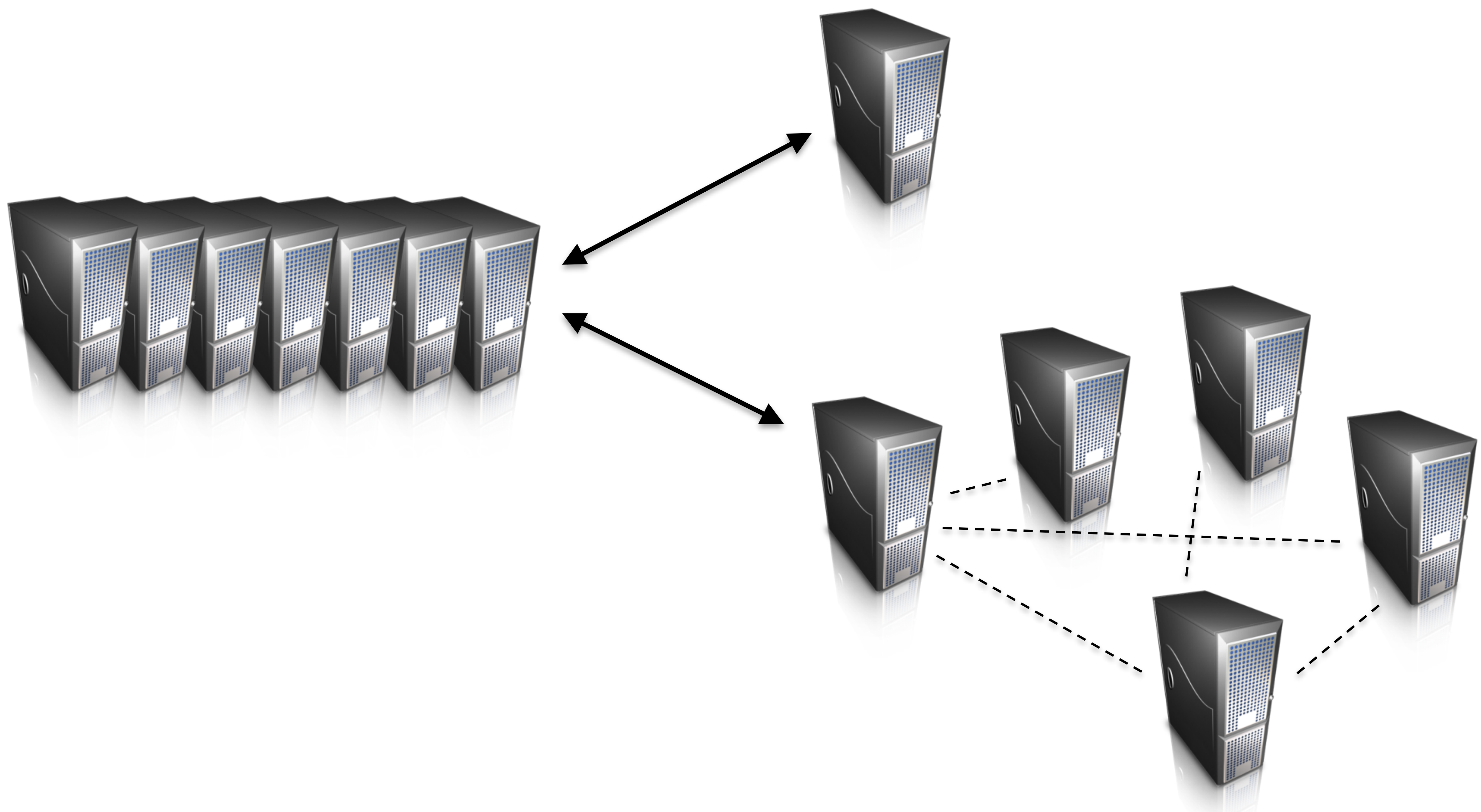
Move data to machines

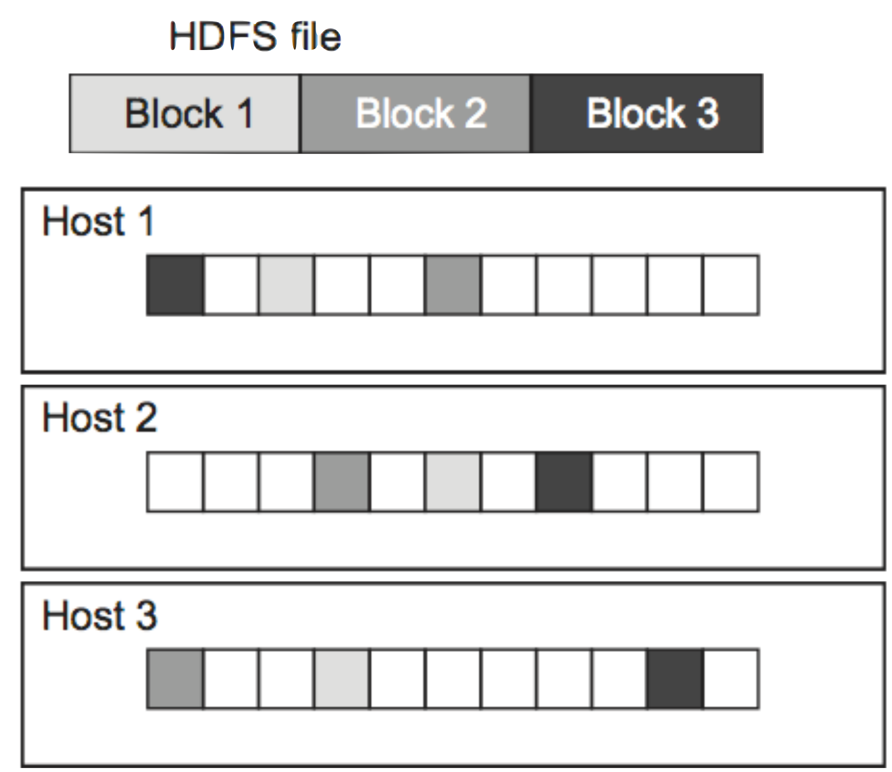


Requirements/Features

- Highly fault-tolerant
  - Failure is the norm rather than exception
- High throughput
  - May consist of thousands of server machines, each storing part of the file system's data.
- Suitable for applications with large data sets
  - Time to read the whole file is more important than the reading the first record
  - Not fit for
    - Low latency data access
    - Lost of small files
    - Multiple writers, arbitrary file modifications
- Streaming access to file system data
- Can be built out of commodity hardware

