CentraleSupelec 2018-2019 MSC DSBA / DATA SCIENCES

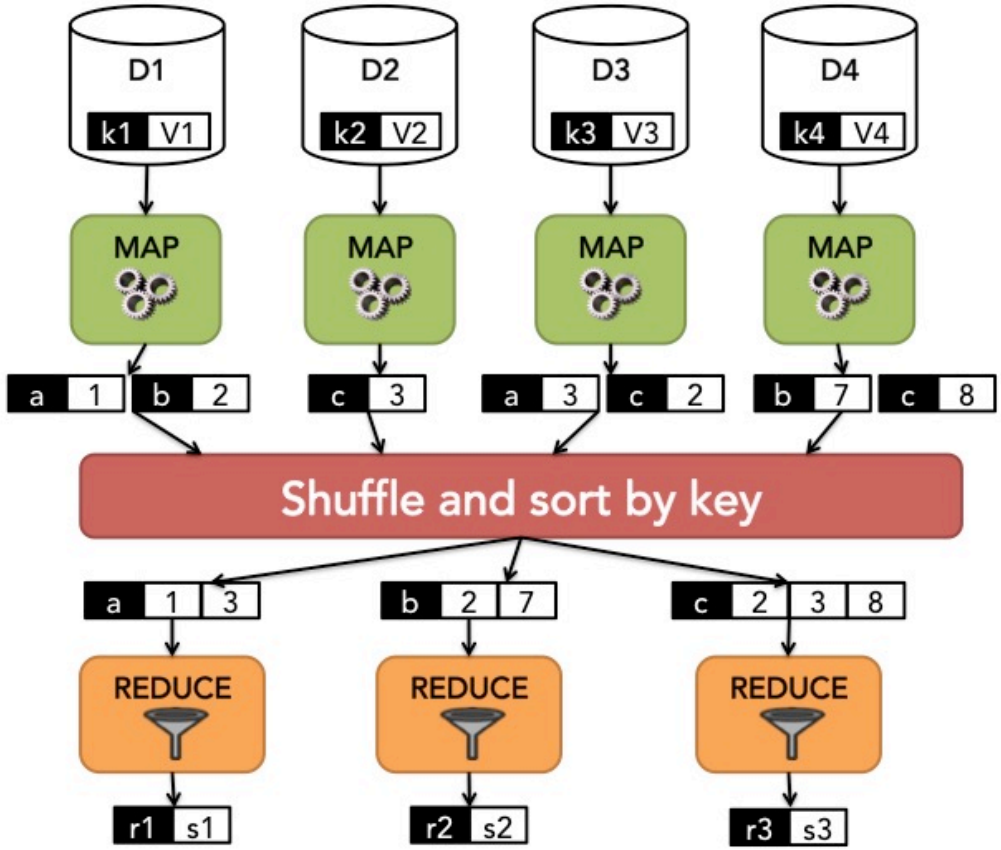
Big Data Algorithms, Techniques and Platforms

Distributed Computing with MapReduce and Hadoop, part 2

Hugues Talbot & Céline Hudelot, professors.

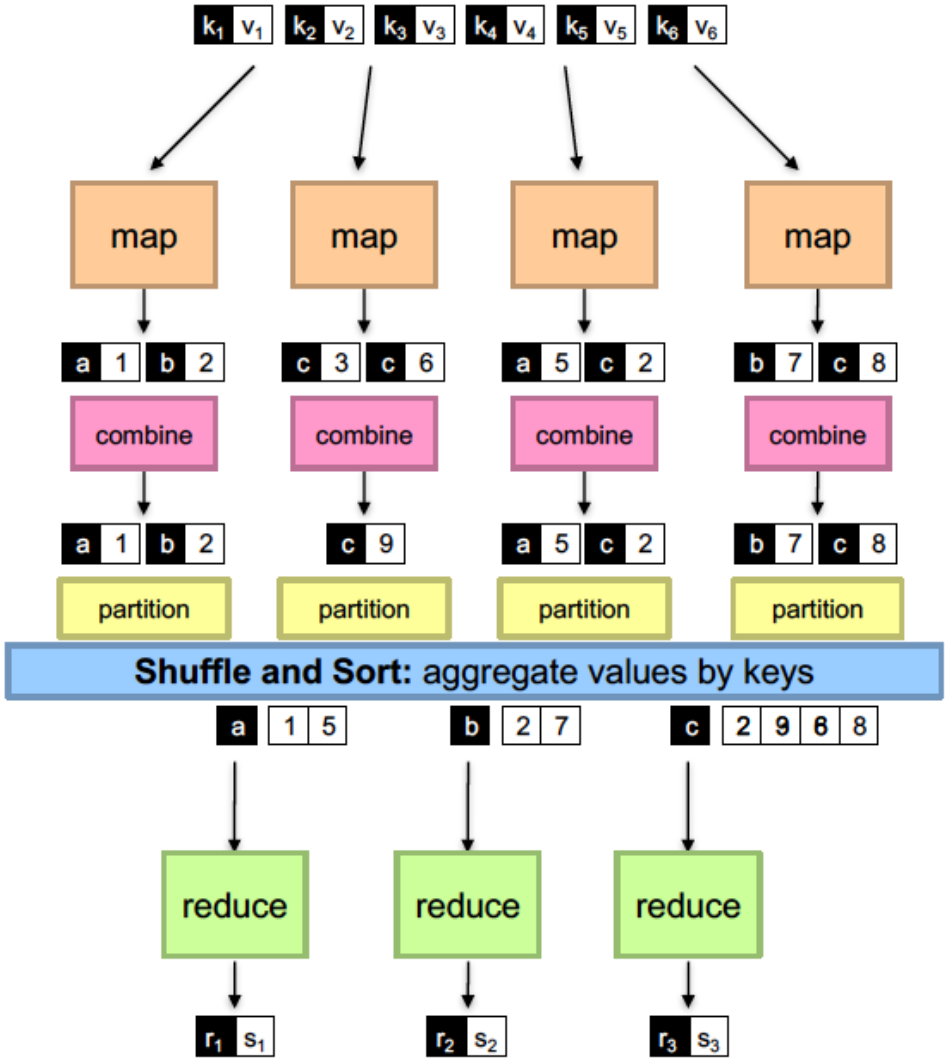
Recap : MapReduce

- A simple programming model for processing huge data sets in a distributed way.
- A framework that runs these programs on clusters of commodity servers, automatically handling the details of distributed computing



Recap : MapReduce

With combiners and partitioners



Recap : Algorithm Design with MapReduce

Developing algorithms involve :

- Preparing the input data.
- Writing the mapper and the reducer.
- And, optionally, writing the combiner and the partitioner.

How to recast existing algorithms in MapReduce :

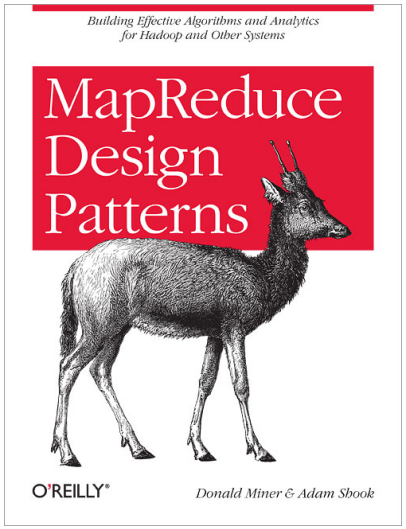
- Not always obvious.
- Importance of data structures : complex data structures as keys and values.
- Difficult to optimize.

Learn by practice and examples

- MapReduce design patterns.
- Synchronization of intermediate results is difficult : the tricky task.

Recap : MapReduce Design Patterns

- Building effective algorithms and analytics for Map Reduce
- 23 patterns grouped into 6 categories :
  - Summarization
  - Filtering
  - Data Organization
  - Joins
  - Input and Output
  - Metapatterns

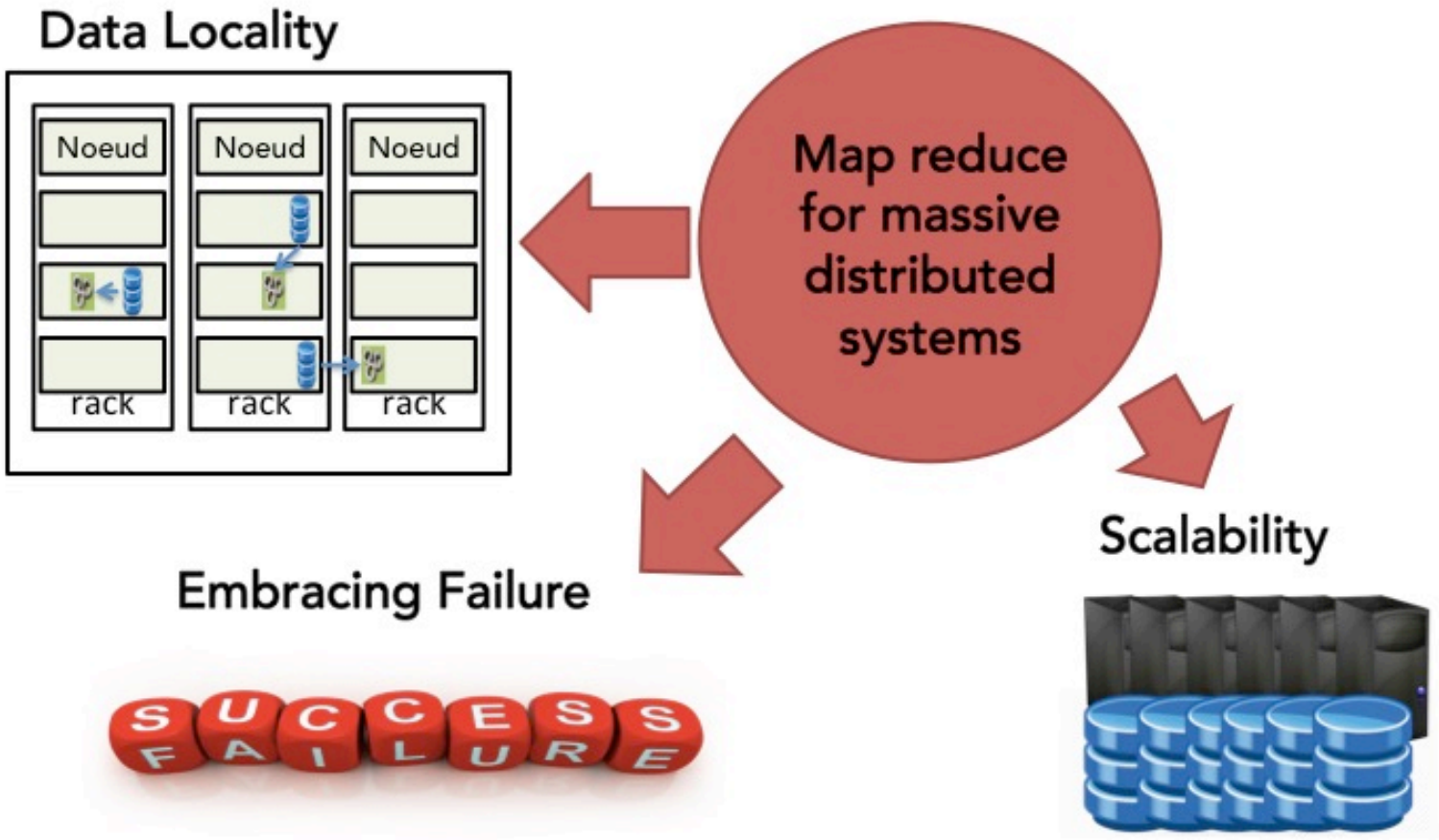


Program of the day

- MapReduce : the execution framework.
- Hadoop and its ecosystem

Plan

MapReduce in real life



MapReduce : the execution framework

MapReduce program : a job

- Code for mappers and reducers.
- Code for combiners and partitioners.
- Configuration parameters.

