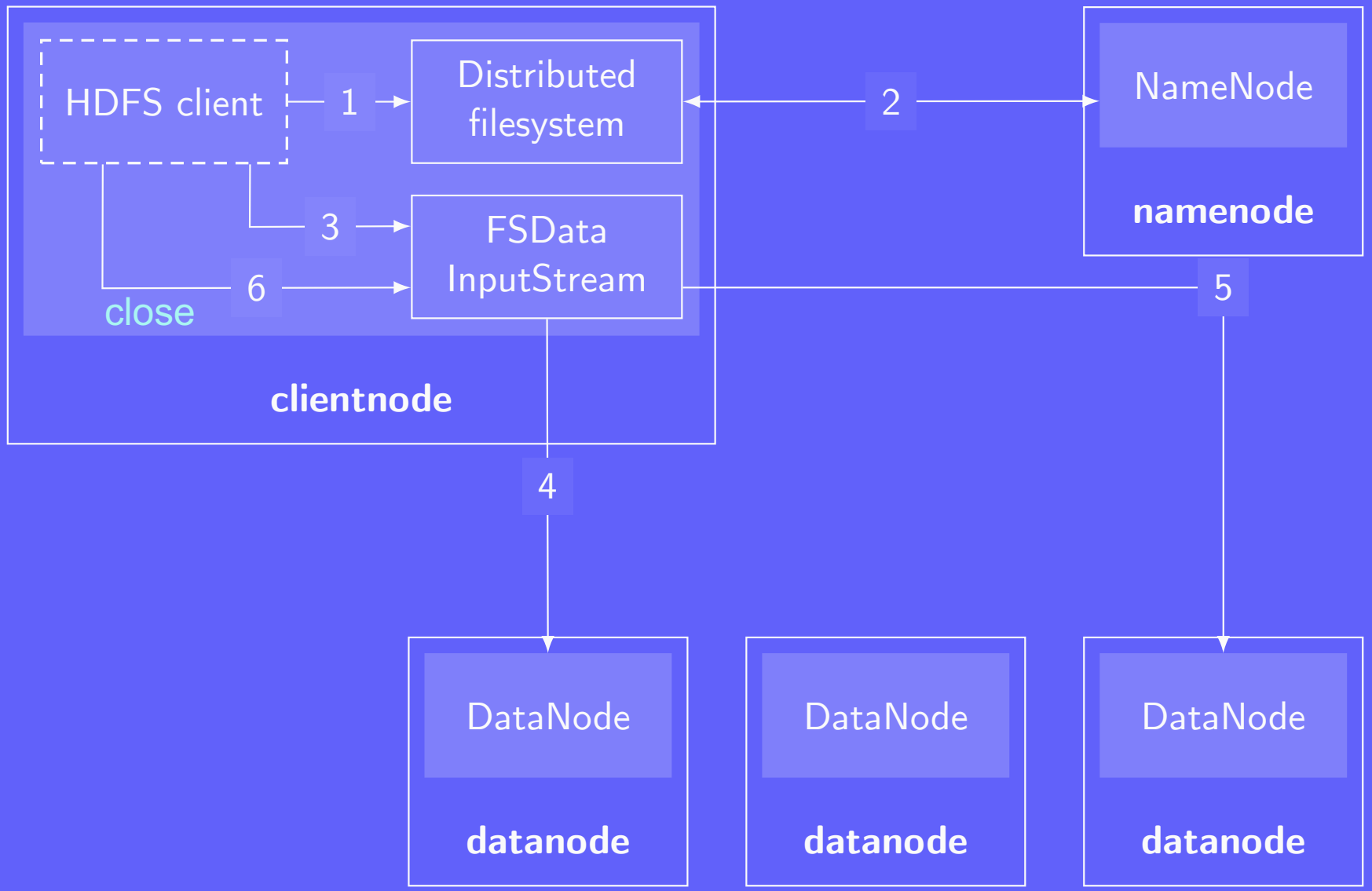
File read


HDFS client
1
Distributed filesystem
2
NameNode
namenode
3
FSData InputStream
clientnode
4
DataNode
datanode
DataNode
datanode
DataNode
datanode

File read

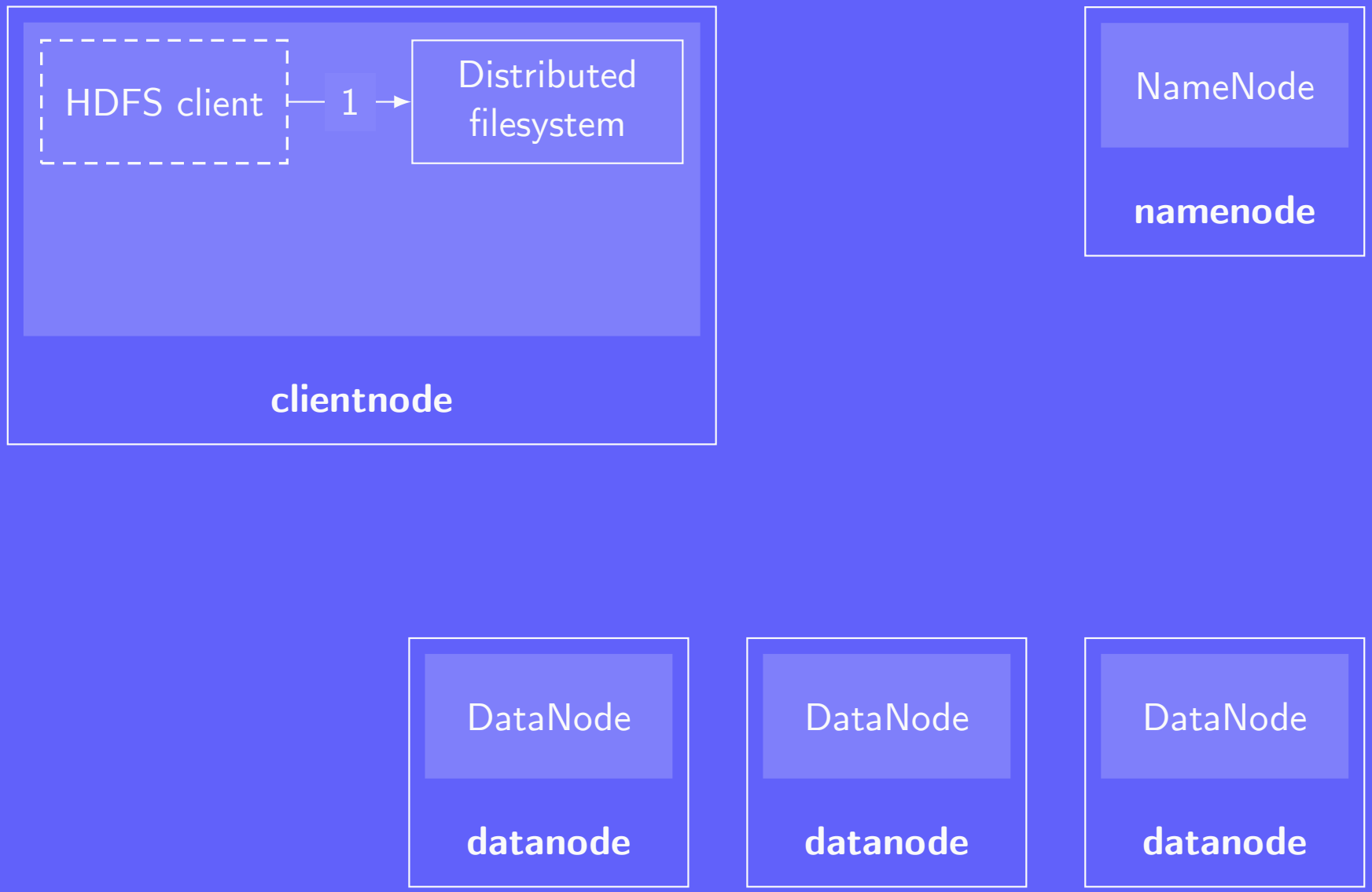

HDFS client
1
Distributed filesystem
2
NameNode
namenode
3
FSData InputStream
ask namenode for the next datanode
5
clientnode
4
read block by block from different data nodes
DataNode
datanode
DataNode
datanode
DataNode
datanode

File read

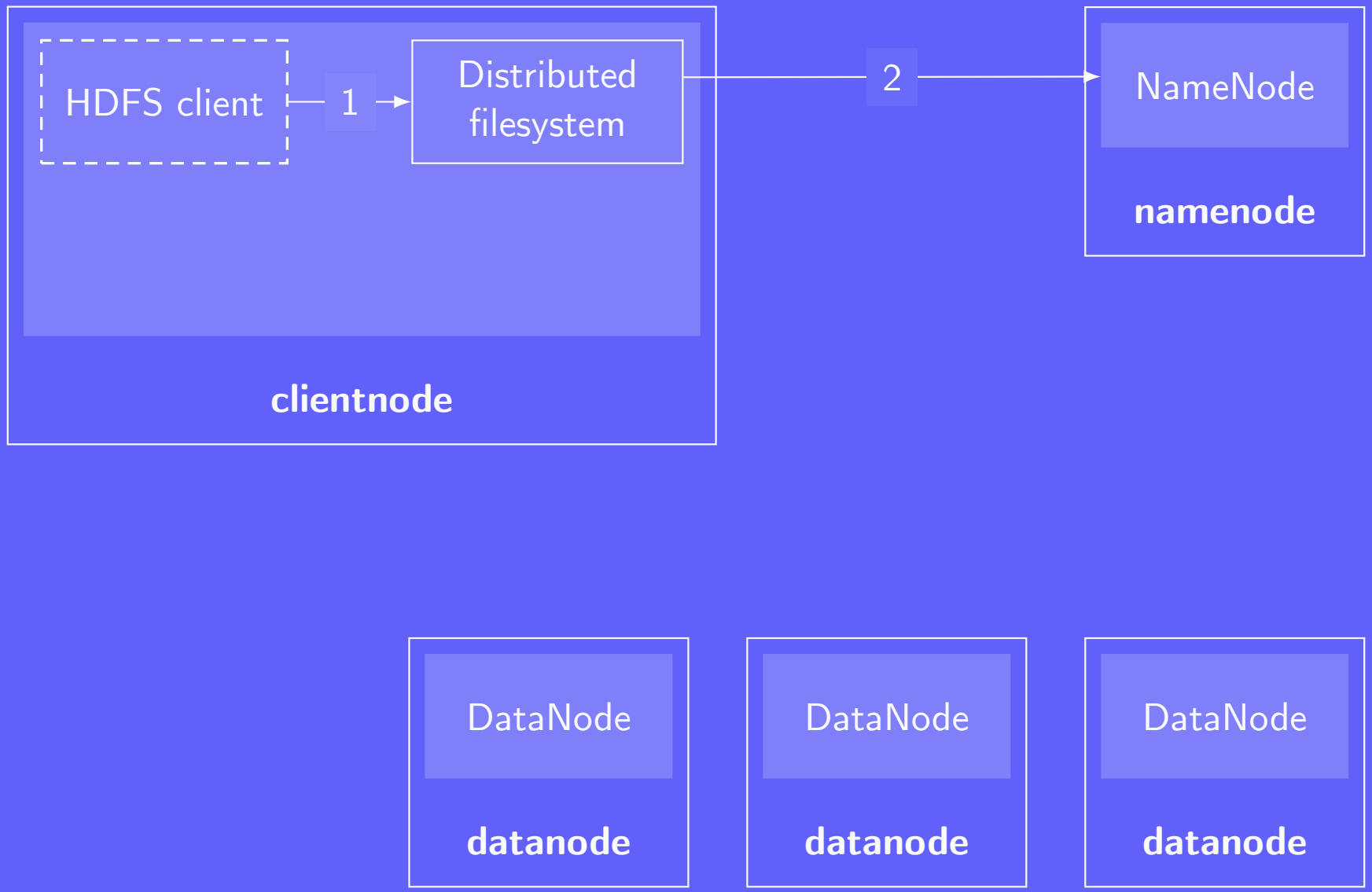

HDFS client
1
Distributed filesystem
2
NameNode
namenode
3
FSData InputStream
6
close
5
clientnode
4
DataNode
datanode
DataNode
datanode
DataNode
datanode