Blocks

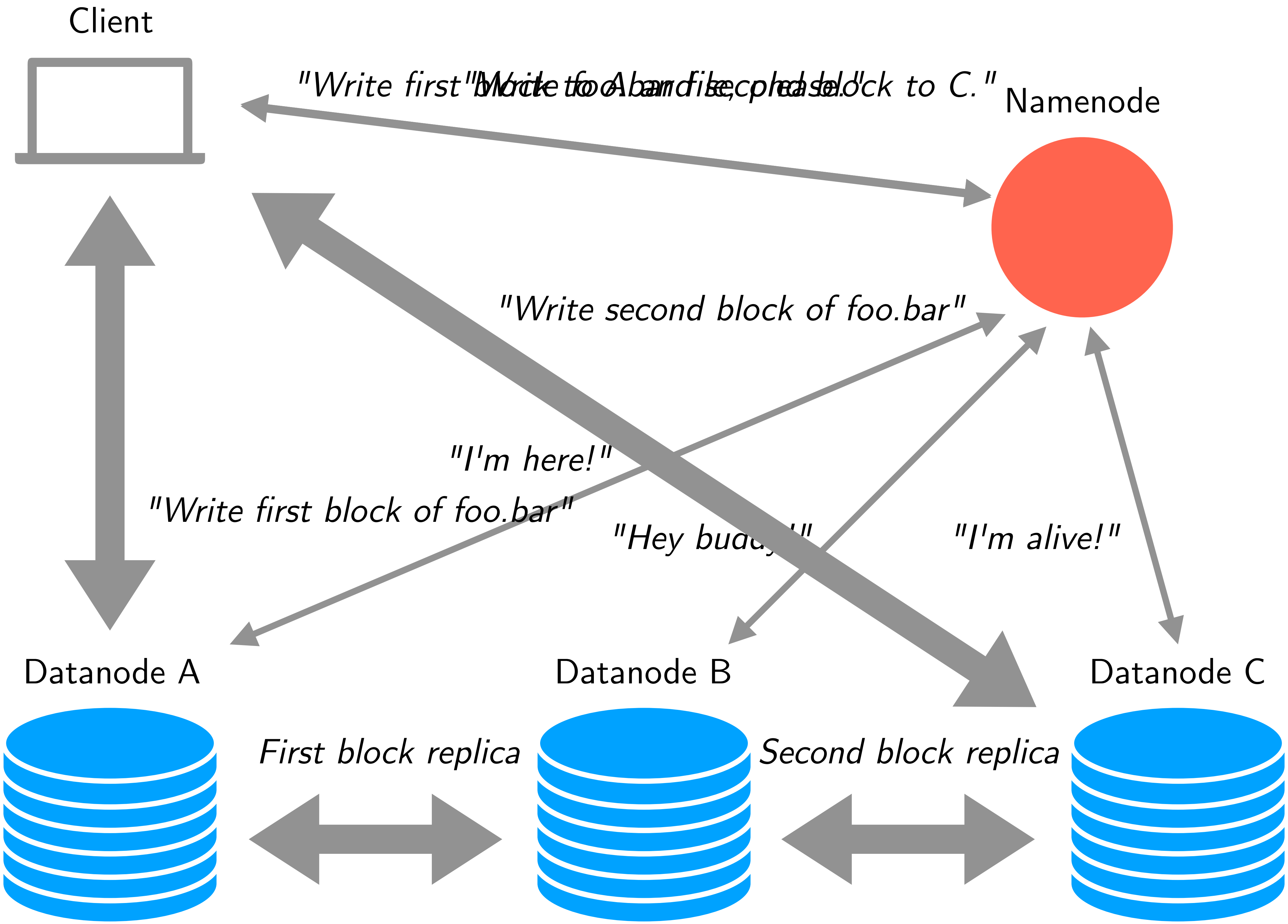
- Minimum amount of data that it can read or write
- File System Blocks are typically few KB
- Disk blocks are normally 512 bytes
- HDFS Block is much larger – 64 MB by default
  - Unlike file system the smaller file does not occupy the full 64MB block size
  - Large to minimize the cost of seeks
  - Time to transfer blocks happens at disk transfer rate
- Block abstractions allows
  - Files can be larges than block
  - Need not be stored on the same disk
  - Simplifies the storage subsystem
  - Fit well for replications
  - Copies can be read transparent to the client