A MapReduce job is submitted to the cluster

- The framework takes care of everything else.

MapReduce : runtime execution

MapReduce runtime execution environment handles :

- sheduling : assigns workers to map and reduce tasks.
- data distribution : partitions the input data and moves processes to data.
- synchronisation : manages required inter-machine communication to gather, sort and shuffle intermediate data.
- errors and faults : detects worker failures and restarts.

MapReduce : Scheduling

- **Each job is broken into tasks (smaller units).**
  - Map tasks work on fractions of the input dataset.
  - Reduce tasks work on intermediate inputs and write back to the distributed file system.
- **The number of tasks may exceed the number of available machines in a cluster**
  - The scheduler takes care of maintaining something similar to a queue of pending tasks to be assigned to machines with available ressources.
- **Jobs to be executed in a cluster requires scheduling as well**
  - Different users may submit jobs.
  - Jobs may be of various complexity.
  - Fairness is generally a requirement.

MapReduce : data flow

- The input and final output are stored on a distributed file system.
  - Scheduler tries to schedule map tasks close to physical storage location of input data.
- The intermediate results are stored on the local file system of map and reduce workers.
- The output is often the input of another MapReduce task.

MapReduce : Synchronization

- Synchronization is achieved by the *Shuffle and Sort* step.
  - Intermediate key-value pairs are group by key.
  - This requires a distributed sort involving all mappers, and taking into account all reducers.
  - If you have m mappers and r reducers, it involves $m \times r$ copying operations.
- Important : the reduce operation cannot start until all mappers have finished (to guarantee that all values associated with the same key have been gathered)

MapReduce : coordination

Master keeps an eye on each task status (Master-slave architecture)

- idle tasks get scheduled as workers become available
- When a map task completes, it sends Master the location and sizes of its R intermediate files, one for each reducer and then Master pushes this info to reducers.
- Master pings workers periodically to detect and manage failures.

MapReduce : implementations

- Google has a proprietary implementation in C++.
- Hadoop is an open-source implementation in JAVA.
  - Development led by Yahoo, used in production.
  - Apache project.
  - A big software ecosystem.
- Lot of custom research implementations.

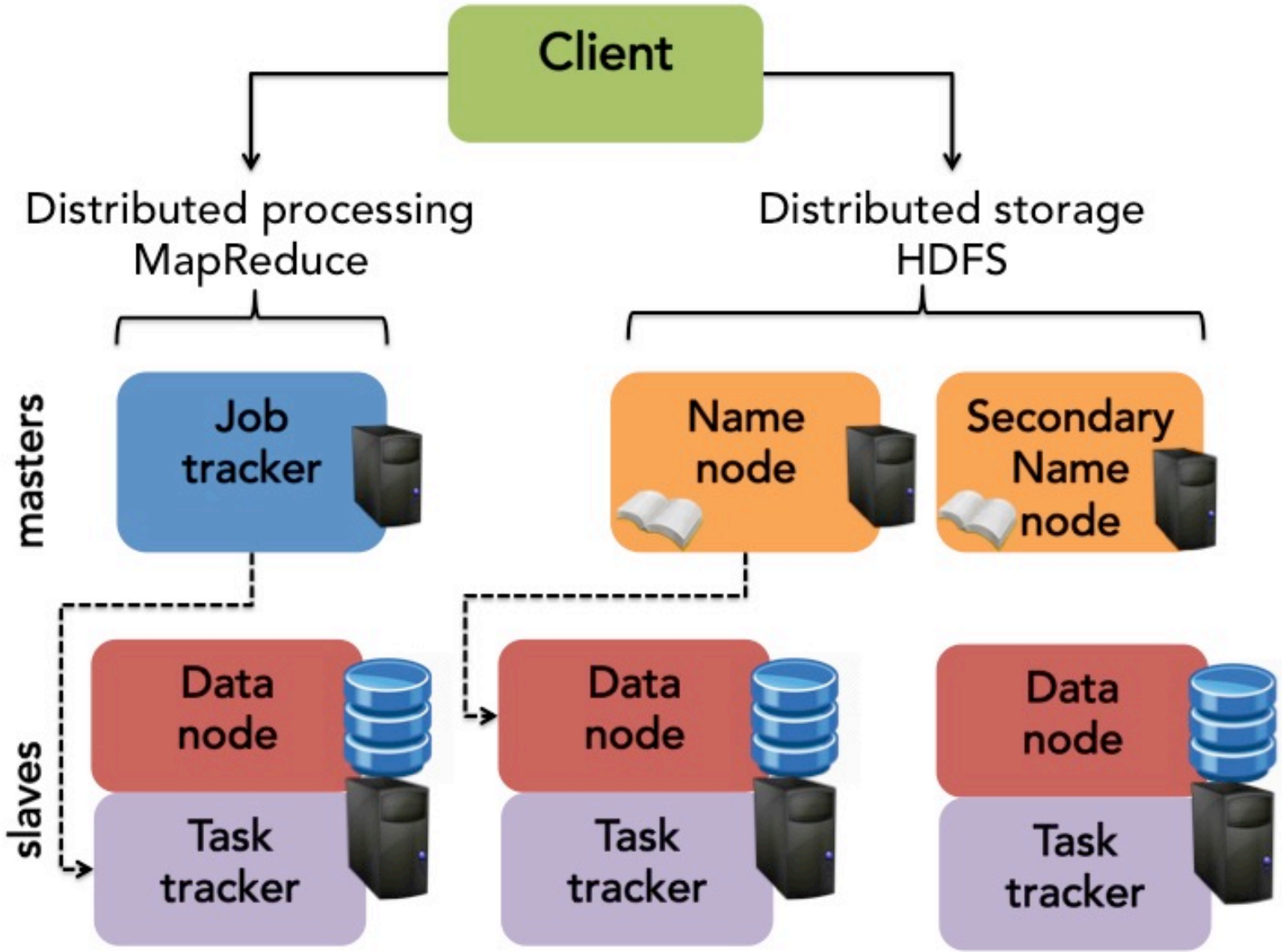
Hadoop

Hadoop

Two main components

- HDFS
- MapReduce

Hadoop : two main components



Hadoop : HDFS

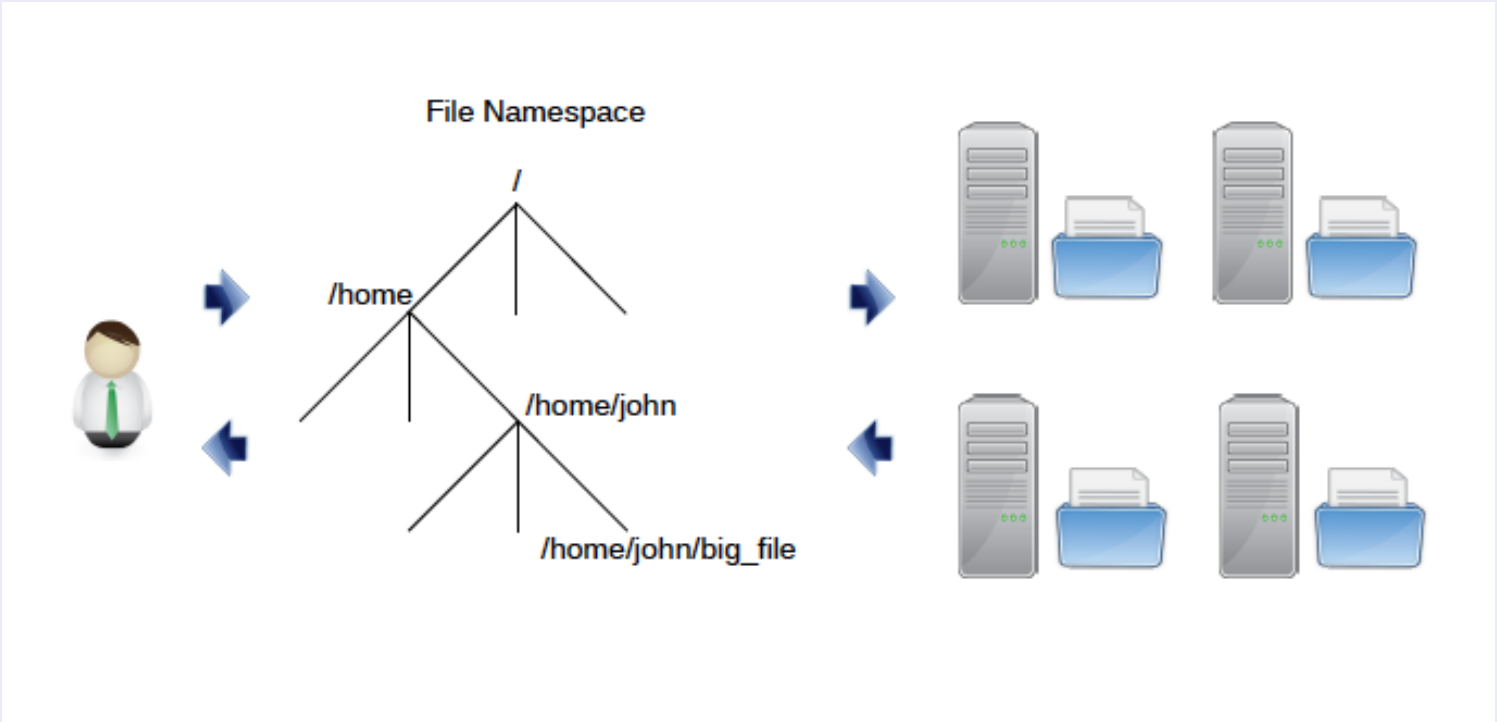
HDFS : A distributed file system.

Why do we need a Distributed File System ?

- Read large data fast.
  - scalability : perform multiple parallel reads and writes.
- Files have to be available even if one computer crashes
  - fault tolerance : replication.
- Hide parallelization and distribution details.
  - transparency : clients can access it like a local filesystem.

Distributed File System

What is a Distributed File System ?