File write

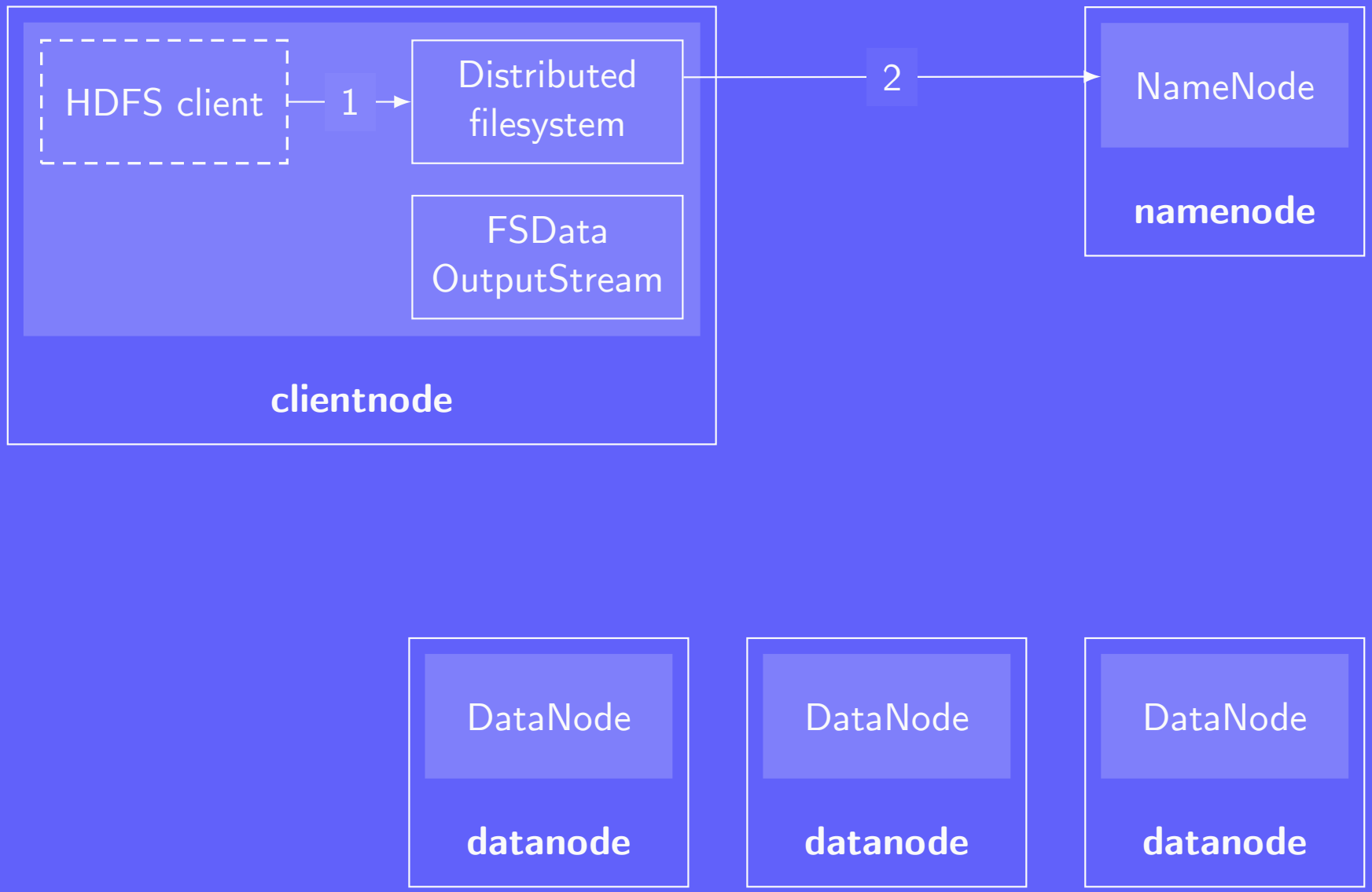
check if exist / permission
create file



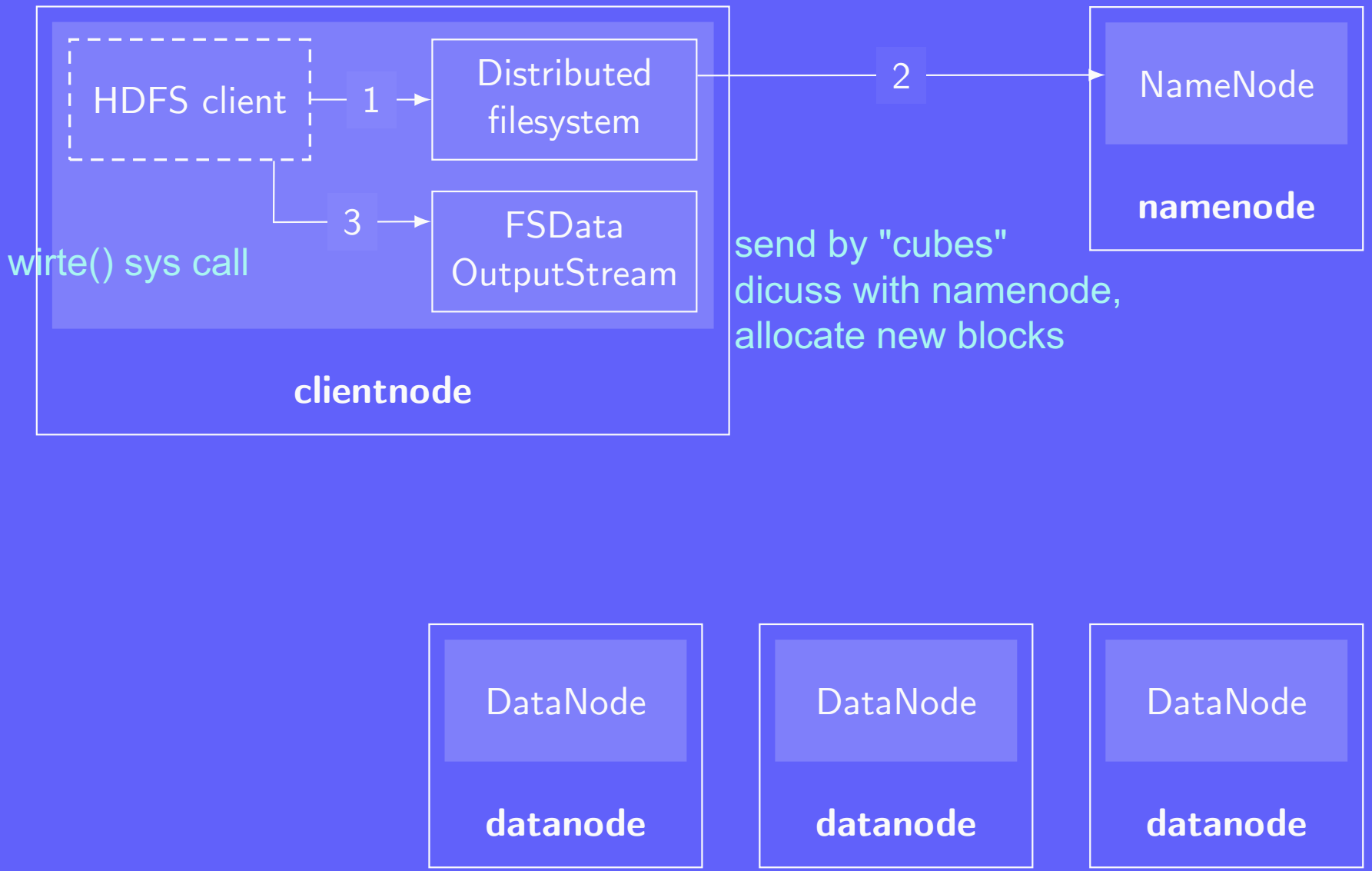
File write


HDFS client
1
Distributed filesystem
2
NameNode
namenode
clientnode
DataNode
datanode
DataNode
datanode
DataNode
datanode

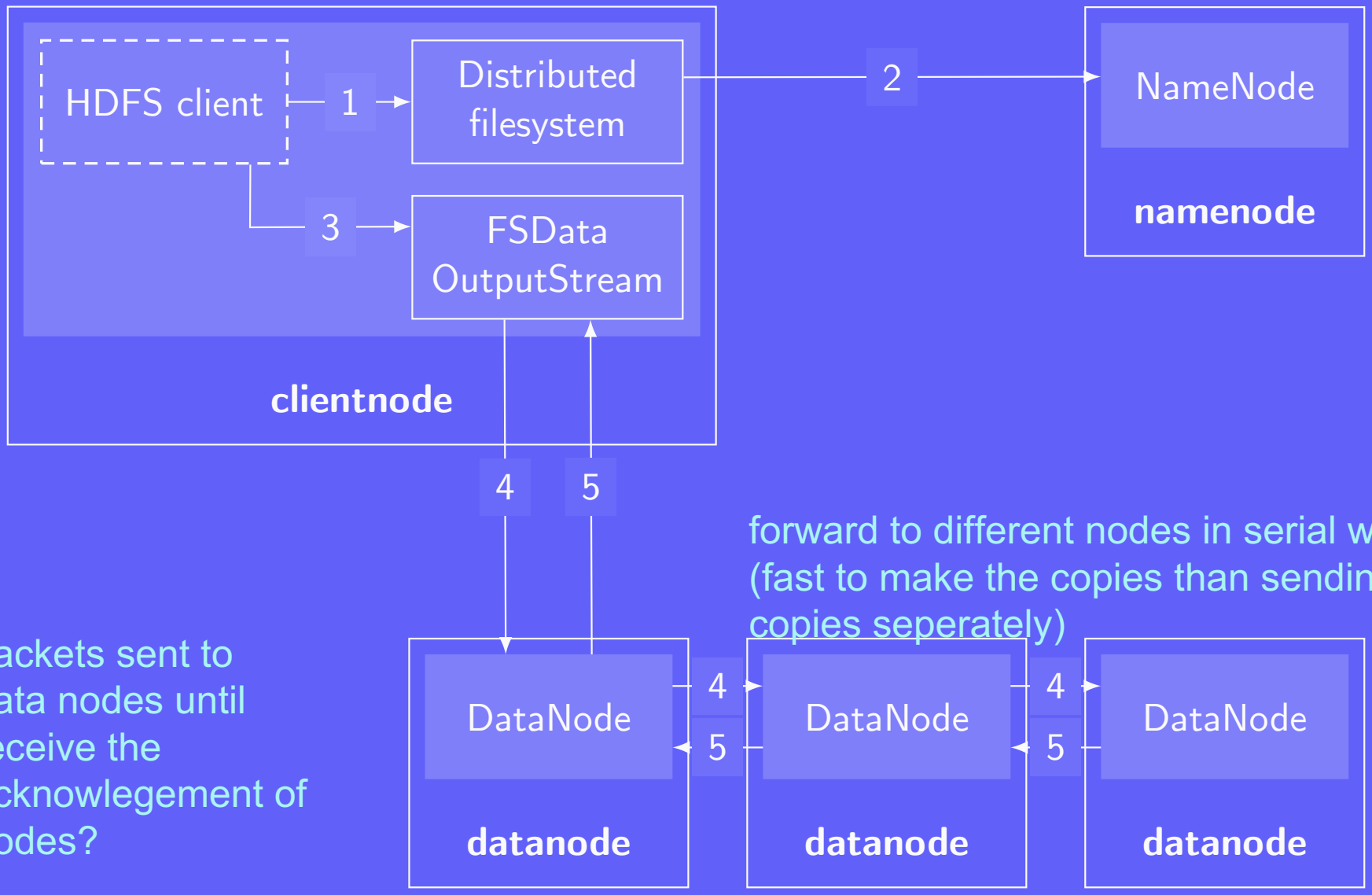
File write


HDFS client
1
Distributed filesystem
FSData OutputStream
clientnode
2
NameNode
namenode
DataNode
datanode
DataNode
datanode
DataNode
datanode

