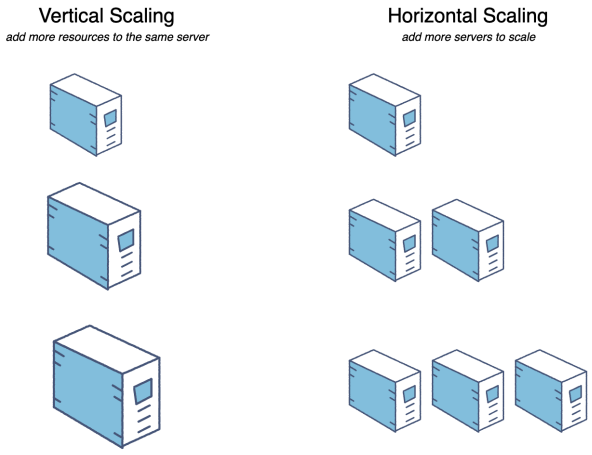
Why do we need distributed systems?

1. **Scalability**: Distributed systems allow us to scale up the resources of a system by adding more computers to the network. This allows us to handle more users and more data without overloading a single computer.

2. **Fault tolerance**: Distributed systems can provide high availability and fault tolerance by replicating data and services across multiple computers. If one computer fails, the system can continue to function without interruption.

3. **Performance**: Distributed systems can improve performance by distributing computation across multiple computers. This allows us to perform complex computations more quickly and handle larger workloads.

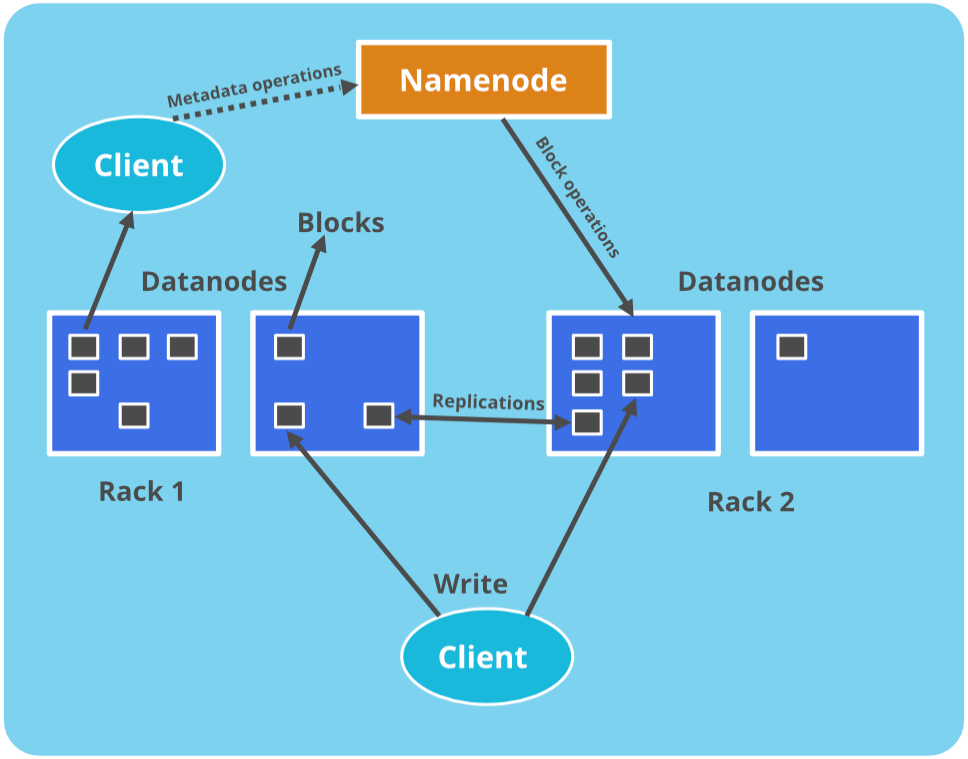
4. **Geographical distribution**: Distributed systems can allow users to access services from anywhere in the world. By placing computers in different locations, we can reduce network latency and provide better service to users in different parts of the world.
5. **Cost-effectiveness**: Distributed systems can be more cost-effective than centralized systems because they can use commodity hardware and can scale up or down as needed



Scalability is the biggest benefit of distributed systems. Horizontal scaling means adding more servers into your pool of resources. Vertical scaling means scaling by adding more power (CPU, RAM, Storage, etc.) to your existing servers.

Architecture of HDFS:

Hadoop 1.x Architecture



The Hadoop Distributed File System (HDFS) architecture consists of two key components: the **NameNod**e and the **DataNode**. The NameNode manages the file system metadata, while the DataNode stores the actual data.

HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes.

The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes.

The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.