Source : Lars Schmidt

Distributed File System

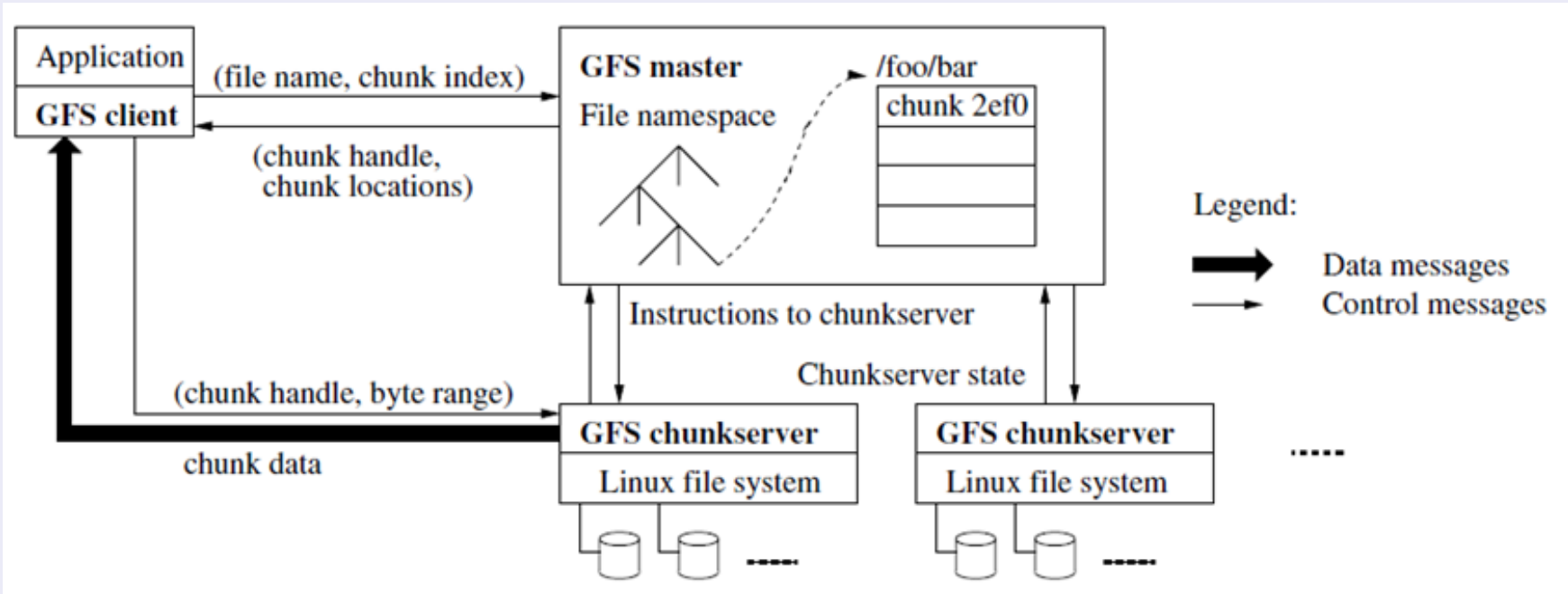
Components a Distributed File System ?

A typical Distributed File System contained :

- Clients : interface with the user.
- Chunk nodes : nodes that store chunks (blocks) of files.
- Master node : node that stores which parts of each file are on which chunk node.

Distributed File System

Google File System



Hadoop : HDFS

- HDFS is a file system which is
  - Distributed : the files are distributed on the different nodes of the cluster.
  - Replicated : in case of failure, data is not lost.
  - *Data locality : programs and data are colocated.*
  - Files in HDFS are broken into block-sized chunks, which are stored as independent units.

Hadoop : HDFS

HDFS blocks

- Files are broken into block-sized chunks.
  - Blocks are big ! [64,128] MB
- Blocks are stored on independent machines.
  - Replicate across the local disks of nodes in the cluster.
  - Reliability and parallel access.
  - Replication is handled by storage node themselves.

Hadoop : HDFS

HDFS looks like an unix file system

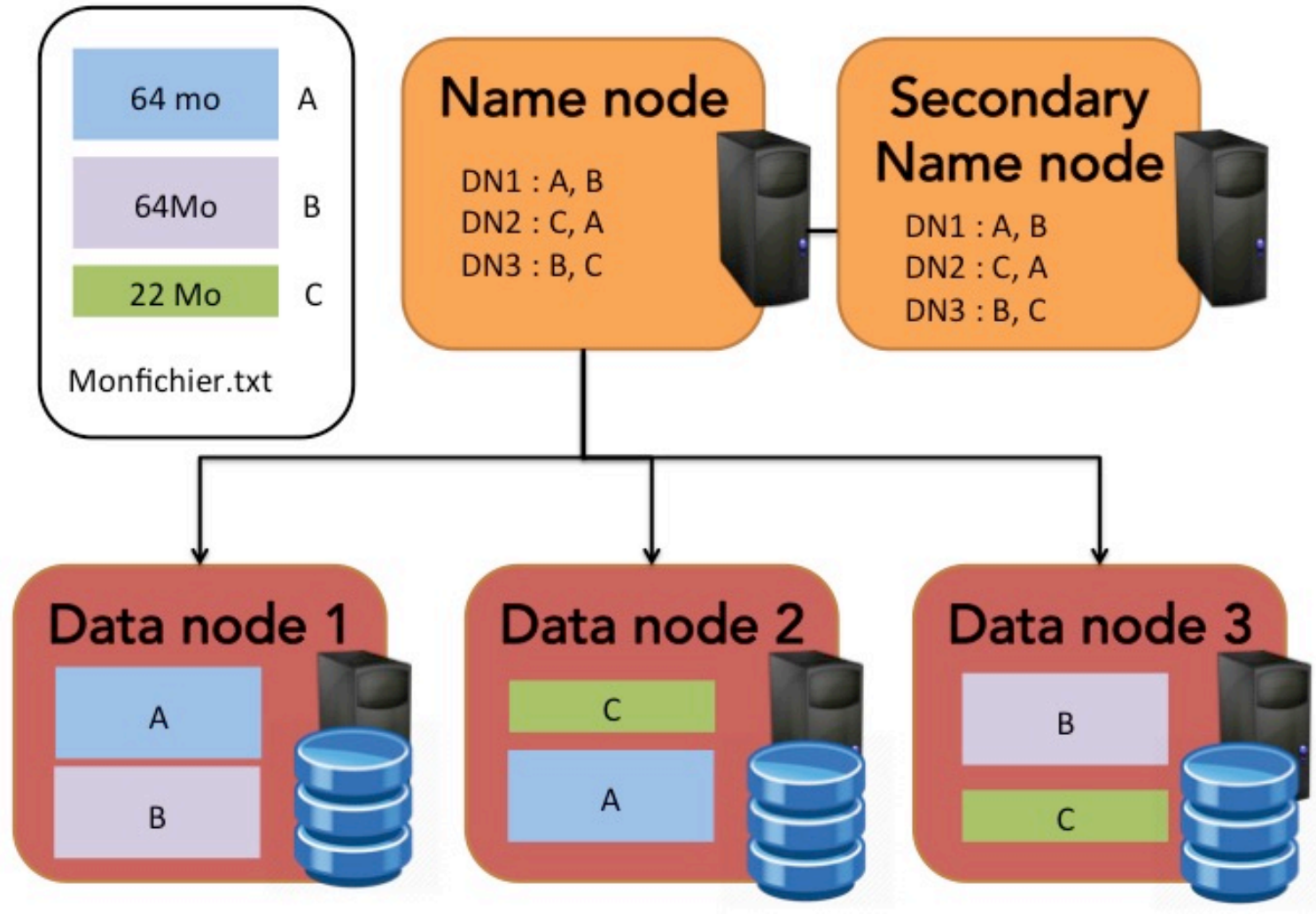
- Root : `/`
- Some directories for the Hadoop services under the root : `/hbase`, `/tmp`, `/var`
- A directory for the personal files of users : `/user`
- A directory for shared files : `/share`