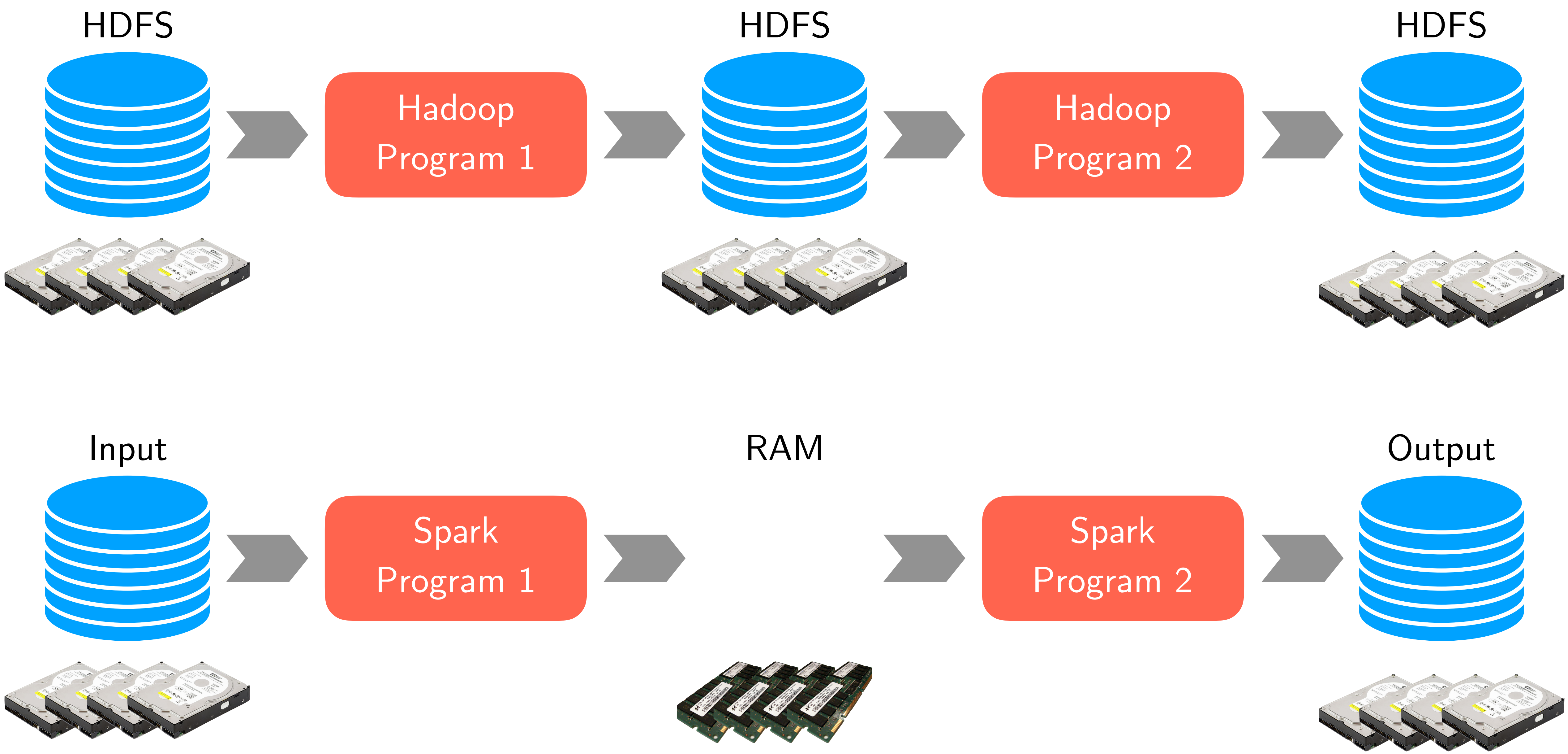
Namenodes & Datanodes

- Master/slave architecture
- DFS cluster consists of a single name node, a master server that manages the file system namespace and regulates access to files by clients.
  - Metadata
  - Directory structure
  - File-to-block mapping
  - Location of blocks
  - Access permissions
- There are a number of data nodes, usually one per node in a cluster.
  - A file is split into one or more blocks and set of blocks are stored in data nodes.
  - The data nodes manage storage attached to the nodes that they run on.
  - Data nodes serve read, write requests, perform block creation, deletion, and replication upon instruction from name node.