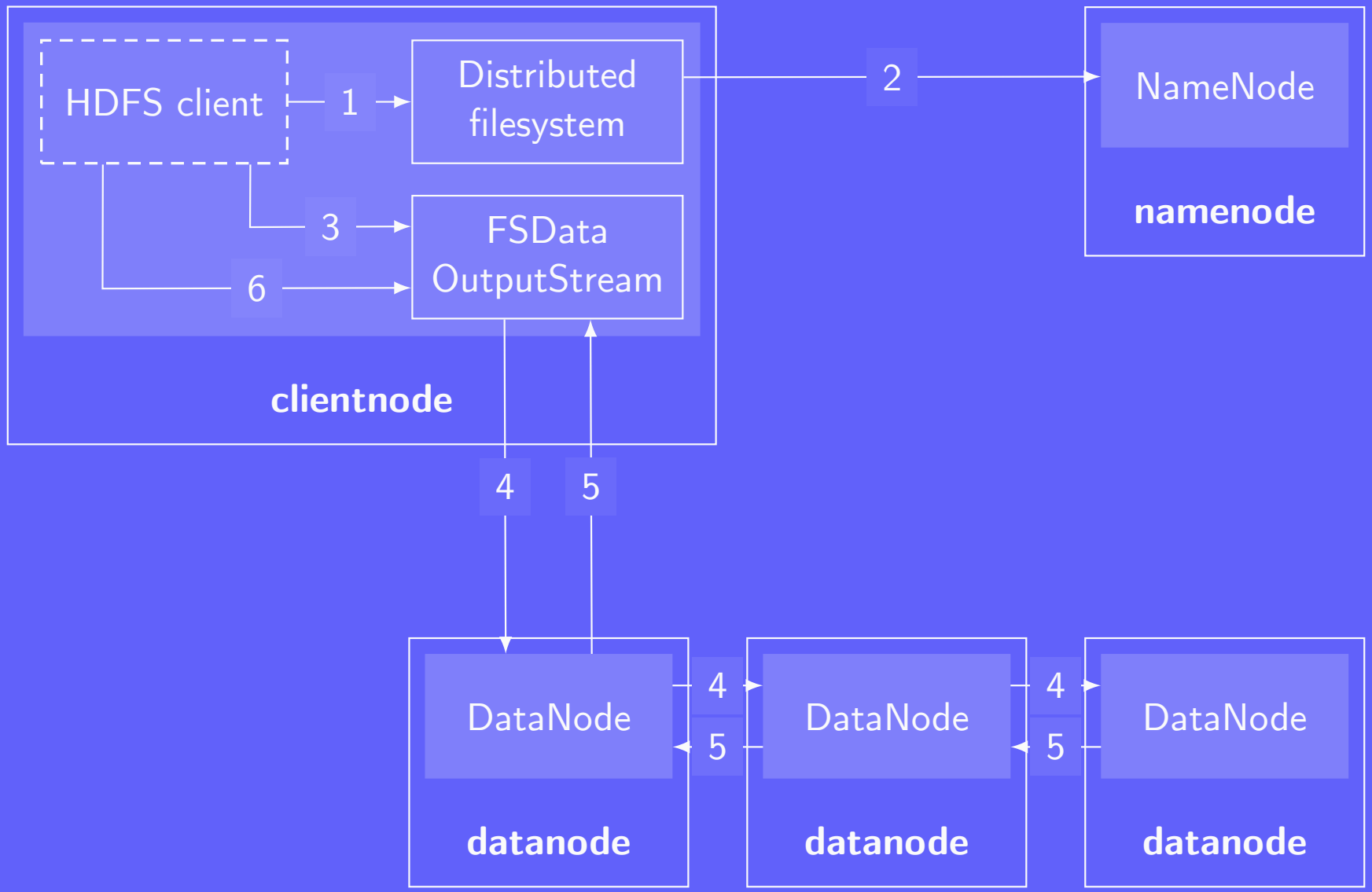
File write


HDFS client
1
Distributed filesystem
2
NameNode
namenode
3
FSData OutputStream
wirte() sys call
send by "cubes"
dicuss with namenode,
allocate new blocks
clientnode
DataNode
datanode
DataNode
datanode
DataNode
datanode

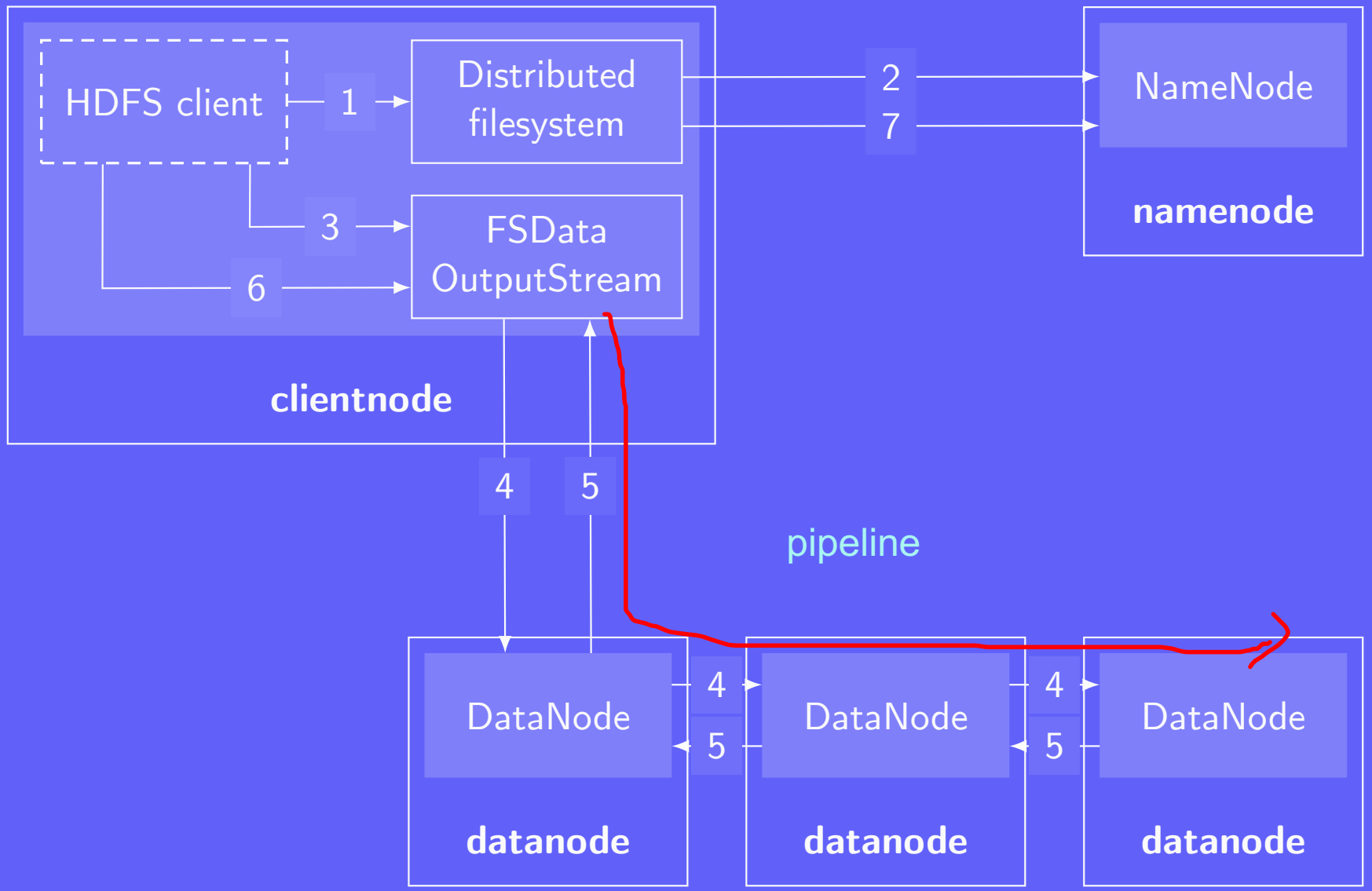
File write


HDFS client
1
Distributed filesystem
2
NameNode
namenode
3
FSData OutputStream
clientnode
4
5
forward to different nodes in serial way (fast to make the copies than sending copies seperately)
packets sent to data nodes until receive the acknowlegement of nodes?
DataNode
4
5
DataNode
4
5
DataNode
datanode
datanode
datanode
send copies to other nodes

File write


HDFS client
1
Distributed filesystem
2
NameNode
namenode
3
FSData OutputStream
6
clientnode
4
5
DataNode
4
5
DataNode
4
5
DataNode
datanode
datanode
datanode

File write


HDFS client
1
Distributed filesystem
2
7
NameNode
namenode
3
FSData OutputStream
6
clientnode
4
5
pipeline
DataNode
4
5
DataNode
4
5
DataNode
datanode
datanode
datanode

Failing file write

等这个node好了，mark blocks as free (block可能有坏之前patially copy的内容，覆盖它

When write on a data node fails:

1. Close the pipeline
2. Add packets in front of the acknowledgment queue
3. Inform the name node of the failing data node
4. Remove the faulty data node from the pipeline
5. Construct a new pipeline using only the healthy data nodes
6. Complete the writing of the block across the pipeline
7. Arrange for the replication of under-replicated blocks

Other filesystems

Alternative filesystems built on top of Hadoop abstract filesystem:

- HAR: pack HDFS files into an archive
- View: used to create mount points for federated FS
- FTP: FS backed by an FTP server
- S3: FS based on Amazon S3
- Azure: FS developed by Microsoft Azure

FUSE

not safe

Other filesystems

Alternative filesystems built on top of Hadoop abstract filesystem:

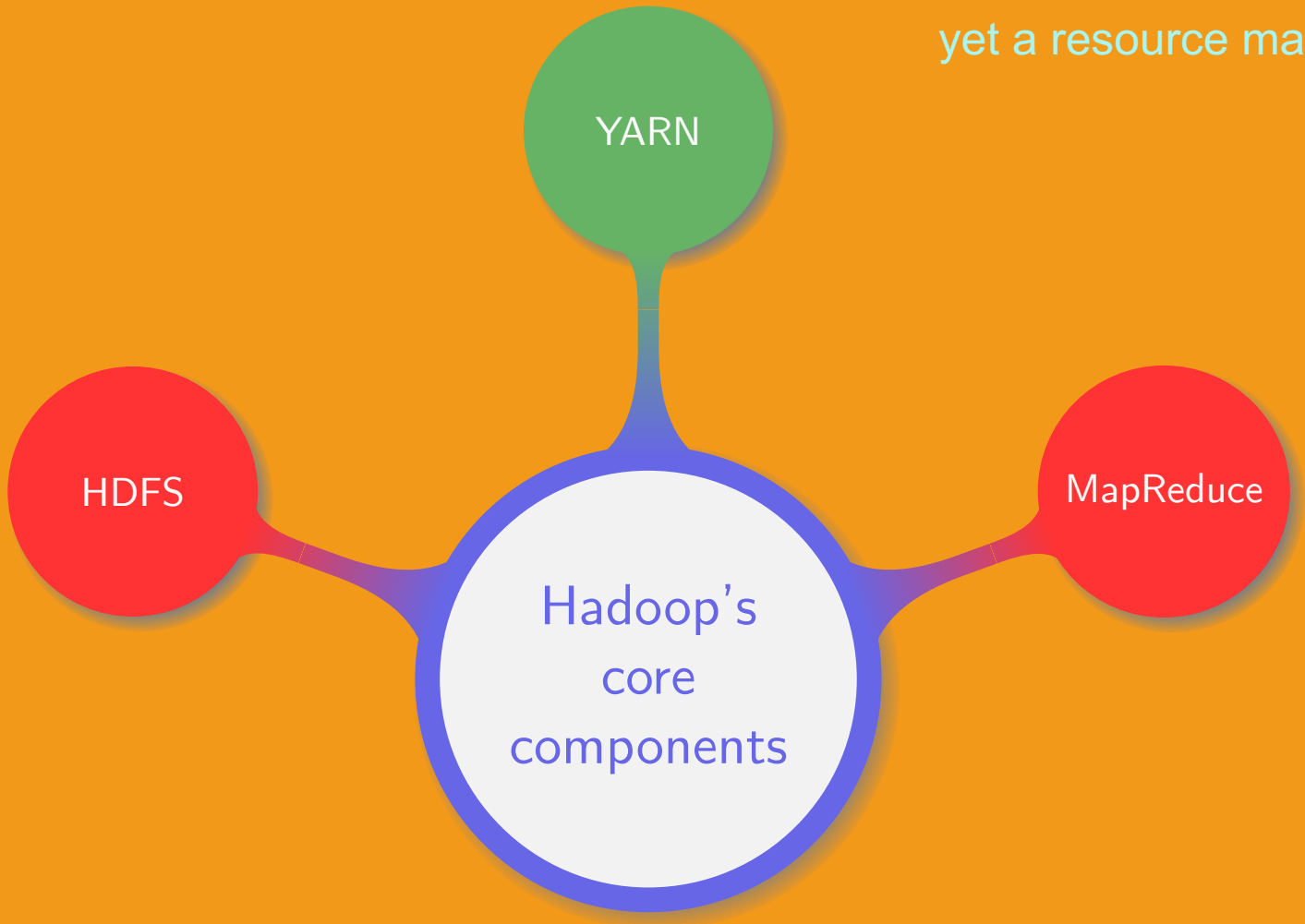
- HAR: pack HDFS files into an archive
- View: used to create mount points for federated FS
- FTP: FS backed by an FTP server
- S3: FS based on Amazon S3
- Azure: FS developed by Microsoft Azure

Interfaces to access HDFS data:

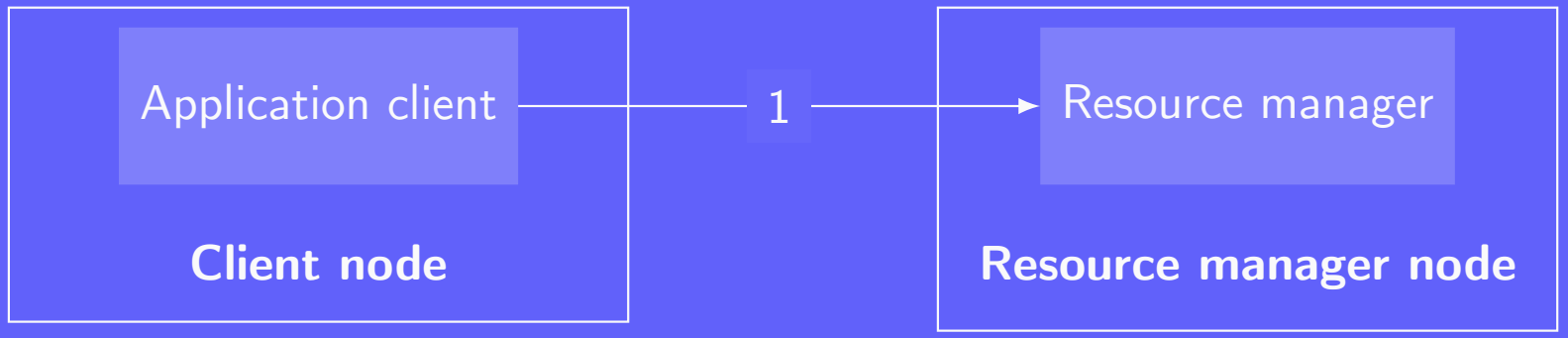
- libhdfs: C library with API similar to the Java one
- NFS: use Hadoop NFS gateway
- FUSE: based on libhdfs

Chapter organisation

yet a resource manager?



Running an application



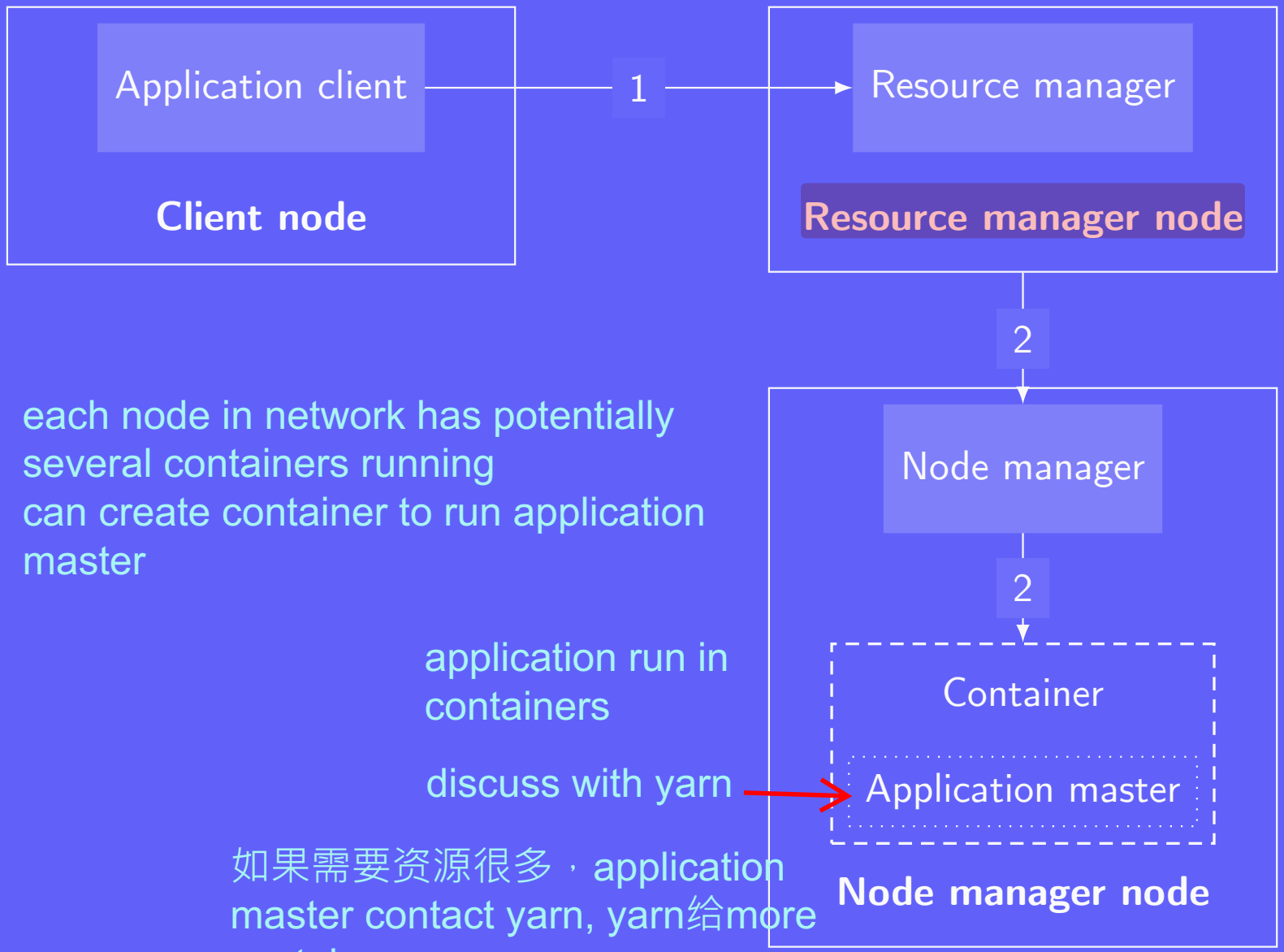
1 datanode where program is initiated

map reduce/ drill/ spark

YARN

HDFS

Running an application



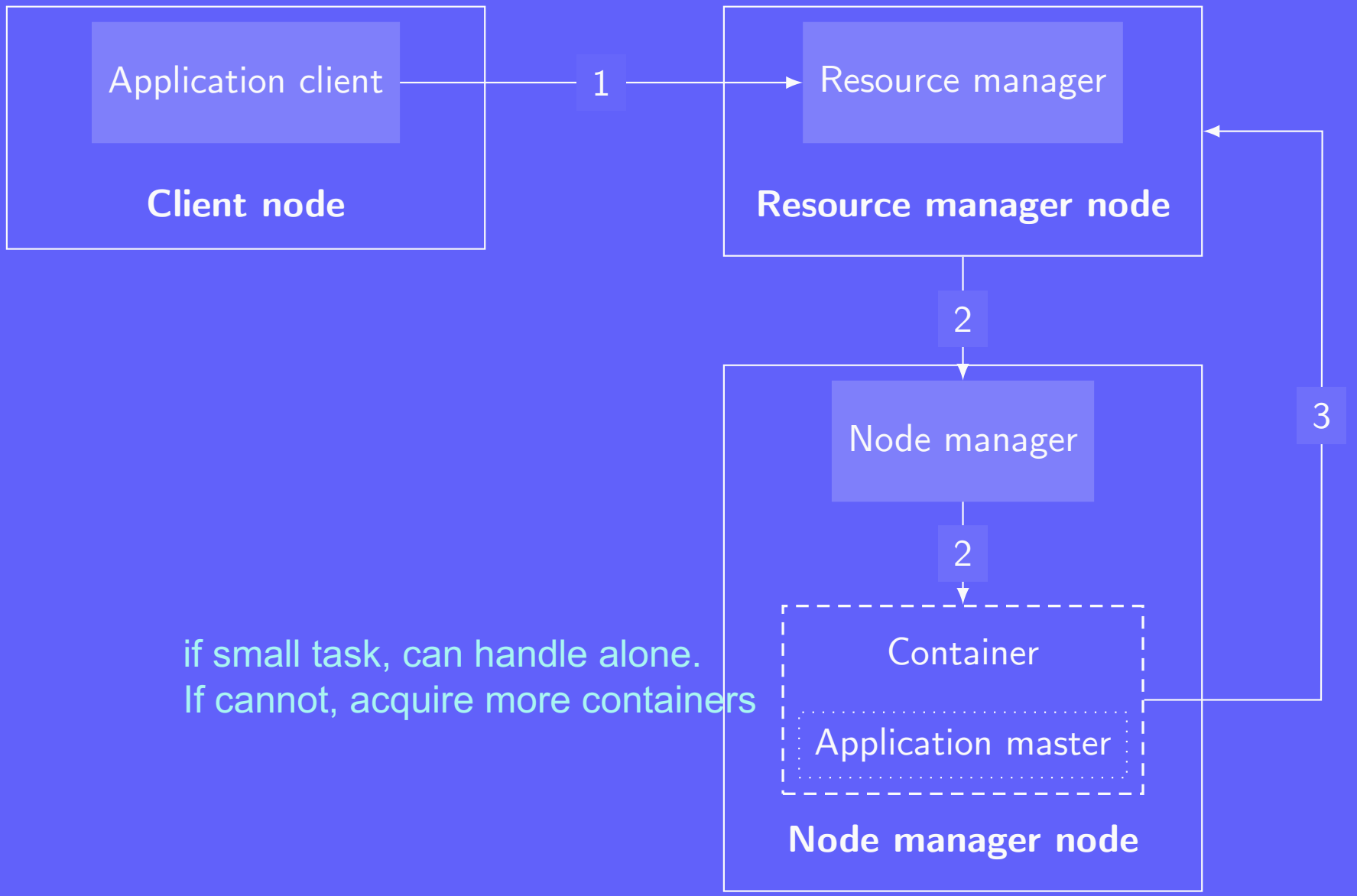
each node in network has potentially several containers running
can create container to run application master