Hadoop : HDFS

- Some commands : `hdfs dfs` or `hadoop dfs`
- An API JAVA

Hadoop : HDFS architecture

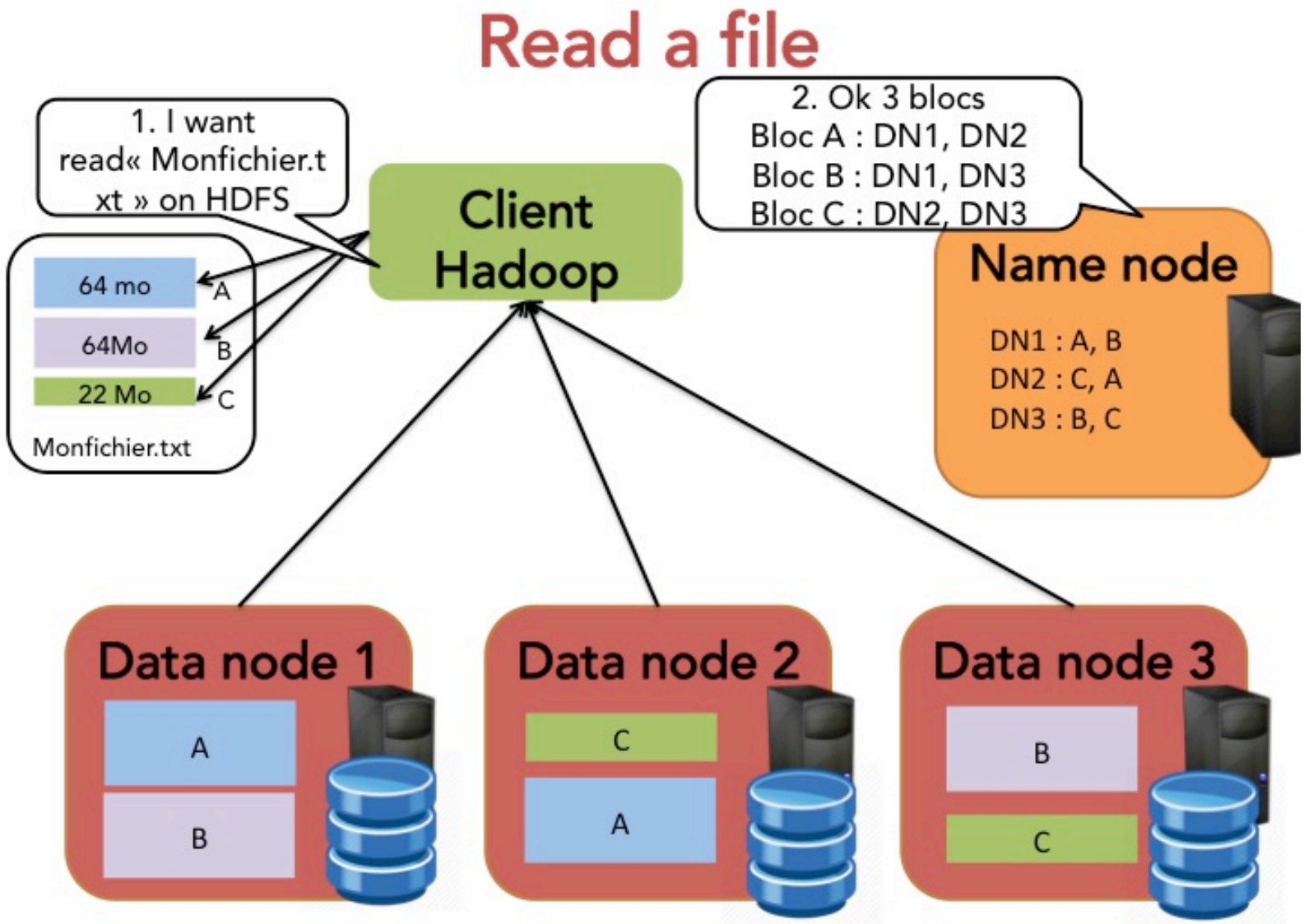
A master slave architecture



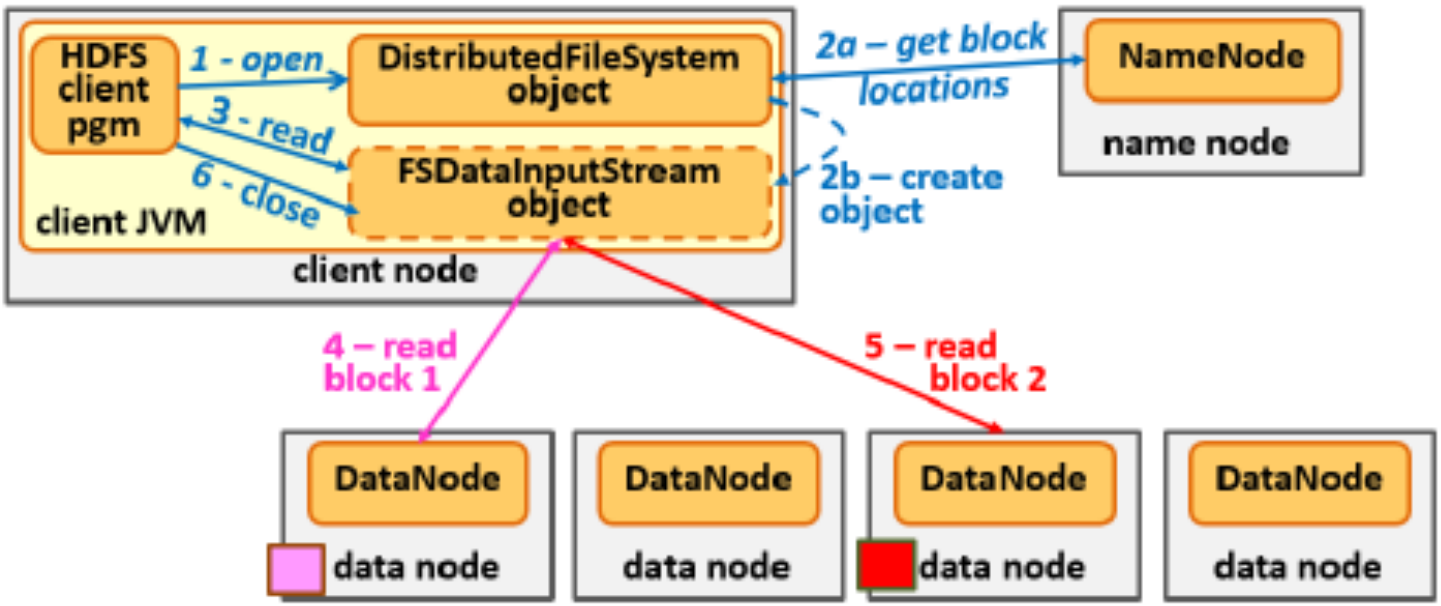
Hadoop : HDFS architecture

- **Namenode** : a kind of big directory : it stores the names and the blocks of all files as well as their location.
- **Secondary Namenode** : replication of the namenode in case of failure.
- **Datanode** : storage node that manages the local storage operations : creation, removal and duplication.