HDFS Architecture



Hadoop vs Spark



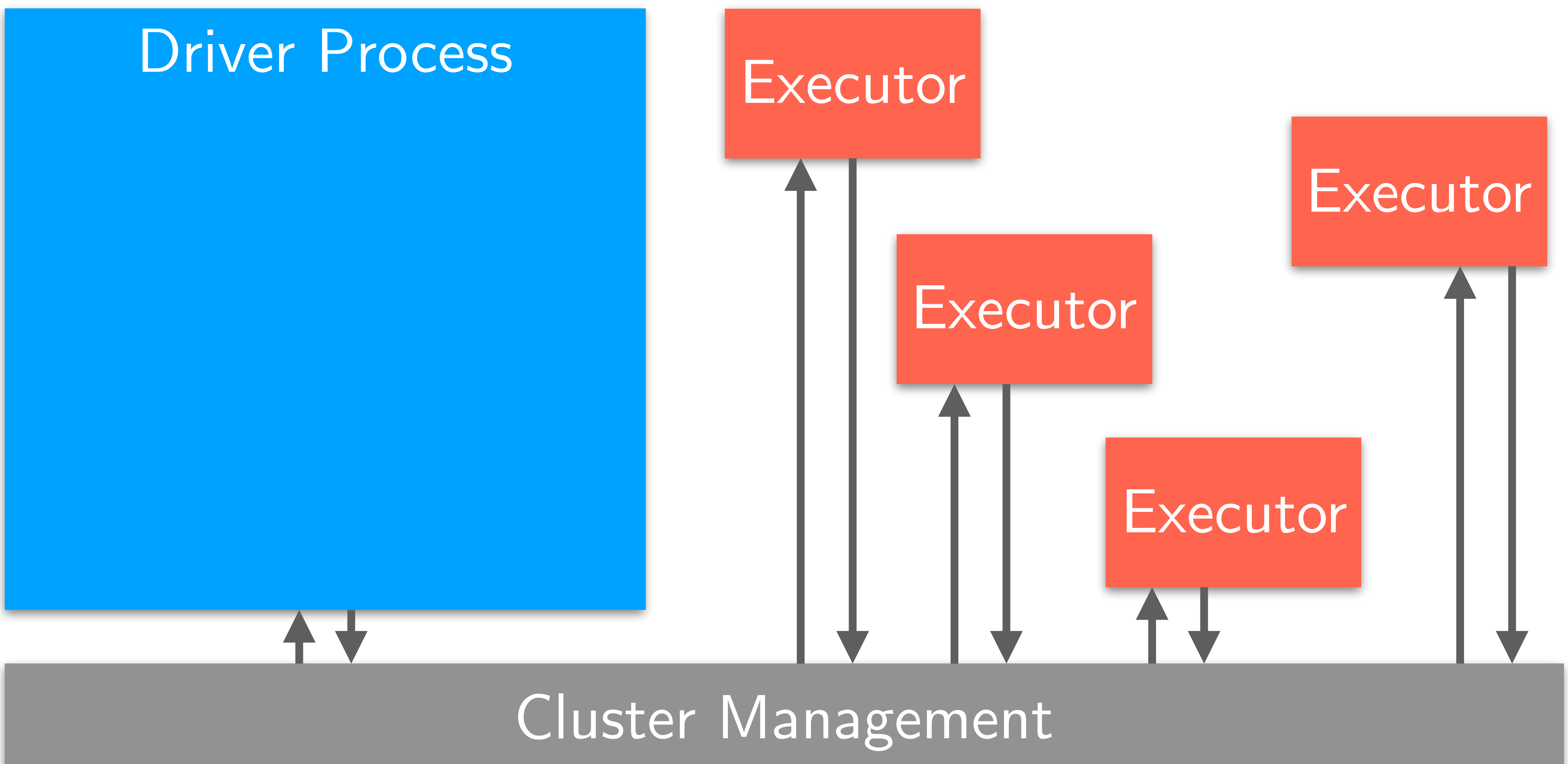
Running Spark

```
$> ssh <student-id>@<host-id>
...
$> pyspark
...
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Python version 3.4.3 (default, Nov 12 2018 22:25:49)
SparkSession available as 'spark'.

In [1]:
```

Some of the following slides are derived from Amir H. Payberah's slides: `https://id2221kth.github.io`