application run in containers

discuss with yarn

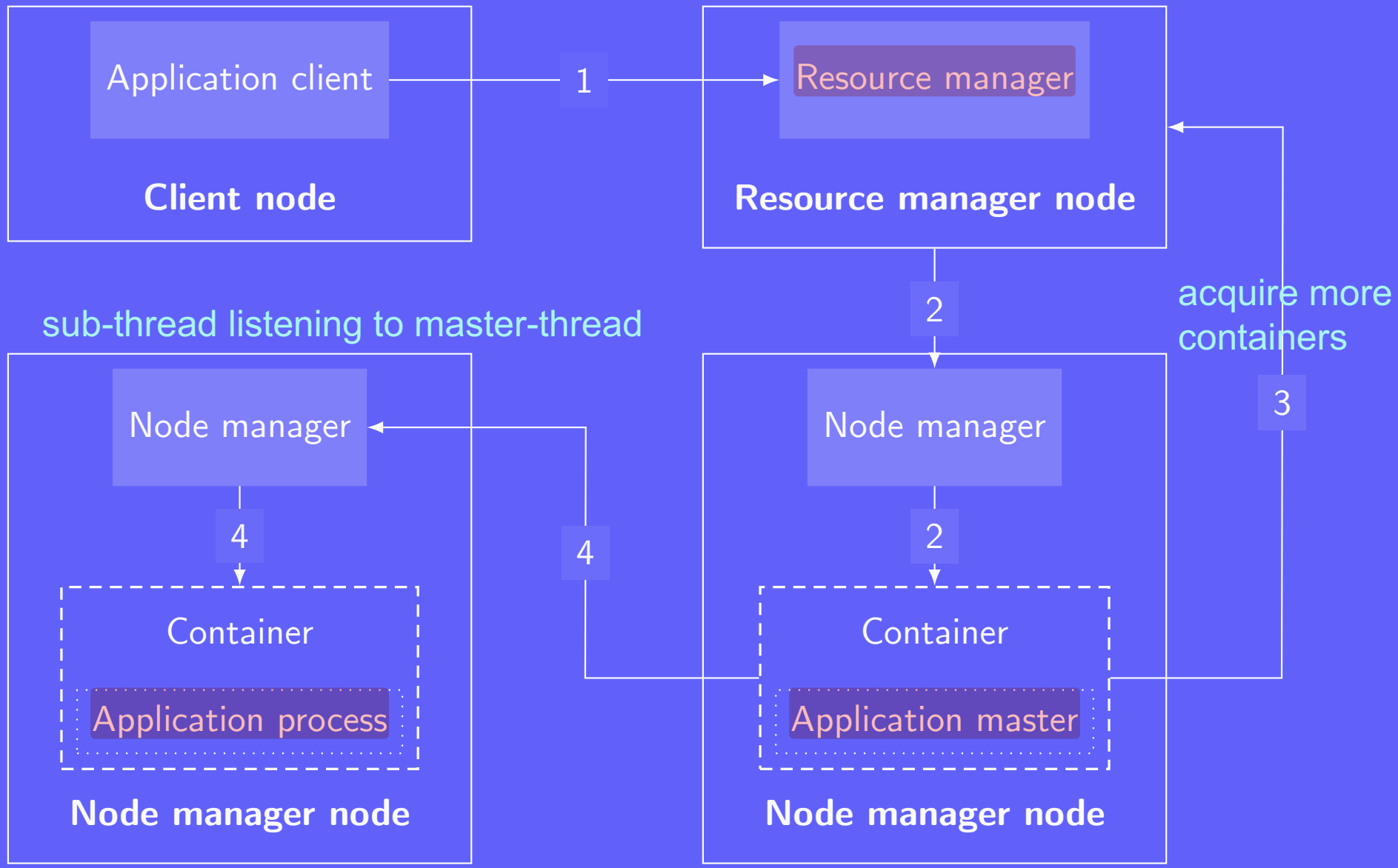
如果需要资源很多，application master contact yarn, yarn给more containers

Running an application



if small task, can handle alone.
If cannot, acquire more containers

Running an application



Requesting resources

A request has two components:

- Amount of resources for each container
- Preferred location of the containers

prefer:
1. other local containers in the same computer
2. same rack
3. same cluster

...

Requesting resources

A request has two components:

- Amount of resources for each container
- Preferred location of the containers

Common strategy for a request:

- Announce all the resources requests at the beginning
- Dynamically request resources based on the needs

Categories of application

YARN can be used in three ways by applications:

- One application per user job: simplest model
  - expensive
  - map reduce
- One application per user session: spark
  - Containers can be reused between jobs
    - don't need to start a container each time
  - Possibility to cache data between jobs
- Long-running application shared among users: drill
  - Application master is always "on"
  - Application master acts as a coordinator for other applications

re-dispatching all the works; may involve multiple users;
may have security issues (different processes may see the resources)

YARN schedulers

Three schedulers available in YARN:

- FIFO: request served one by one in a queue
- Capacity:
  - Define queues based on the "size" of the jobs to complete
  - All the jobs start early
  - Resources are wasted when unused
- Fair:
  - Resources are dynamically balanced over all the jobs
  - All the resources are fully used
  - Delay due the resource reallocation

如果很多job同时出现 -> allocate for all jobs -> may take some recources from a job to another

Capacity scheduler

default schedular

Capacity scheduler setup:

- Each queue is handled as a FIFO
- Queue elasticity
  - A single job cannot exceed the capacity of the queue
  - The capacity can be exceeded when several jobs wait and resources are available
- Containers are not preempted
- Can control:
  - The number of resources per user/application
  - The number of applications that can be run at a time
  - Access Control Lists (ACL) on the queues

Capacity scheduler

Capacity scheduler setup:

- Each queue is handled as a FIFO
- Queue elasticity
  - A single job cannot exceed the capacity of the queue
  - The capacity can be exceeded when several jobs wait and resources are available
- Containers are not preempted
- Can control:
  - The number of resources per user/application
  - The number of applications that can be run at a time
  - Access Control Lists (ACL) on the queues

The challenge is to find a reasonable trade-off for the capacity

Fair scheduler

Fair scheduler queues:

- One or more queues allowed:
  - Single queue: resources are fairly shared among all applications
  - Several queues, each having:
    - Its own scheduling policy
    - A max/min resources and number of applications
- Queues can be precisely configured using an allocation file
- By default a queue is dynamically created for each user

Fair scheduler

Preemption in the fair scheduler:

- Efficiency is reduced as killed containers must be restarted
- Two timeout settings t_1 and t_2 to trigger preemption:
  - Minimum share: if a queue waits longer than t_1
  - Fair share: if a queue remains below half of its fair share for longer than t_2
- Timeouts can be set per queue or globally

Delay scheduling

Locality problem when scheduling:

- An application requests a specific node
- The node is busy
- Should the application wait for the node or loosen its request?

Delay scheduling

Locality problem when scheduling:

- An application requests a specific node
- The node is busy
- Should the application wait for the node or loosen its request?

YARN schedulers' approach:

- Every second each node manager sends a heartbeat reporting the running containers and available resources
- Capacity scheduler: wait for a predefined number of heartbeats before loosening the requirement
- Fair scheduler: wait for a predefined portion of nodes in the cluster to offer opportunities before loosening the requirement

Dominant resource fairness

How to fairly share resources between applications when they do not use the same type of resources?

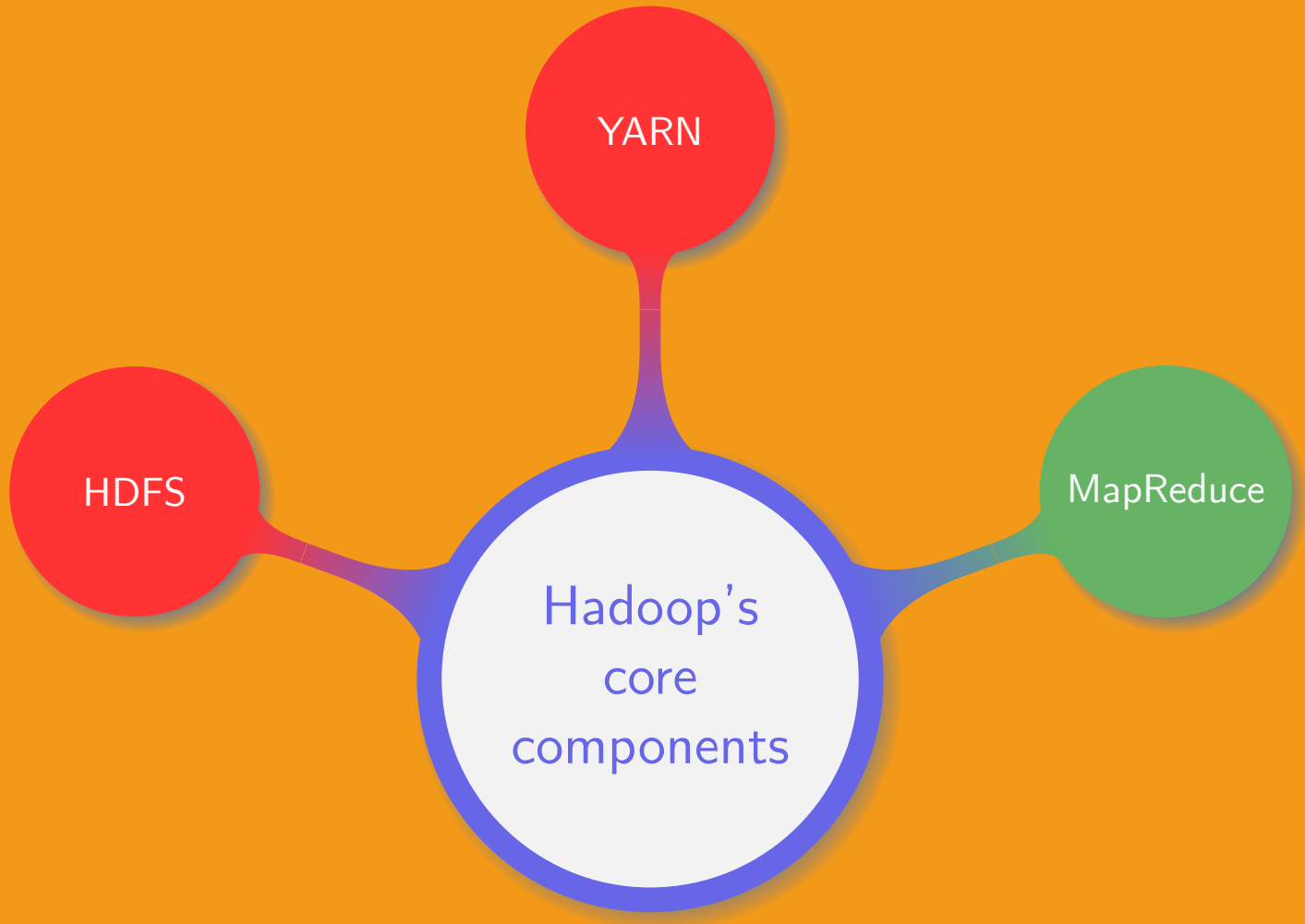
Dominant resource fairness

How to fairly share resources between applications when they do not use the same type of resources?