HDFS architecture : read a file

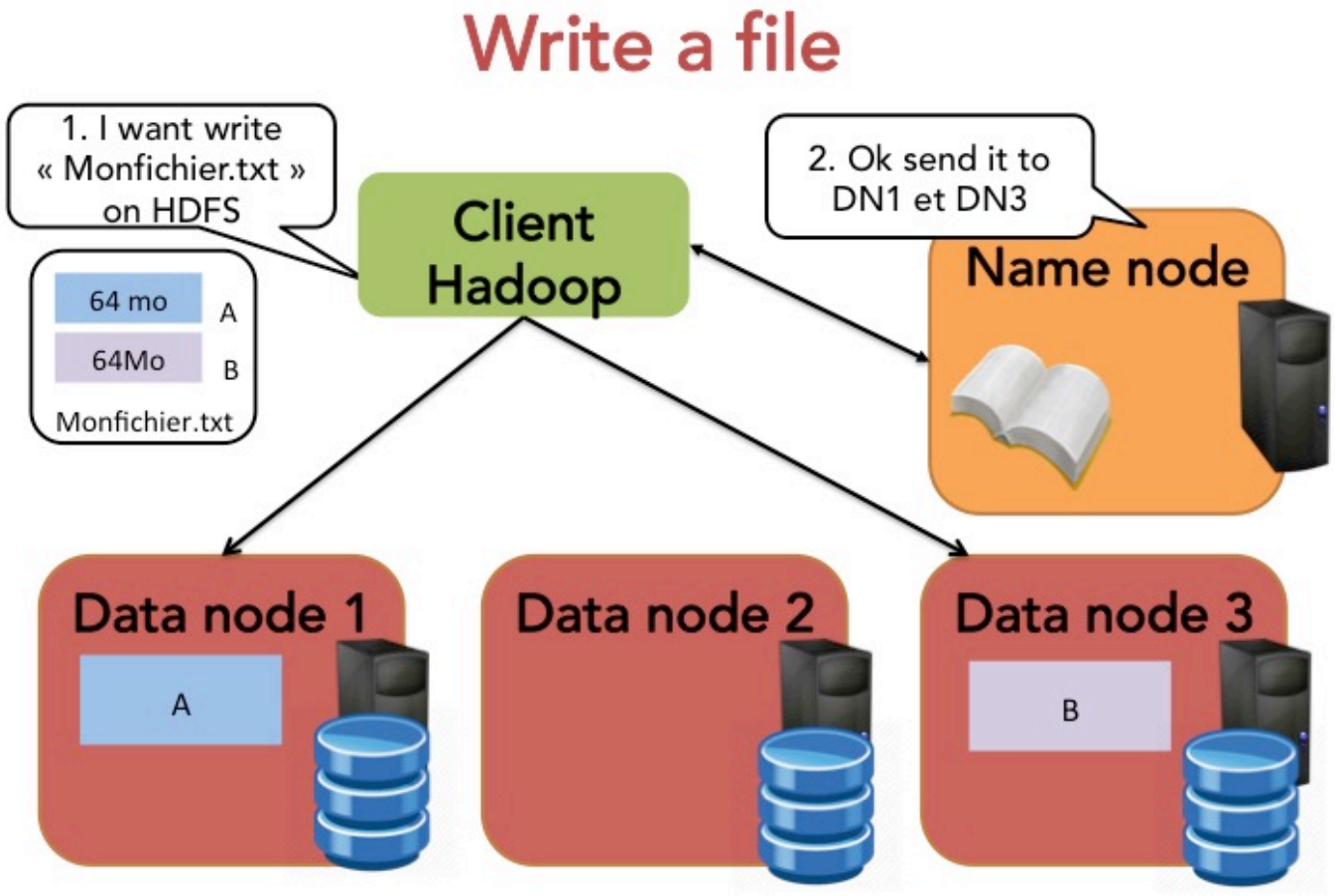


HDFS architecture : read a file



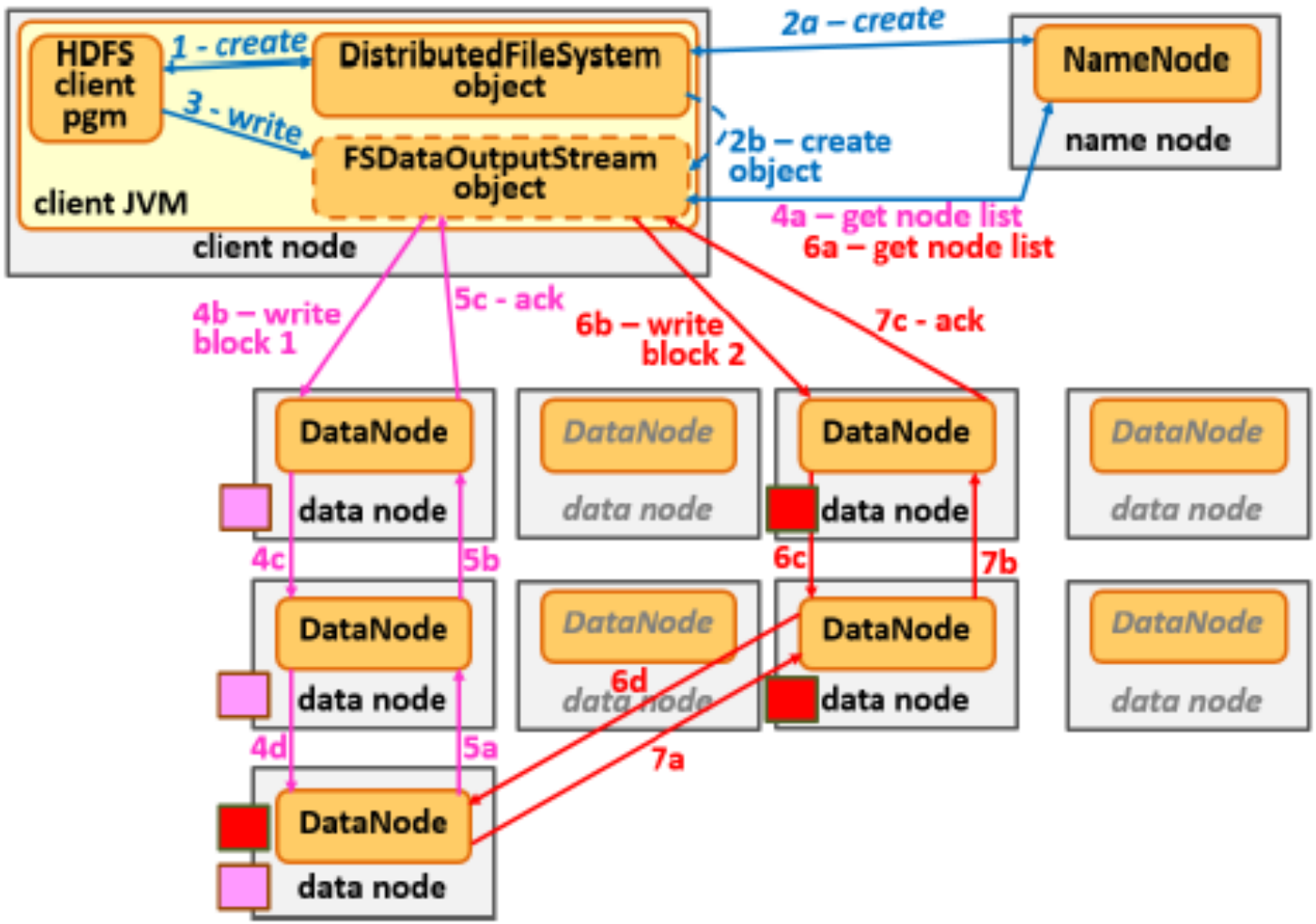
Source : S. Vialle

HDFS architecture : write a file

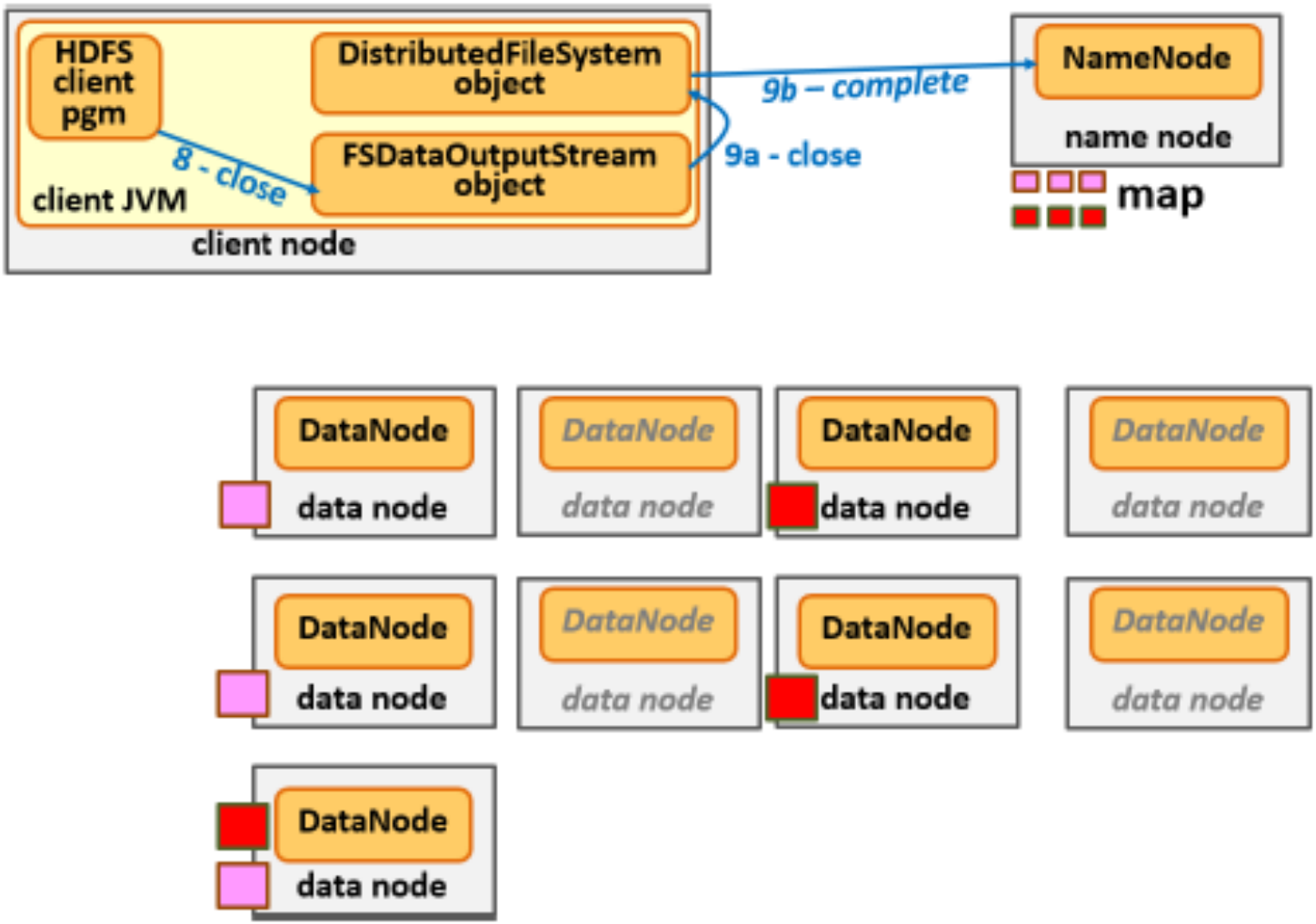


More details here : `http://bradhedlund.com/2011/09/10/understanding-hadoop-clusters-and-the-network/`

HDFS architecture : write a file



HDFS architecture : write a file



Replication Management

- Each chunk is stored redundantly on several chunk nodes (replication).
- Chunk node regularly send an *I-am-alive* message to the master (heartbeat)
- A chunk node without heartbeat for a longer period is considered offline/down/dead.
- If a chunk node is found to be offline, the namenode creates new replicas of its chunks .

HDFS Setup

Prerequesites

- Several machines (≥ 1) with password-less ssh login.
  - here : h_0, h_1
- Java install on all machines
  - Test : on h_0 : java -version and on h_1 : ssh h_1 java -version
- Hadoop downloaded and unpacked on all machines
  - Hadoop binaries have to be added in the path.
  - Test : on h_0 : hadoop version and on h_1 : ssh h_1 hadoop version

HDFS Setup

Configuration

- With the two files core-site.xml and hdfs-site.xml

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

- Test : on h_0 : hdfs getconf -namenodes and on h_1 : ssh ...

HDFS Setup

Start hdfs

- On h_0.
  - hdfs namenode -format : format disk /, crate data structures
  - hdfs namenode : start namenode daemon
  - hdfs datanode : start datanode daemon
- On h_1.
  - hdfs datanode : start datanode daemon
- test : on h_0 : hdfs dfsadmin -report or with the web interface.

HDFS FileSystem Interface

hdfs dfs -<command>

- hdfs dfs -df <path> : show free disk space
- hdfs dfs -ls <path> : list directory
- hdfs dfs -mkdir <path> : create directory
- hdfs dfs -put <files> ... <dir> : upload files to hdfs
- hdfs dfs -get <paths> ... <dir> : download files from hdfs
- hdfs dfs -cat <paths> : pipe files from hdfs to stout
- hdfs dfs -mv <src> ... <dest> : move or rename files on hdfs
- hdfs dfs -cp <src> ... <dest> : copy files on hdfs

HDFS FileSystem Interface

Inspect File Health hdfs fsck <path> -files -blocks -locations

```
root@hadoop-master:~# hdfs fsck input/file1.txt
Connecting to namenode via http://hadoop-master:50070/fsck?ugi=root&path=%2
FSCK started by root (auth:SIMPLE) from /172.18.0.2 for path /user/root/inp
.Status: HEALTHY
 Total size: 13 B
 Total dirs: 0
 Total files: 1
 Total symlinks: 0
 Total blocks (validated): 1 (avg. block size 13 B)
 Minimally replicated blocks: 1 (100.0 %)
 Over-replicated blocks: 0 (0.0 %)
 Under-replicated blocks: 0 (0.0 %)
 Mis-replicated blocks: 0 (0.0 %)
 Default replication factor: 2
 Average block replication: 2.0
 Corrupt blocks: 0
 Missing replicas: 0 (0.0 %)
 Number of data-nodes: 2
 Number of racks: 1
```

HDFS Java API

Two main classes :

- FileSystem : to represent the file system and to manage file and directory (copy, rename, create, delete).
- FileStatus : to manage the information related to a file.
- Configuration : to know the configuration of the HDFS cluster.
- Path : to represent the complete name of a file.

HDFS in Python

- snakebite, a Pure Python HDFS client can be used (proposed by Spotify).
  `https://github.com/spotify/snakebite`

DFS Summary

- Basic requirements for Distributed File Systems :
  - scalability : perform multiple parallel reads and writes.
  - fault tolerance : replicate files on several nodes.
  - transparency : clients can access files like on a local file system
- Distributed File Systems partition files into chunks/blocks.
  - datanodes : store individual blocks of a file.
  - namenode (master) : stores the index
- reading can be done from any datanode storing a block.
- writing needs to be synchronized over datanodes storing a block.
  - For every block, there is a primary datanode.

Hadoop MapReduce : anatomy of a job

Job = MapReduce Task

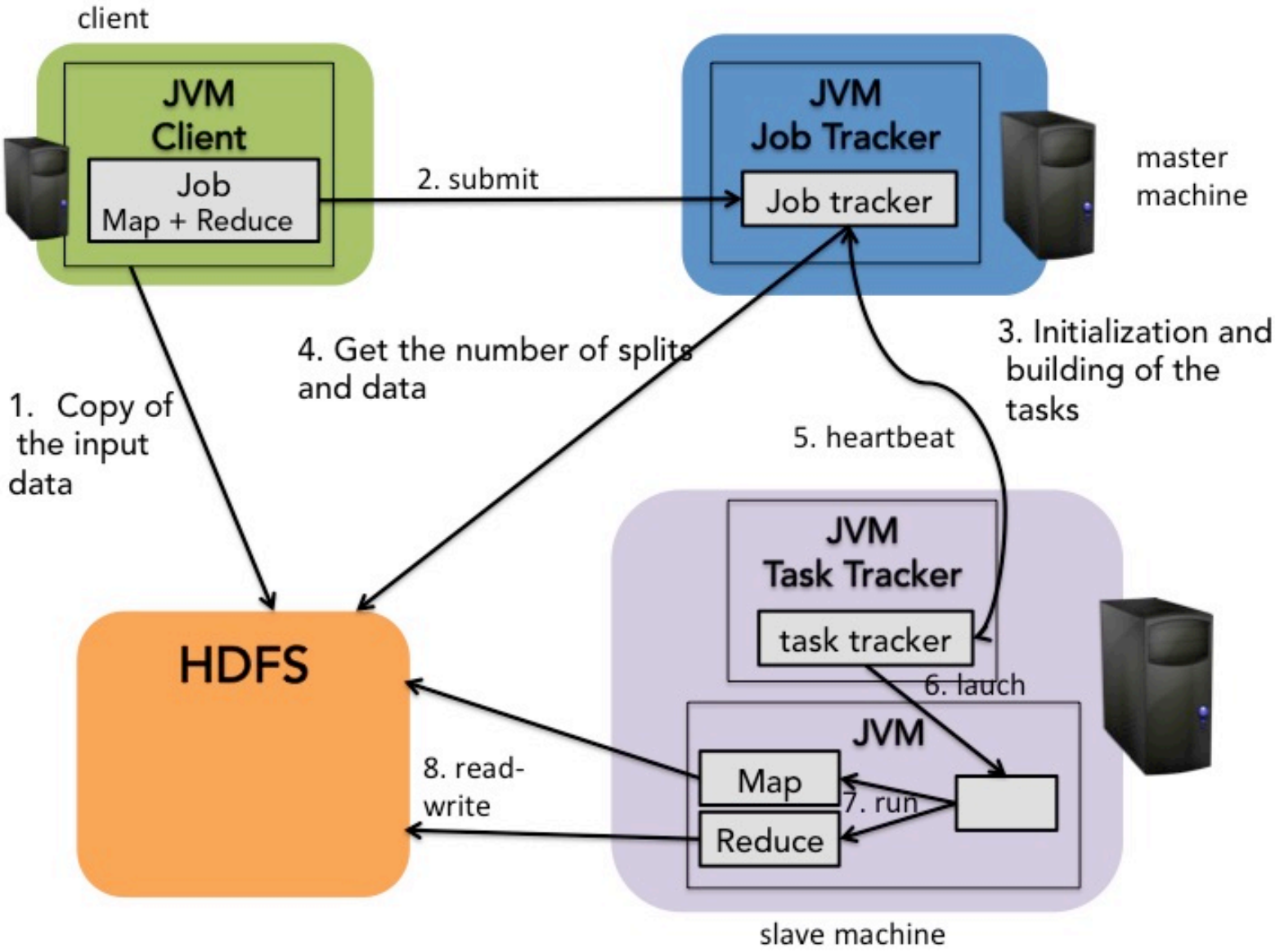
Job submission process

- Client creates a job, configures it and submits it (as a jar archive) to the job tracker.
- The job tracker asks to the namenode where are the blocks corresponding to the input data.
- The job tracker puts jobs in shared location, enqueue tasks
- Task trackers poll for tasks (and heartbeat)