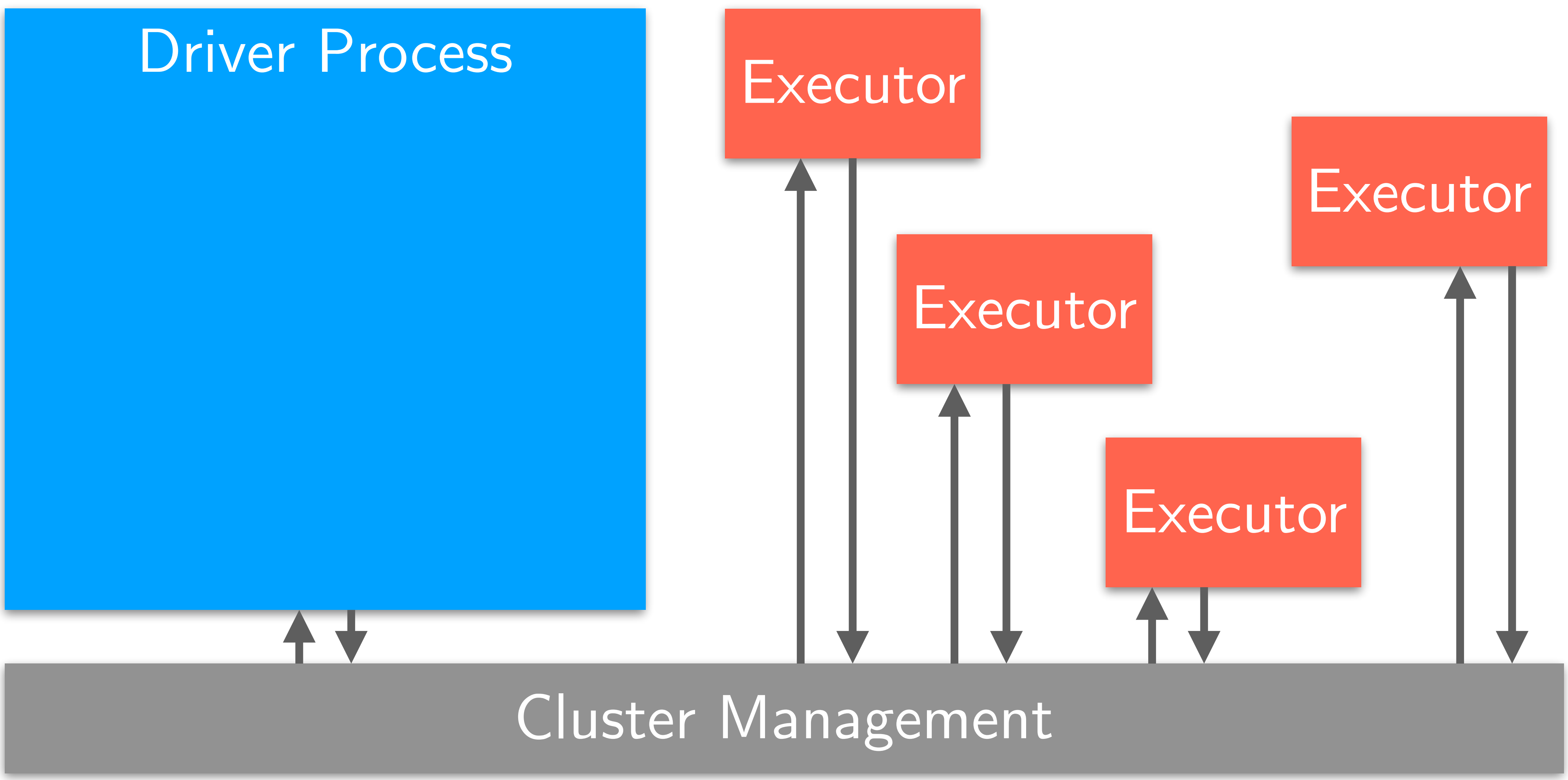
Spark Applications Architecture

- A **Spark application** consists of
  - a **driver** process
  - a **set of executor** processes



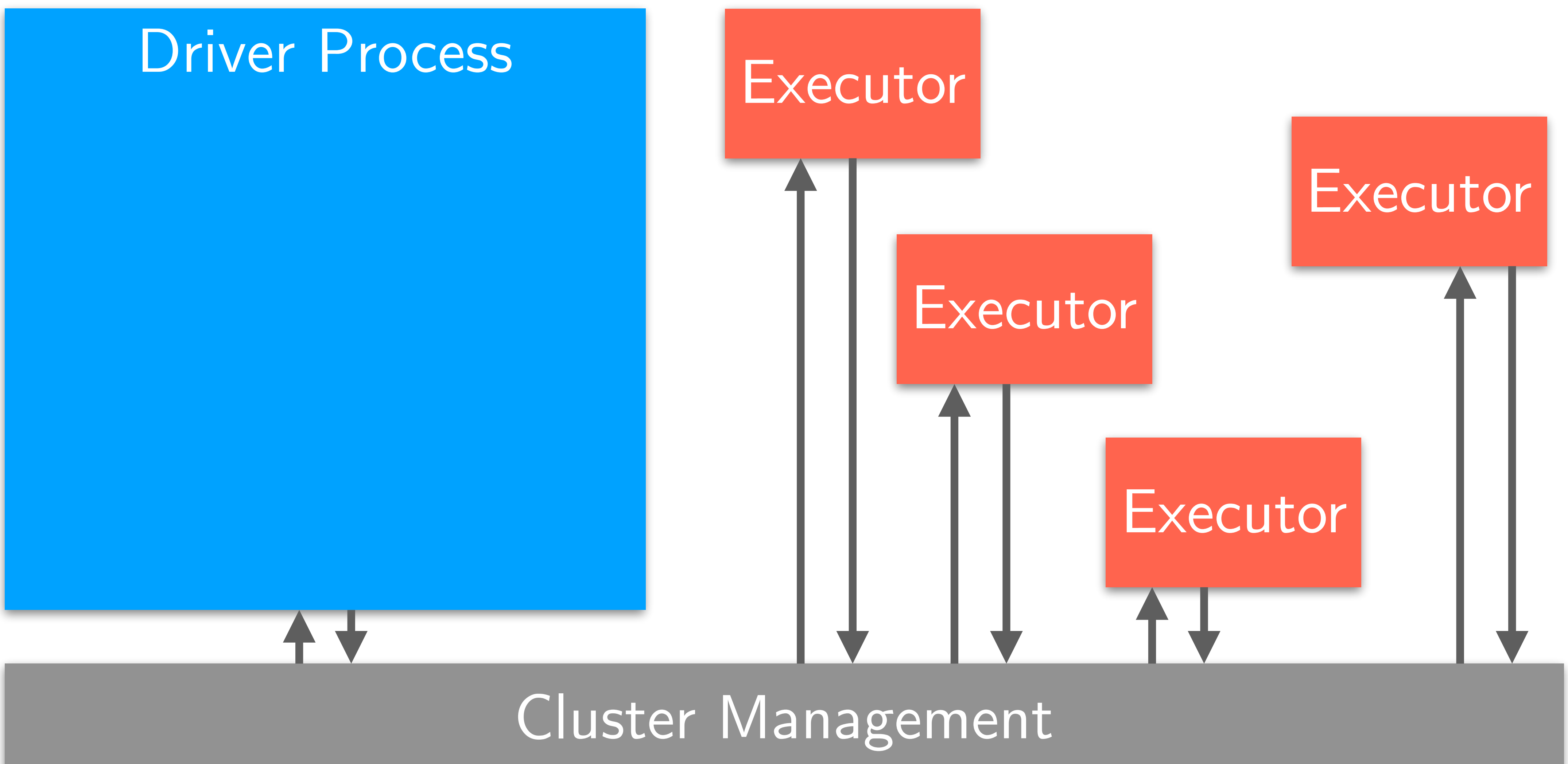
Spark Driver

- The driver process is
  - the heart of a Spark application
  - runs in a node of the cluster
  - runs the `main()` function