Basic idea for two applications:

- Consider the proportion of resources requested for a container by each application
- Call the largest proportion the dominant resource and use it as measure of cluster usage
- Proportionally offer less containers to the more demanding application

Chapter organisation



MapReduce job

Parties involved in a MapReduce job:

- A client which initiates the job
- YARN resource manager
- YARN node manager
- MapReduce application master
- HDFS

Job initialisation

Starting a MapReduce job:

1. Request a new application ID to the resource manager
2. Check the job parameters
3. Split the job into subtasks
4. Copy the splits and other necessary information to run the job onto the shared FS
5. Effectively submit the job on the resource manager

Job startup

Running a MapReduce job:

1. YARN scheduler allocates a container
2. Application master launched by the resource manager
3. Setup the tasks
4. Retrieve the splits from the shared FS
5. Create a Map task for each split and specify the number of tasks for the Reduce part
6. Resources for
   a. Small tasks: run on the same node
   b. Large tasks: contact the resource manager for more containers
      i. Request resources for the maps (high priority)
      ii. Request resources for the reducers when enough maps have completed
7. Locate the data on the distributed FS and start the task

Job failure

Potential points of failure:

- Task failure
- Application master failure
- Node manager failure: YARN level (no heartbeat received)
- Resource manager failure: YARN level (high availability mode)

Job failure

Potential points of failure:

- Task failure
- Application master failure
- Node manager failure: YARN level (no heartbeat received)
- Resource manager failure: YARN level (high availability mode)

Protection and progress monitoring:

- Tasks are run in a separate JVM: avoid crashing namenode
- Each task has a status and some counters
- Each task reports its progress to the application master
- Client application polls the application master every second to retrieve the latest status

Task failure

Task failure occurrences and solutions:

- The map or reduce task fails:
  - When receiving a failure notice the application master marks the task as failed
  - The container is freed and resources released
- The JVM crashes: the node manager notices the application manager of the failure
- A task hangs:
  - Tasks marked as failed if no report is received
  - The JVM is killed by the application master

Task failure

Task failure occurrences and solutions:

- The map or reduce task fails:
  - When receiving a failure notice the application master marks the task as failed
  - The container is freed and resources released
- The JVM crashes: the node manager notices the application manager of the failure
- A task hangs:
  - Tasks marked as failed if no report is received
  - The JVM is killed by the application master

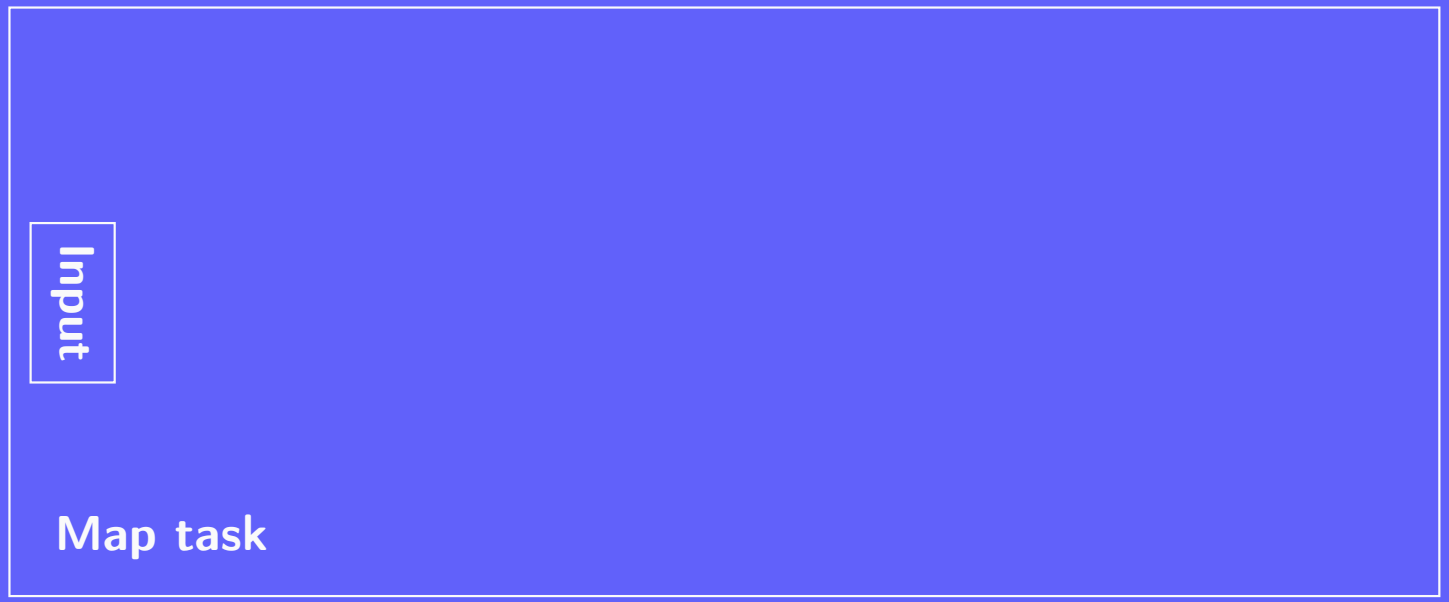
Failed tasks are rescheduled on a different nodemanager

Application master failure

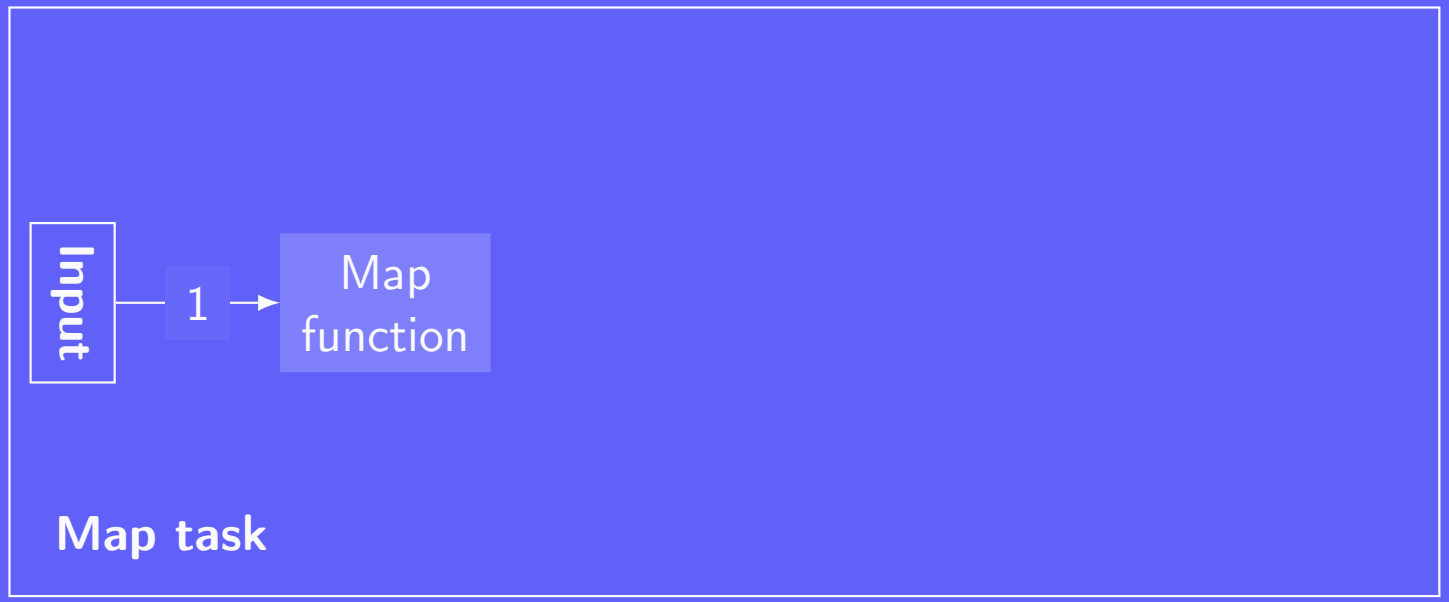
Failure detection and recovery:

- The application master sends periodic heartbeats to the resource manager
- On failure a new instance is run on a new container
- The tasks progress is known so it is possible to resume without rerunning the completed tasks
- Each task caches the application master's address
- Use a timeout after which the resource manager is contacted for the new application's master address