Hadoop MapReduce : anatomy of a job

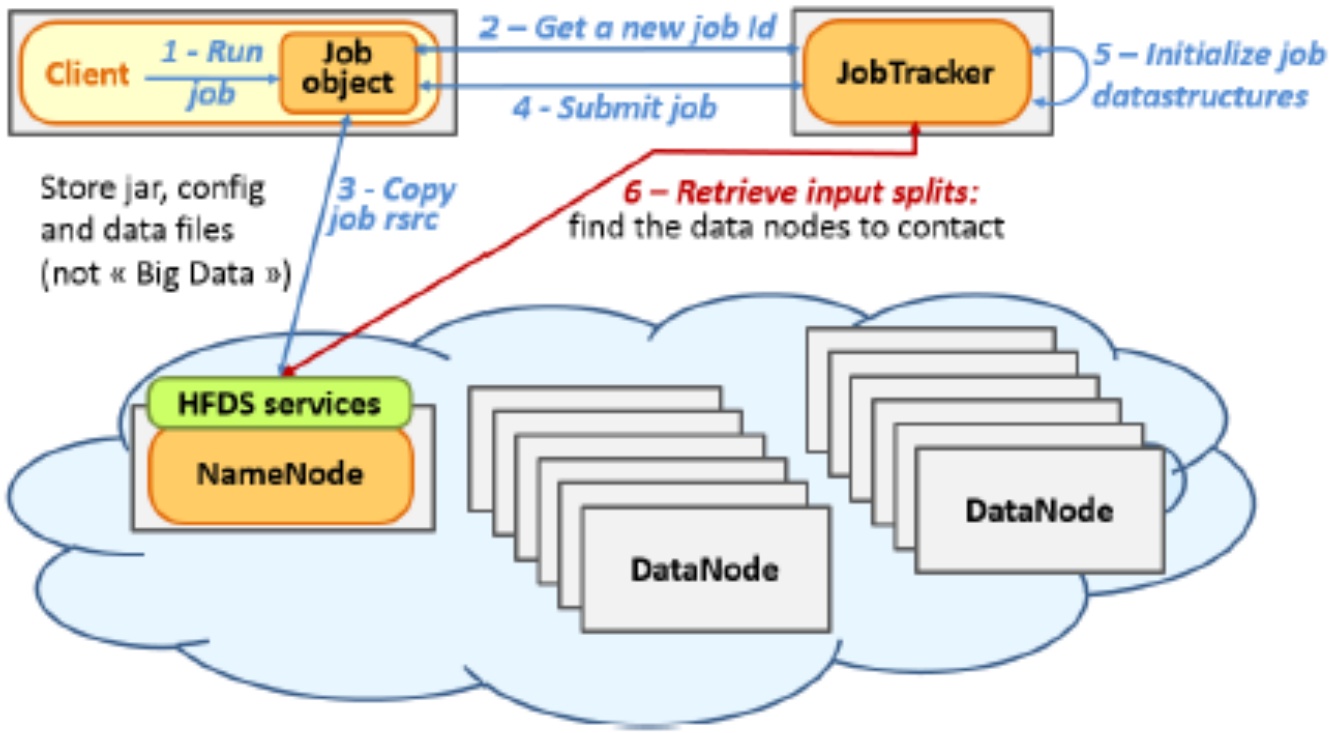
- **Job tracker** : master processus that will be in charge of both processing scheduling and ressource management. It distributes the different mapreduce tasks to the task trackers by taking into account the location of the data.
- **Task tracker** : a slave computing unit. It runs the execution of map and reduce tasks (by the launch of a JVM). It communicates with the job tracker on its status.