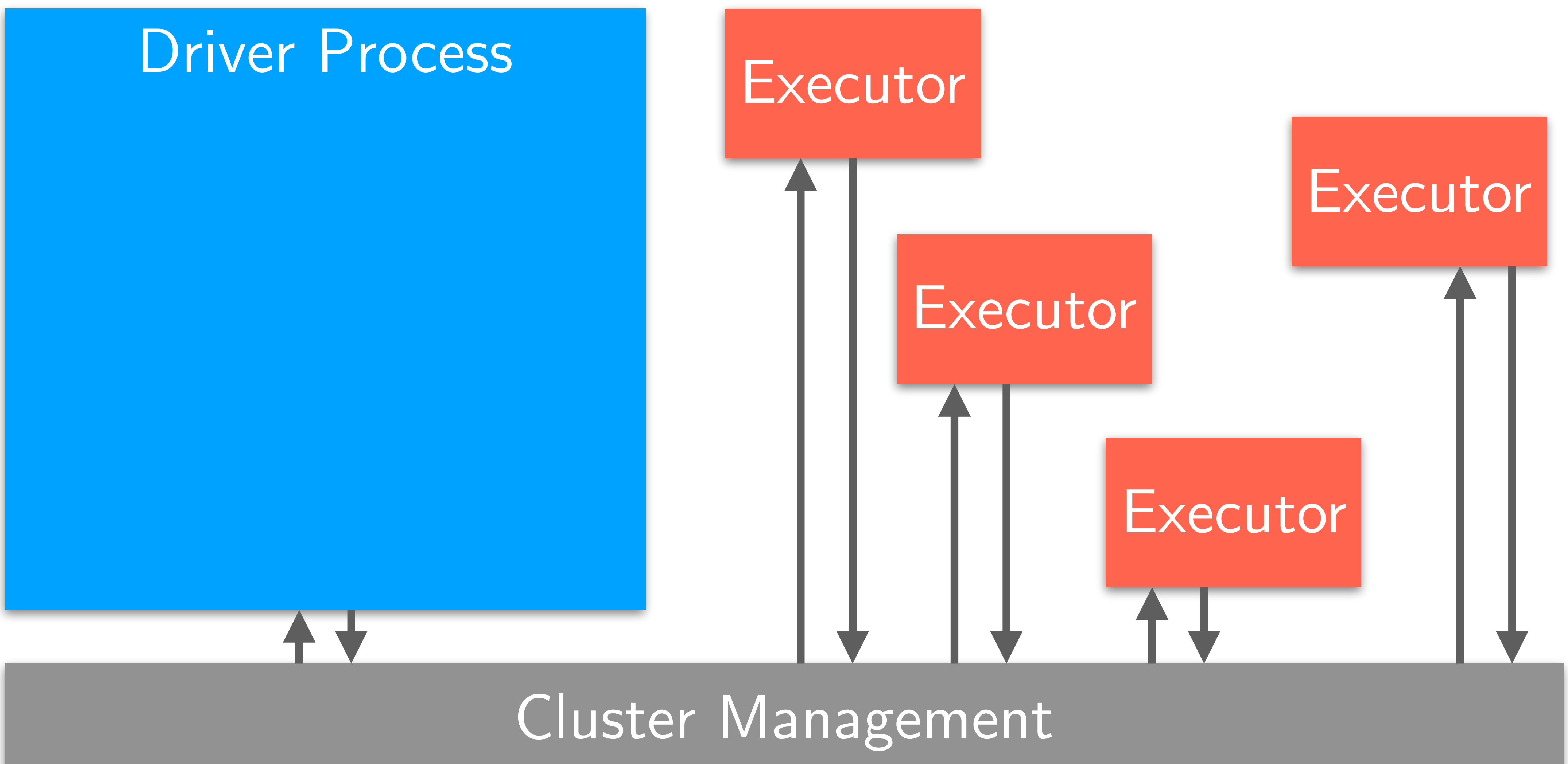
Spark Driver

- Responsible for three things:
  1. **Maintaining information** about the Spark application
  2. **Interacting** with the user
  3. **Analyzing, distributing and scheduling** work across the executors