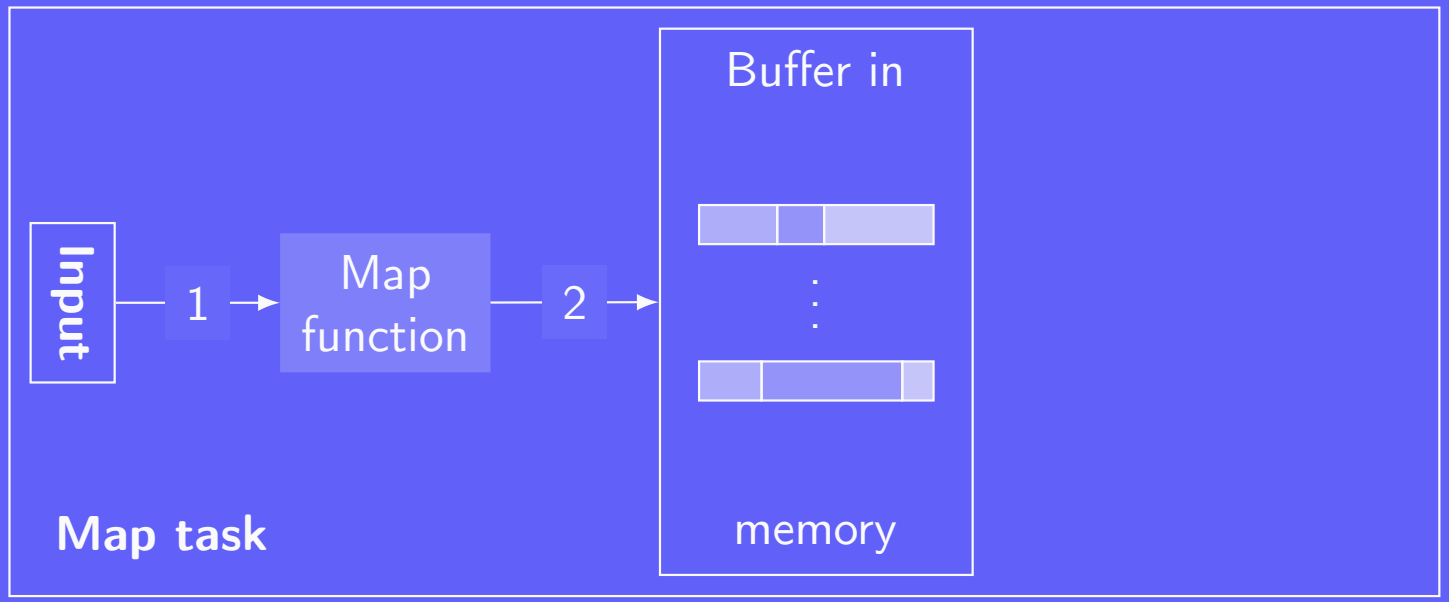
Map and Reduce



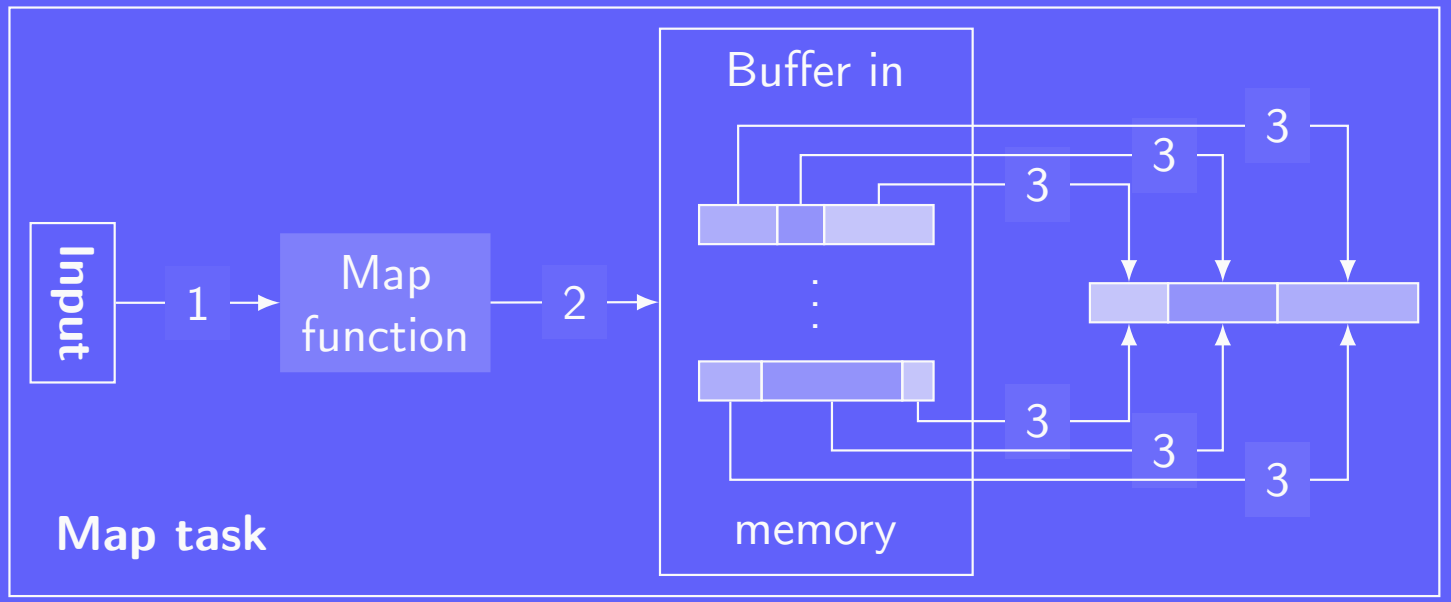
Map and Reduce



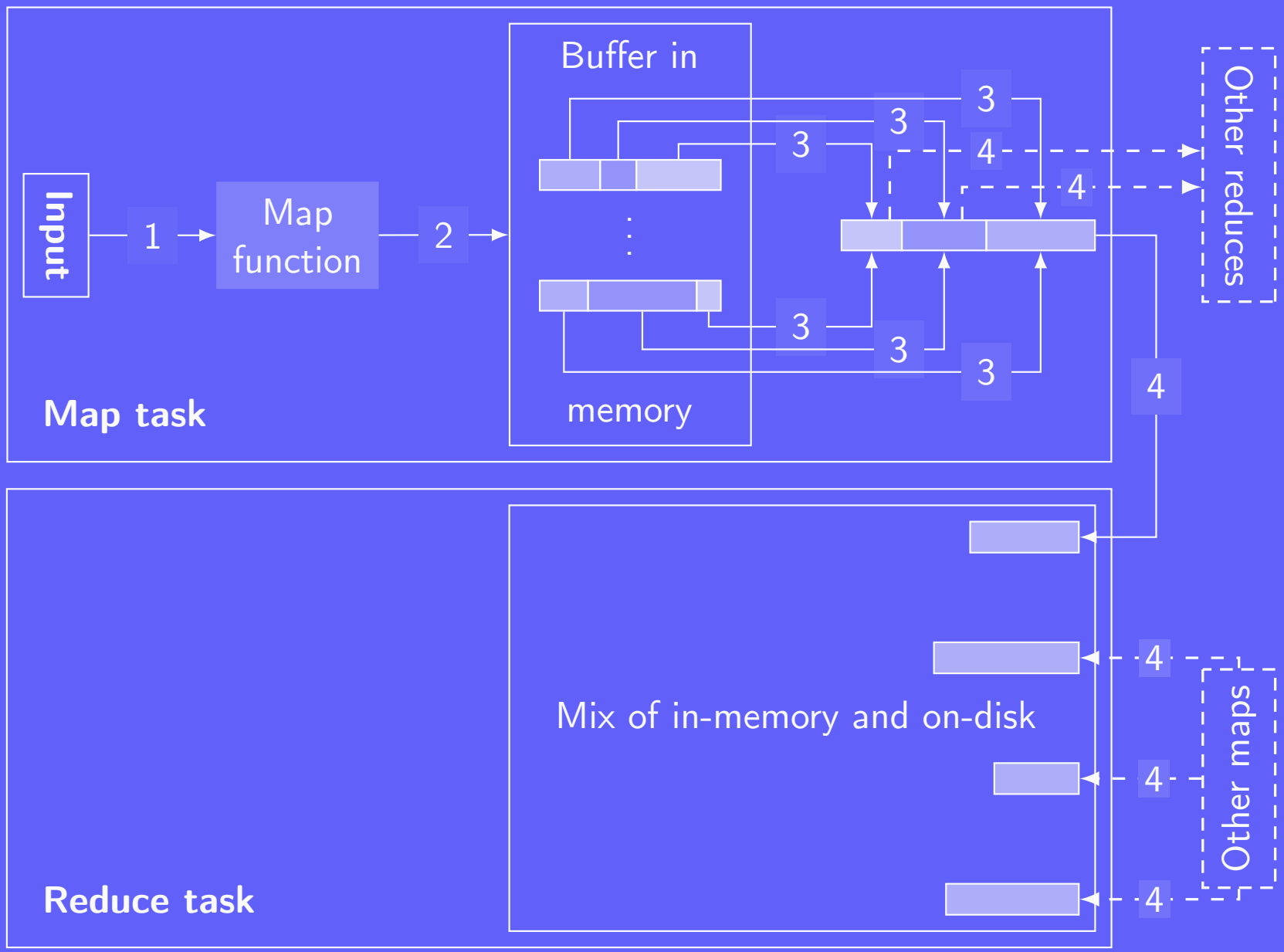
Map and Reduce



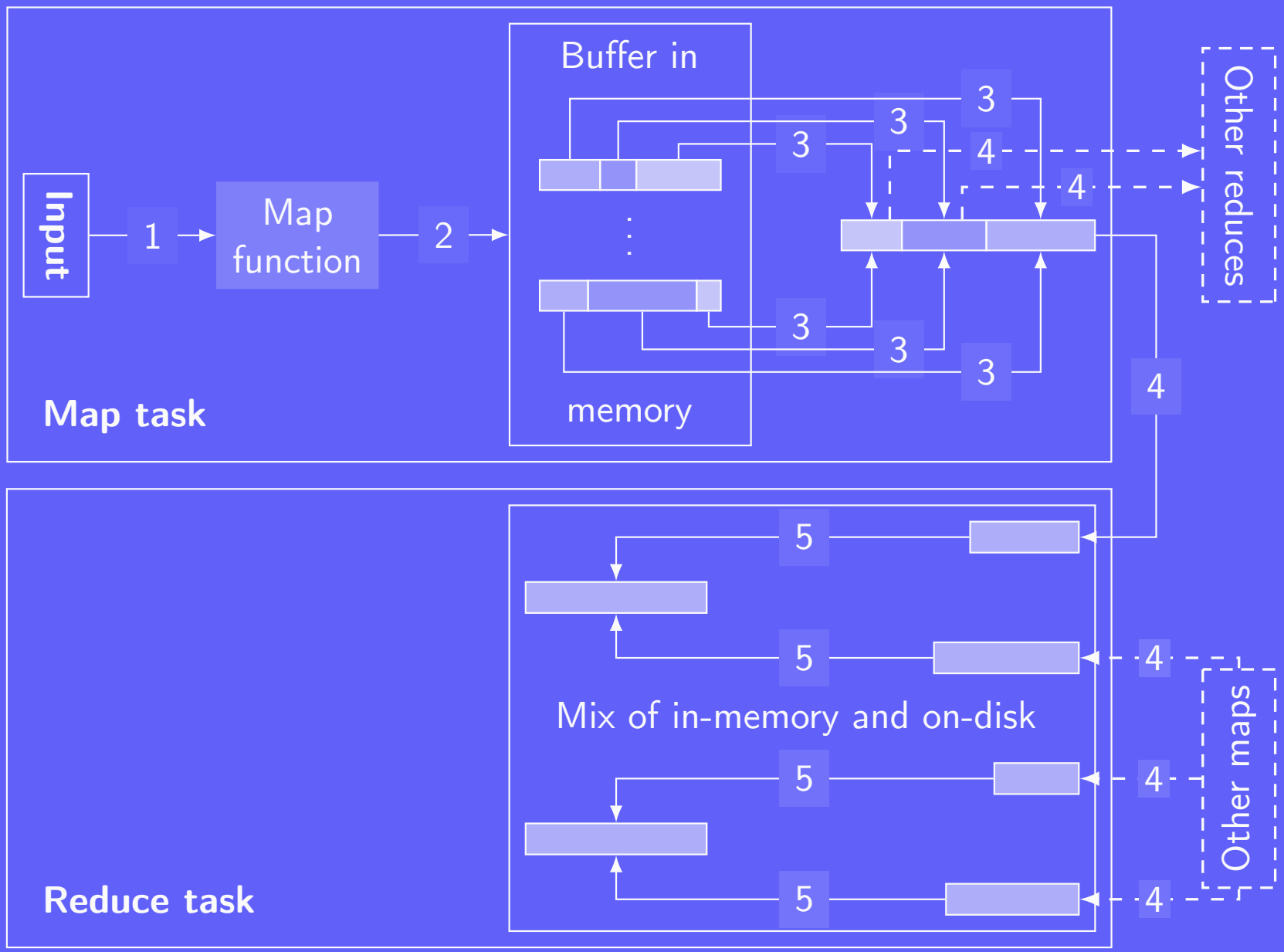
Map and Reduce



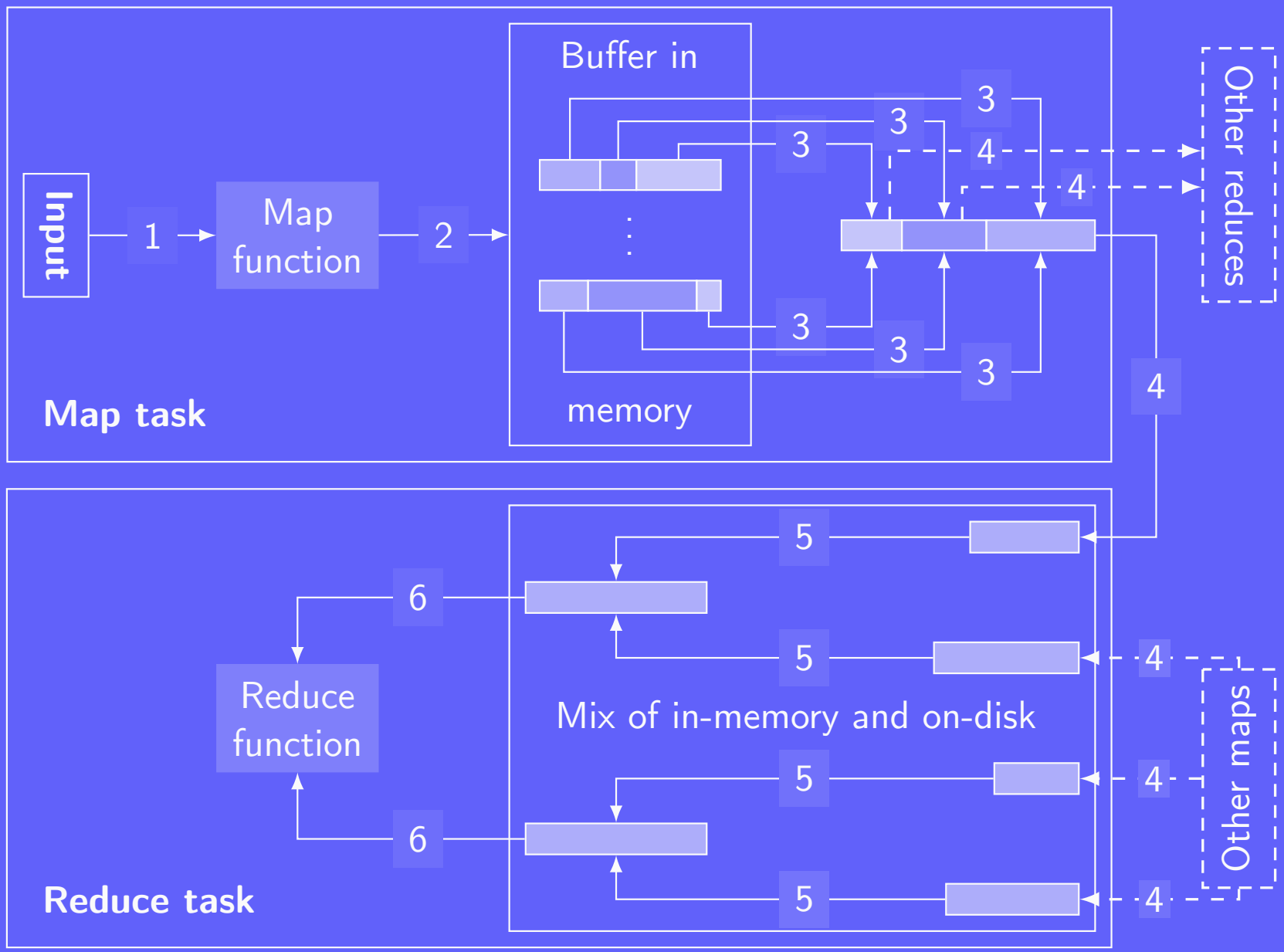
Map and Reduce



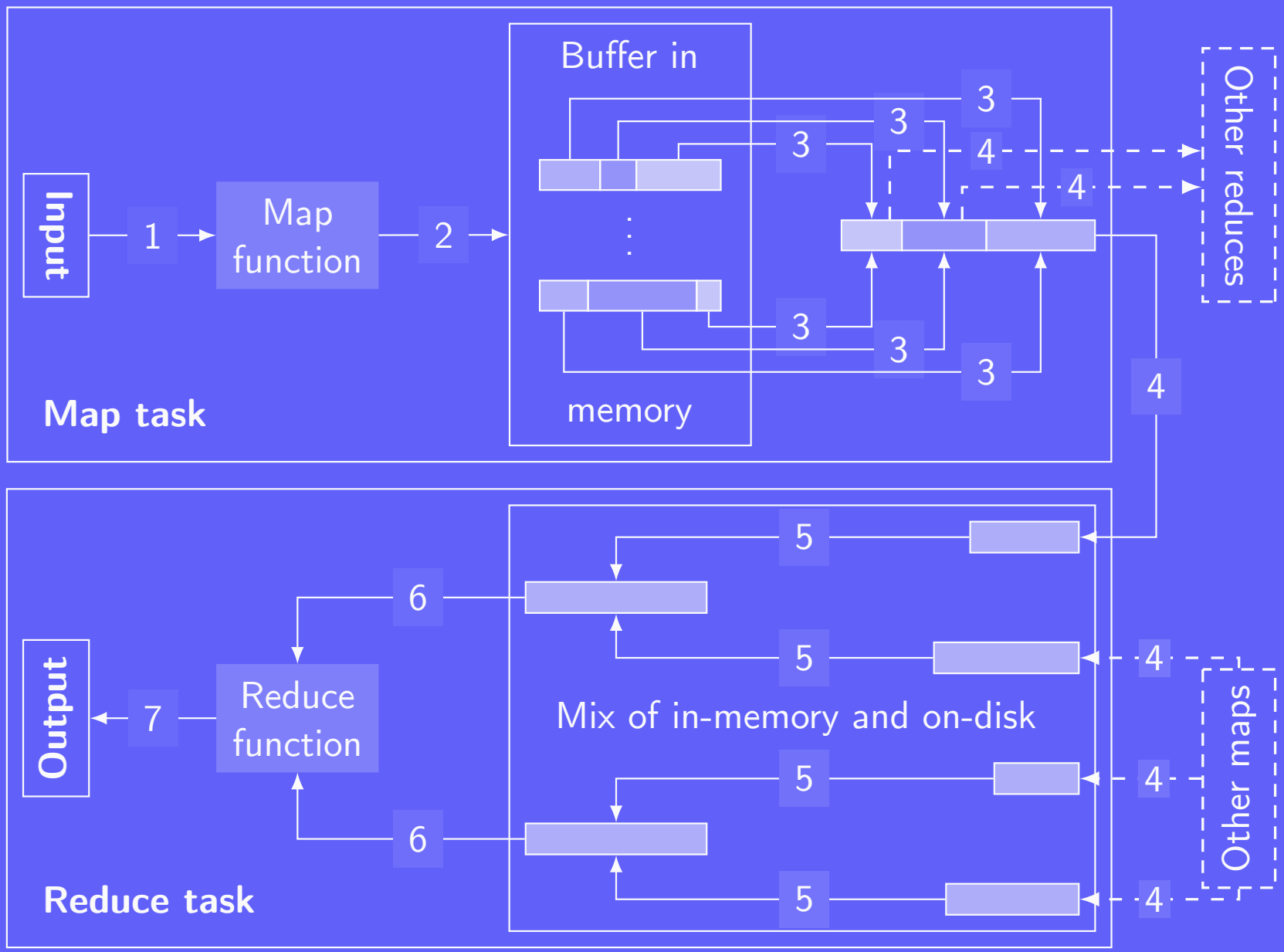
Map and Reduce



Map and Reduce



Map and Reduce



Speeding up a MapReduce job

Optimized configuration setup:

- Provide shuffle with as much memory as possible
- Keep enough memory for map and reduce functions
- Optimize the code with respect to memory consumption
- Minimize the number of spills for the map part
- As much as possible keep intermediate reduce data in memory