Hadoop MapReduce : anatomy of a job



Hadoop MapReduce : anatomy of a job

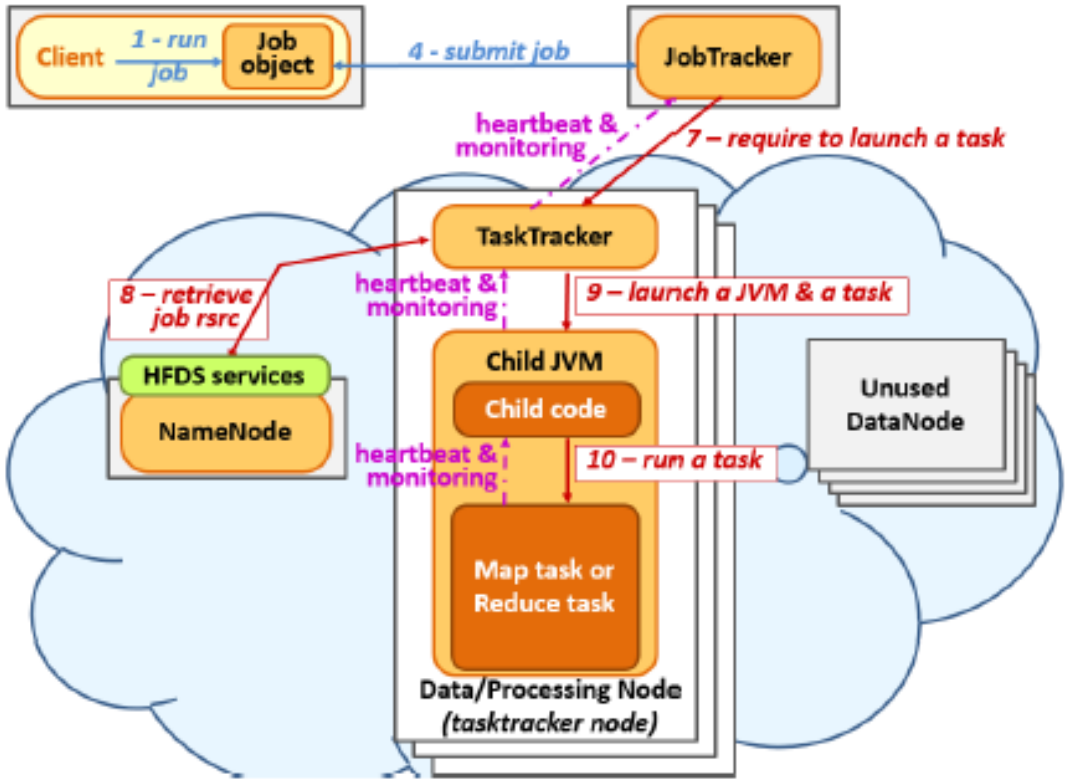
More precisely



Source : S. Vialle

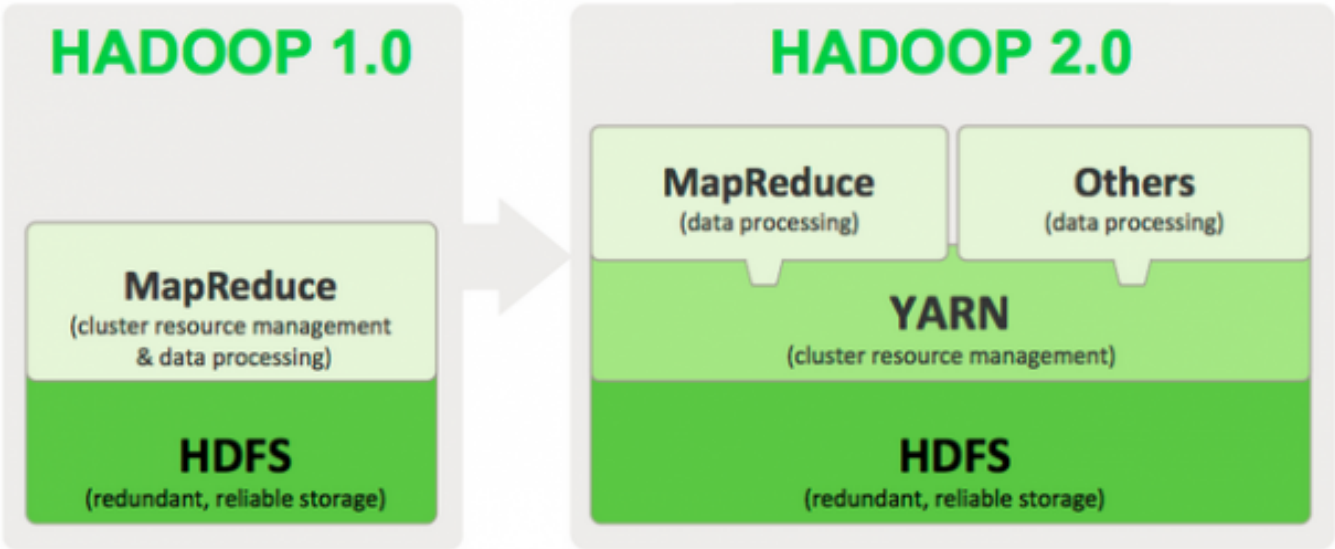
Hadoop MapReduce : anatomy of a job

More precisely



Source : S. Vialle

Hadoop 2.0 : YARN



Hadoop 2.0 : YARN

YARN : Yet another Ressource Negociator

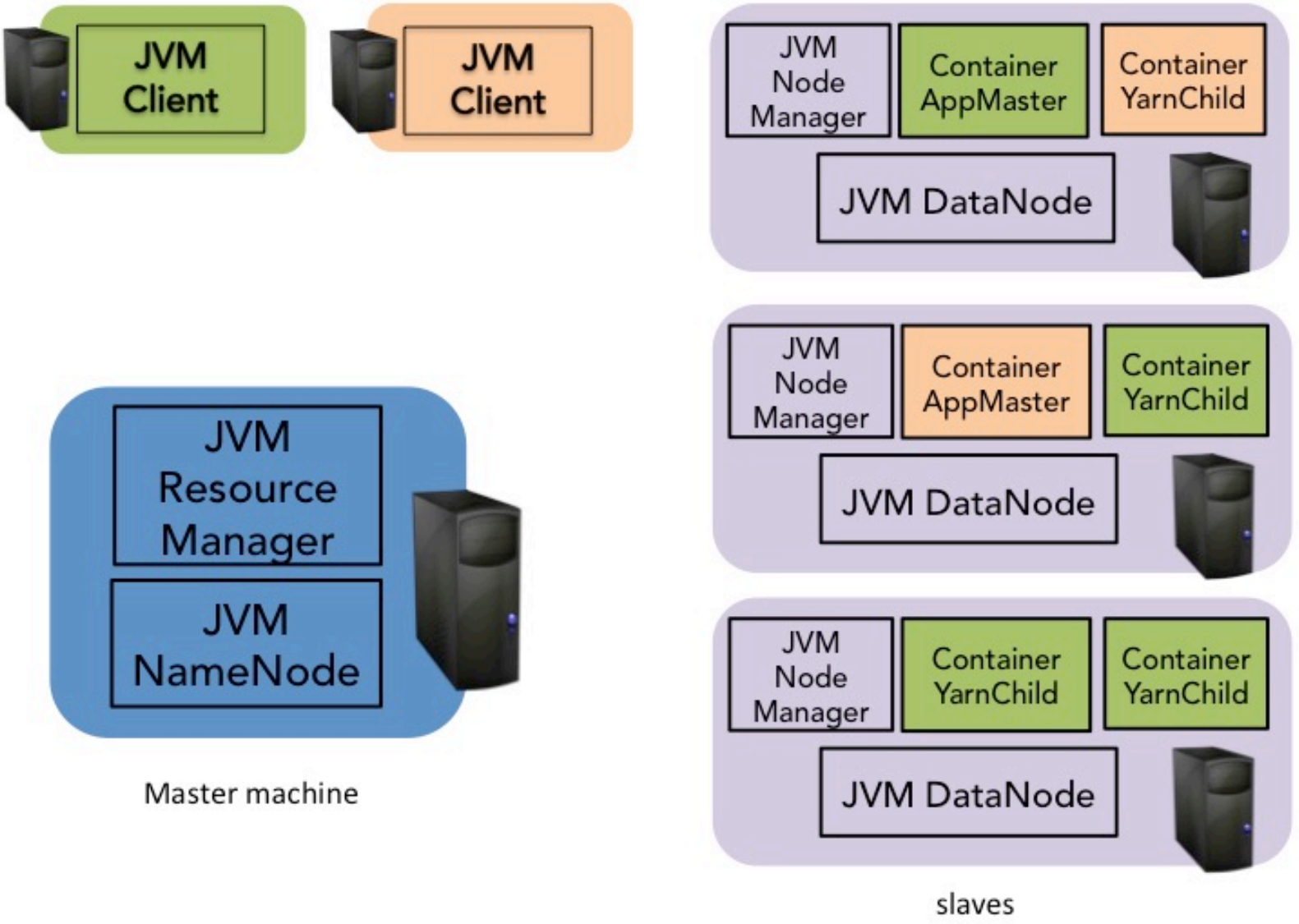
- Enable the execution on the cluster of every type of distributed application (not only MapReduce application).
- Separation of the management of the cluster and of the management of the MapReduce jobs.
- The nodes have some resources that can be allocated to the applications on demand.

Hadoop 2.0 : YARN

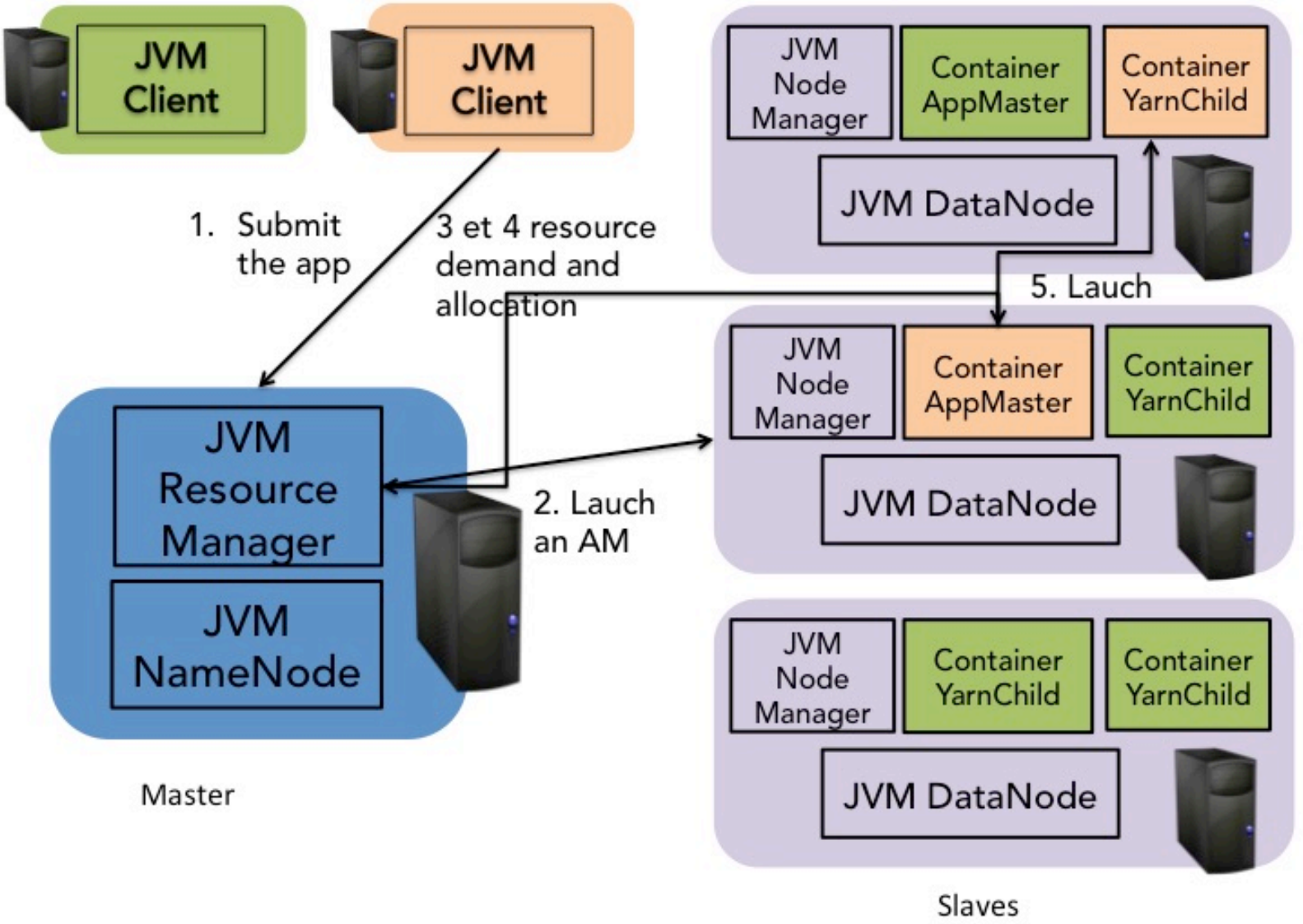
YARN : Yet another Ressource Negociator

- **Resource manager** : orchestration of the resources of the cluster. It schedules the client requests and manage the cluster with the help of **node managers** on each node of the cluster.
- **Application master** : process runned on each slave node and which manages the resources needed for the submitted job.
- **Containers** : resources abstractions that cab be used to execute a map or reduce task or to run an application master.