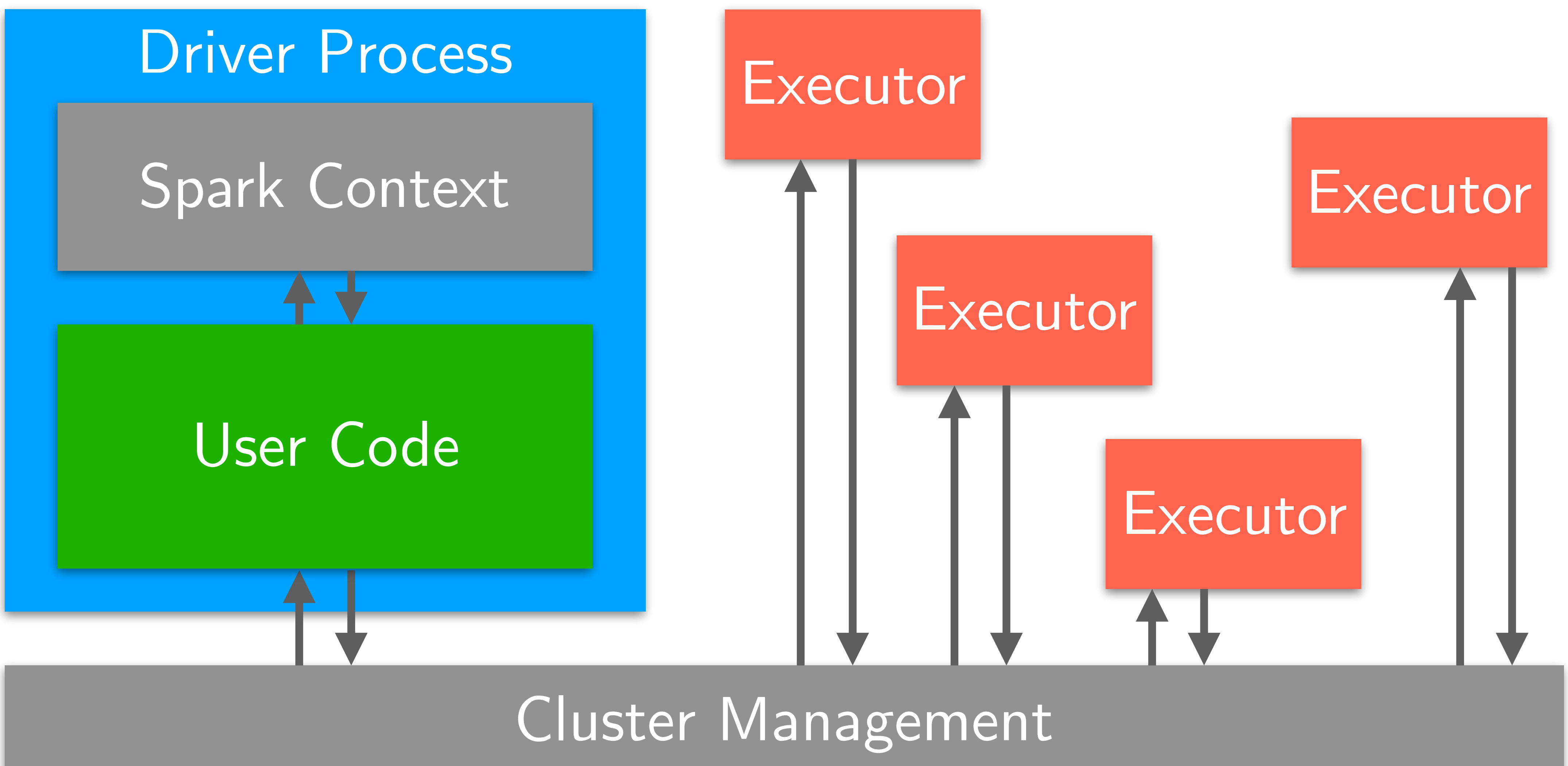
Spark Executors

- Responsible for two things:
  1. **Executing code** assigned to it by the driver
  2. **Reporting the state** of the computation on that executor back to the driver



Spark Context

- The driver process is composed by:
  - A **spark context**
  - A **user code**



Spark Context

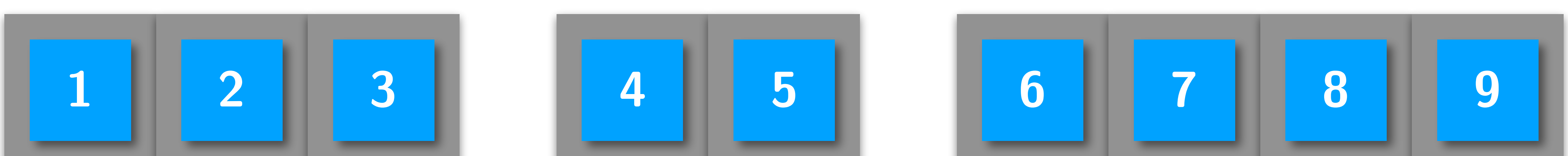
- The `SparkContext` object represents a connection with the cluster system.
- In the `pyspark` shell
  - a `SparkContext` is created automatically on start
  - It is accessible through the variable `sc`
- In a Python script including a Spark application you need to create it as soon as necessary

```
# import spark
from pyspark import SparkContext

# initialize a new Spark Context to use for the execution of the script
sc = SparkContext(appName="MY-APP-NAME", master="local[*]")
```

RDD

- A **resilient distributed dataset** (RDD) is a distributed memory abstraction
- Immutable collection of objects spread across the cluster

- An RDD is divided into a number of **partitions**, which are atomic pieces of information
- Partitions of an RDD can be stored on different nodes of a cluster



Creating an RDD

- Use the parallelize method on a SparkContext object sc
- Turns a single node collection into a parallel collection.
- You can also explicitly state the number of partitions.

```
In [1]: numbers = [1,2,3,4,5]
In [2]: rdd_numbers = sc.parallelize(numbers)
In [3]: print(rdd_numbers)

In [4]: words = "nel mezzo del cammin di nostra vita".split(" ")
In [5]: rdd_words = sc.parallelize(words,2)
In [6]: print(rdd_words)
```