Speeding up a MapReduce job

Optimized configuration setup:

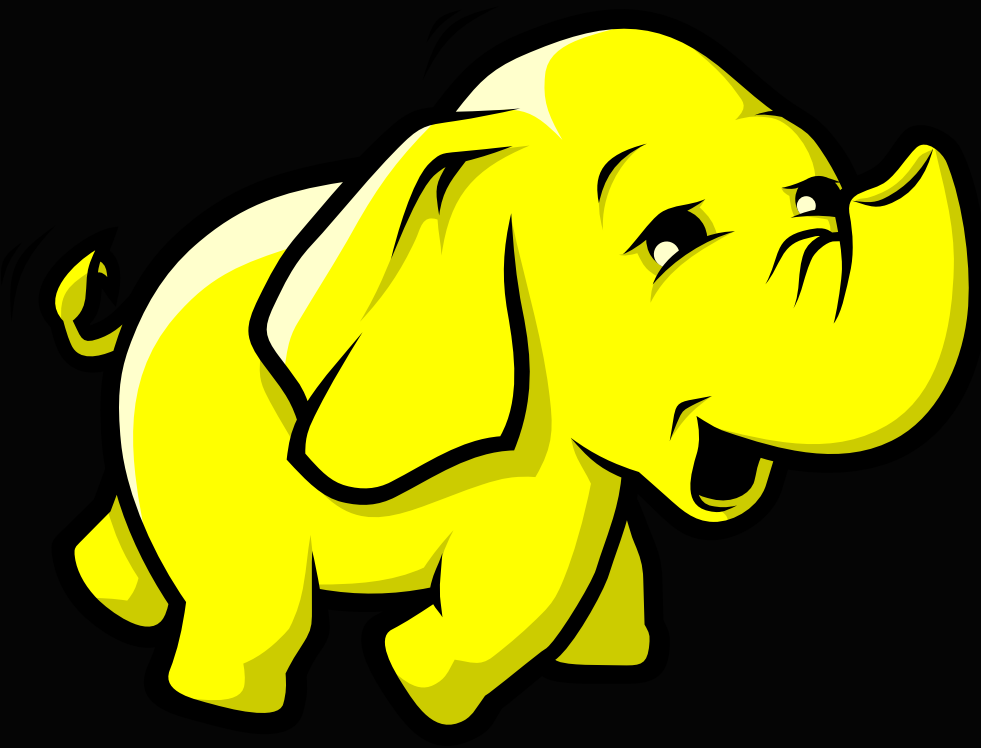
- Provide shuffle with as much memory as possible
- Keep enough memory for map and reduce functions
- Optimize the code with respect to memory consumption
- Minimize the number of spills for the map part
- As much as possible keep intermediate reduce data in memory

Speculative execution:

- A task is detected as much slower than average
- Re-run it on a different node
- Kill all the other duplicates as soon as one completes

Key points

- Explain the basic structure of Hadoop
- What are the main principles behind HDFS?
- How is YARN used by higher level applications?
- Describe the structure of a MapReduce job



Thank you!