Hadoop 2.0 : YARN Architecture

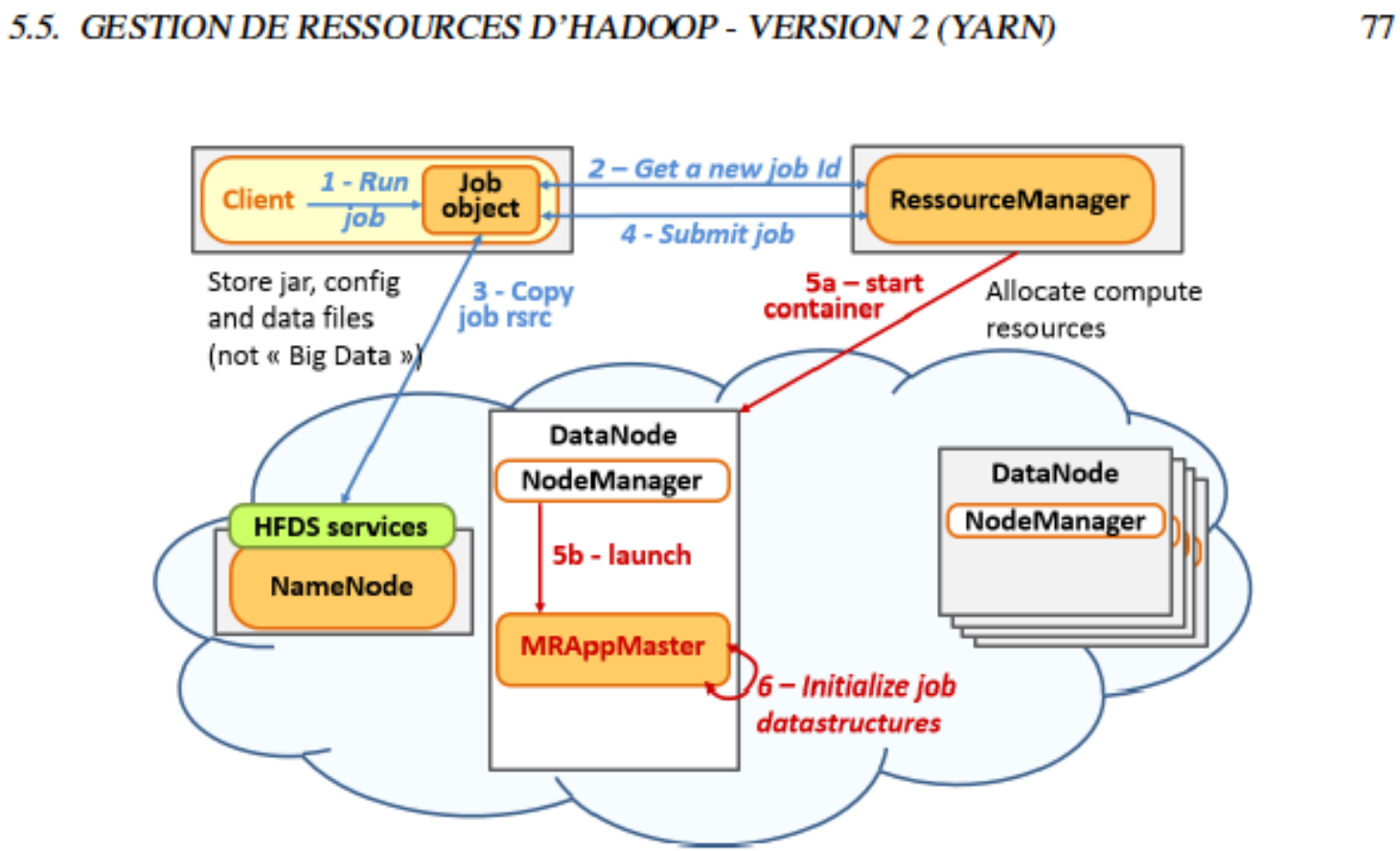


Hadoop 2.0 : YARN Architecture



Hadoop 2.0 : YARN Architecture

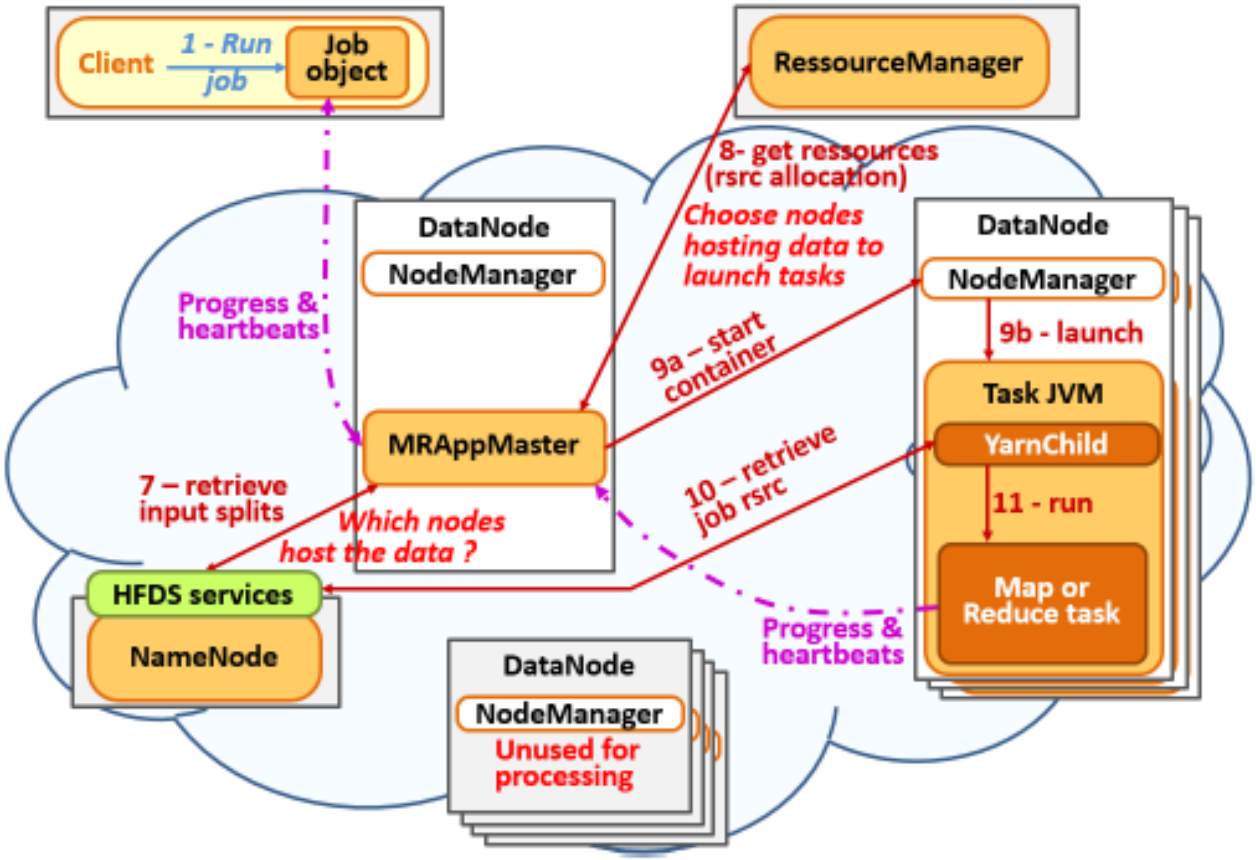
More precisely



Source : S. Vialle

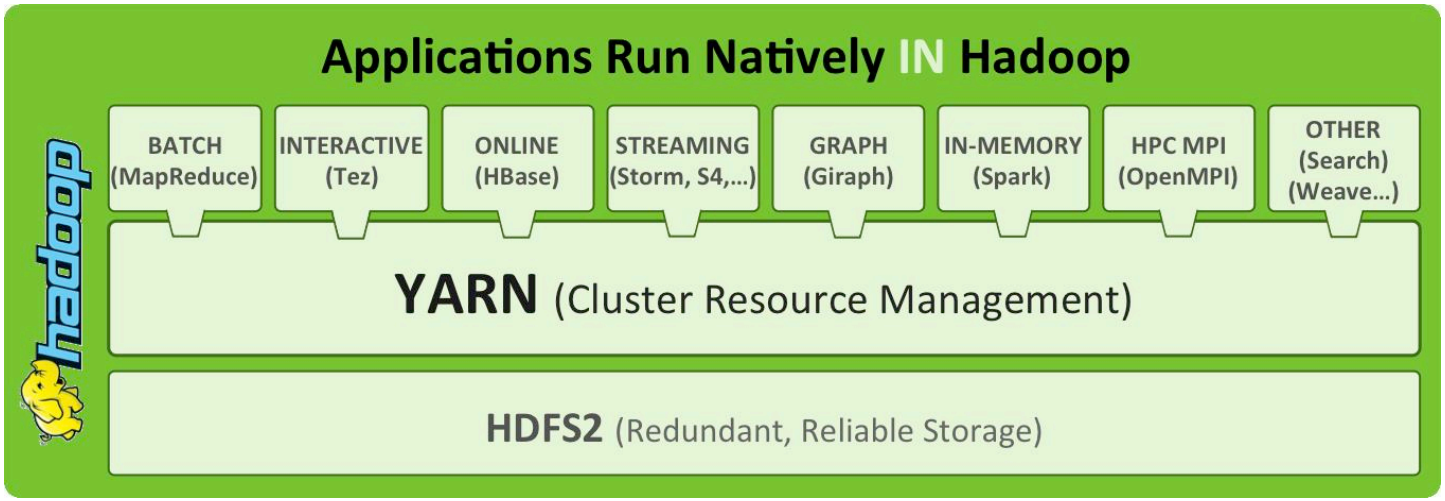
Hadoop 2.0 : YARN Architecture

More precisely



Source : S. Vialle

Hadoop 2.0 : a big ecosystem



Map Reduce in JAVA with Hadoop

A MapReduce job with the JAVA API of Hadoop

- Extend the `Mapper` class of the Hadoop API and override the abstract `map` method.
- Extend the `Reducer` class of the Hadoop API and override the abstract `reduce` method.
- Build a general and main class (often named `Driver`) that builds a job making some references to the two previous classes. This main class extends the `Configured` class of the Haddop API and implements the `Tool`interface.

Map Reduce in JAVA with Hadoop : Map

```
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
import java.io.IOException;

// To complete according to your problem
public class ExempleMap extends Mapper<TypeKeyE, TypeValE,
    TypekeyI, TypeValI> {
// Overriding of the map method
@Override
protected void map(TypeKeyE keyE, TypeValE valE, Context context)
    throws IOException,InterruptedException
    {
        // To complete according to the processing
        // Processing : keyI = ..., valI = ...
        TypeKeyI keyI = new TypeKeyI(...);
        TypeValI valI = new TypeValI(...);
        context.write(keyI,valI);
    }
}
```

Map Reduce in JAVA with Hadoop : Reduce

```
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;
import java.io.IOException;
import java.io.Iterable;
public class ExempleReduce extends
    Reducer<TypeKeyI,TypeValI,TypeKeyS,TypeValS> {
    @Override
    protected void reduce(TypeKeyI cleI, Iterable<TypeValI>
        listevalI, Context context) throws
        IOException,InterruptedException
    {
        // À compléter selon le probleme
        TypeKeyS keyS = new TypeKeyS(...);
        TypeValS valS = new TypeValS(...);
        for (TypeValI val: listevalI) {
            // traitement keyS.set(...), valS.set(...)
        }
        context.write(keyS,valS);
    } }
```

Map Reduce in JAVA with Hadoop : Reduce

```
public class ExempleMapReduce extends Configured implements Tool {

    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.println("Usage: [input] [output]");
            System.exit(-1); }
        // Création d'un job en lui fournissant la configuration et une description
            textuelle de la tâche
        Job job = Job.getInstance(getConf());
        job.setJobName("notre probleme exemple");
        // On précise les classes MyProgram, Map et Reduce
        job.setJarByClass(ExempleMapReduce.class);
        job.setMapperClass(ExampleMap.class);
        job.setReducerClass(ExempleReduce.class);
        // Définition des types clé/valeur de notre problème
        job.setMapOutputKeyClass(TypecleI.class);
        job.setMapOutputValueClass(TypevalI.class);
        job.setOutputKeyClass(TypeCleS.class);
        job.setOutputValueClass(TypeValS.class);
        // Définition des fichiers d'entrée et de sorties (ici considérés comme des
            arguments à préciser lors de l'exécution)
        FileInputFormat.addInputPath(job, new Path(ourArgs[0]));
        FileOutputFormat.setOutputPath(job, new Path(ourArgs[1]));
        //Suppression du fichier de sortie s'il existe déjà
        FileSystem fs = FileSystem.newInstance(getConf());
        if (fs.exists(outputFilePath)) {   fs.delete(outputFilePath, true);  }
        return job.waitForCompletion(true) ? 0: 1;
    }
    public static void main(String[] args) throws Exception {
        ExempleMapReduce exempleDriver = new ExempleMapReduce();
        int res = ToolRunner.run(exempleDriver, args);
        System.exit(res); }}
```

Hadoop Streaming

- Other programming languages than JAVA can be used to write MapReduce programs in Hadoop.
- Hadoop streaming : an utility of the Hadoop distribution that enables the creation and running of Map/Reduce jobs with any executable or script as the mapper and/or the reducer.

Hadoop Streaming

How

```
hadoop jar hadoop-streaming.jar -input [fichier entree HDFS] \
                        -output [fichier sortie HDFS] \
                        -mapper [programme MAP] \
                        -reducer [programme REDUCE]
```

Example in Python

```
hadoop jar hadoop-streaming.jar -input /lejourseleve.txt
                    -output /results
                    -mapper ./WordCountMapper.py
                    -reducer ./WordCountReducer.py
```

Hadoop Streaming

Data formats

- Mapper and Reducer are executables that read the input from stdin (line by line) and emit the output to stdout.
- Data Format :

  `KEY [TABULATION] VALUE`

- A pair by line

Hadoop Streaming : Wordcount in python

Map operation : WordCountMapper.py

```
1  import sys
2  # For each line in entry
3  for line in sys.stdin:
4      # Supprimer les espaces
5      line = line.strip()
6      # get the words and for each word
7      words = line.split()
8      #  map, for each word, generate the pair (word, 1)
9      for word in words:
10         print("{0}\t{1}".format(word,1))
```

Hadoop Streaming : Wordcount in python

Reduce operation : WordCountReducer.py

```
1  import sys
2  curr_count = 0
3  curr_word = None
4  for line in sys.stdin: # process each key value pair from the
       mapper
5      # get the key and the value
6      word, count = line.split('\t',1)
7      count = int(count)
8      # go to the next word (several keys are possible)
9      if word == curr_word :
10         curr_count += count
11     else:
12         if curr_word:
13                 print("{0}\t{1}".format(curr_word,  curr_count)
14     curr_word=word
15         curr_count = count
16   #output the count for the last word
17   if curr_word = word:
18     print("{0}\t{1}".format(curr_word,  curr_count)
```

Others python frameworks

mrjob : a python MapReduce library

https://github.com/Yelp/mrjob

```
1
2  from mrjob.job import MRJob
3  import re
4  WORD_RE = re.compile(r"[\w']+")
5
6  class MRWordFreqCount(MRJob):
7
8      def mapper(self, _, line):
9          for word in WORD_RE.findall(line):
10             yield (word.lower(), 1)
11
12     def combiner(self, word, counts):
13         yield (word, sum(counts))
14
15     def reducer(self, word, counts):
16         yield (word, sum(counts))
17
18 if __name__ == '__main__':
19     MRWordFreqCount.run()
```

Others python frameworks

Pydoop : Python interface to Hadoop

http://crs4.github.io/pydoop/

```
1
2  class Mapper(api.Mapper):
3
4      def map(self, context):
5          for w in context.value.split():
6              context.emit(w, 1)
7
8  class Reducer(api.Reducer):
9
10     def reduce(self, context):
11         context.emit(context.key, sum(context.values))
```