Creating an RDD

- RDDs can be created from external storage
  - Local disk, HDFS, Amazon S3, ...
- Text file RDDs can be created using the **`textFile()`** method

```
In [1]: # rdd_file = sc.textFile("file.txt")
In [2]: # rdd_hdfs = sc.textFile("hdfs://namenode:9000/path/to/file")
In [3]: shakespeare_rdd = sc.textFile("hdfs://masterbig-1.itc.unipi.it:
54310/masterbig_data/shakespeare.txt")
```

RDD Operations

- RDDs support **two types** of operations:
    1. **Transformations**: allow us to build the logical plan
    2. **Actions**: allow us to trigger the computation
- **Transformations** create a **new RDD** from an **existing RDD**.
    - **Not compute** their results right away (**lazy**).
    - **Remember** the transformations applied to the base dataset.
    - They are only computed when an action requires a result to be returned to the driver program.
- **Actions** **trigger** the computation.
    - Instruct Spark to **compute a result** from a series of transformations.
    - There are **three** kinds of actions:
        - Actions to **view data** in the console
        - Actions to **collect data** to native objects in the respective language
        - Actions to **write to output** data sources

RDD Actions

- `collect` returns all the elements of the RDD as an **array** at the driver
- `first` returns the first **value** in the RDD
- `take` returns an **array** with the first ***n*** elements of the RDD
  - Variations on this function: `takeOrdered` and `takeSample`
- `count` returns the **number** of elements in the dataset
- `max` and `min` return the **maximum** and **minimum** values, respectively.
- `reduce` aggregates the elements of the dataset using a given **function**.
  - The given function should be **commutative** and **associative** so that it can be computed correctly in parallel.
- `saveAsTextFile` writes the elements of an RDD as a **text file**.
  - Local filesystem, HDFS or any other Hadoop-supported file system.

RDD Actions Examples

```
In [1]: numbers = sc.parallelize([1, 2, 2, 2, 1, 1, 4, 3, 3, 5, 5])
In [2]: numbers.collect() # triggers execution on ALL elements, takes time
# list [1, 2, 2, 2, 1, 1, 4, 3, 3, 5, 5]
In [3]: numbers.first()
# int 1
In [4]: numbers.take(4) # triggers execution on 4 elements, good for debug
# list [1, 2, 2, 2]
In [5]: numbers.takeOrdered(4)
# list [1, 1, 1, 2]
In [6]: numbers.takeOrdered(4)
# list [1, 1, 1, 2]
In [7]: withReplacement = True
In [8]: numberToTake = 4
In [9]: randomSeed = 123456
In [10]: numbers.takeSample(withReplacement, numberToTake, randomSeed)
# list [1, 5, 2, 5]
```

RDD Actions Examples

```
In [1]: numbers = sc.parallelize([1, 2, 2, 2, 1, 1, 4, 3, 3, 5, 5])
In [2]: numbers.count()
# int 11
In [3]: numbers.countByValue()
# defaultdict(int, {1: 3, 2: 3, 4: 1, 3: 2, 5: 2})
In [4]: numbers.max()
# int 5
In [5]: numbers.min()
# int 1
In [6]: numbers.reduce(lambda x, y: x + y)
# int 29
In [7]: numbers.saveAsTextFile('numbers.txt')
# exit pyspark check contents of file 'numbers.txt'
# ls -ltrh snumbers.txt
```

Generic RDD Transformations

- distinct removes duplicates from the RDD
- filter returns the RDD records that match some **predicate function**

```
In [1]: numbers = sc.parallelize([1,2,2,2,3,3,4,5,5,5,5])
In [2]: distinct_numbers = numbers.distinct()
In [3]: print(distinct_numbers.collect()) # this is an action
[2, 4, 1, 3, 5]
In [4]: even_numbers = distinct_numbers.filter(lambda x: x % 2 == 0)
In [5]: print(even_numbers.collect()) # this is an action
[2, 4]
```

- sample draws a random sample of the data, with or without replacement

```
In [1]: data = sc.parallelize(range(20))
In [2]: sampled_data = data.sample(withReplacement = False, fraction = 0.20)
In [3]: print(sampled_data.collect()) # this is an action
[5, 13, 14, 16]
```

Generic RDD Transformations

- `map` and `flatMap` apply a given function to each RDD element **independently**
- `map` transforms an RDD of **length *n*** into another RDD of **length *n***.
- `flatMap` allows returning **0, 1 or more elements** from map function.

```
In [1]: data = sc.parallelize(range(10))
In [2]: squared_data = data.map(lambda x: x * x)
In [3]: print(squared_data.collect()) # this is an action
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [4]: squared_cubed_data_1 = data.map(lambda x: ( x * x, x * x * x))
In [5]: print(squared_cubed_data_1.collect()) # this is an action
[(0, 0), (1, 1), (4, 8), (9, 27), (16, 64), (25, 125), (36, 216), (49, 343),
(64, 512), (81, 729)]

In [6]: squared_cubed_data_2 = data.flatMap(lambda x: ( x * x, x * x * x))
In [7]: print(squared_cubed_data_2.collect()) # this is an action
[0, 0, 1, 1, 4, 8, 9, 27, 16, 64, 25, 125, 36, 216, 49, 343, 64, 512, 81,
729]
```