The existence of a **single NameNod**e in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata.

Secondary NameNode:
When NameNode runs out of disk space, a secondary NameNode is activated to perform a checkpoint.

DataNode
Every slave machine that contains data organizes a DataNode.DataNodes do the following:

- DataNodes store every data.
- It handles all of the requested operations on files, such as reading file content and creating new data

Hadoop 2.x Architecture

Problem: As you know in Hadoop 1.x architecture Name Node was a single point of failure, which means if your Name Node daemon is down somehow, you don't have access to your Hadoop Cluster. How to deal with this problem?

Solution: Hadoop 2.x is featured with Name Node HA which is referred as HDFS High Availability (HA).

- Hadoop 2.x supports two Name Nodes at a time one node is active and another is standby node
- Active Name Node handles the client operations in the cluster
- StandBy Name Node manages metadata same as Secondary Name Node in Hadoop 1.x
- When Active Name Node is down, Standby Name Node takes over and will handle the client operations then after
- HDFS HA can be configured by two ways
    a. Using Shared NFS Directory
    b. Using Quorum Journal Manager

Now, how the synchronization between Active Namenode and Standby Namenode is going to happen. Who will tell the Standby namenode that now you are an active node and who will tell the data node to send the message to the latest active namenode?

There free tool available which can manage all that synchronization:

1. **Zookeeper:**

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.

It'll also help to elect the leader so inside the zookeeper there is an algorithm which will elect the master node. You have to provide the IP address of all the namenodes present in your cluster to the zookeeper.

How does the Zookeeper elect the master and maintain high availability?

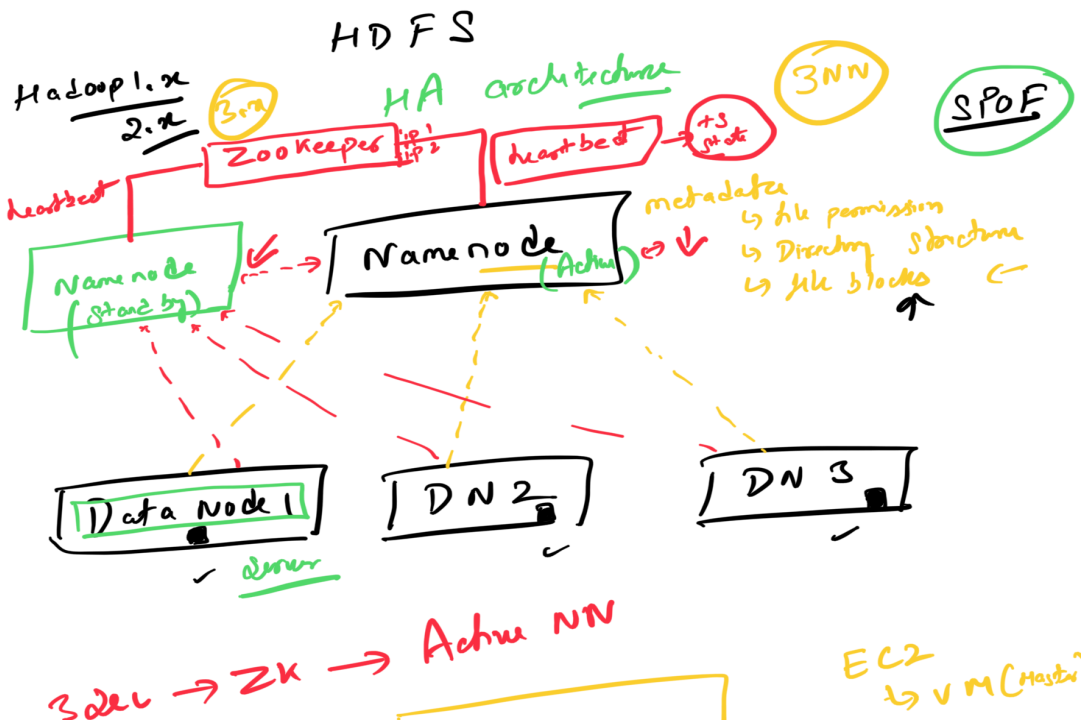
It does this through a simple feedback loop. The Zookeeper, through its Fail-over Controller, monitors the Leader and Follower/Stand-by nodes.

It receives a heartbeat or instant notification of the current health/status of each node. The moment a leader fails, one of the stand-by nodes is elected as the new leader by Zookeeper almost immediately. The election is completed and the new leader sends a message to the data nodes to report to the new master node.

Heartbeat Mechanism:

By default every 3 seconds, zookeeper will send a message to the active node and will do it 10 times by default. If the current Active node is not responding then Zookeeper is going to mark that node as a down.

It'll then send the message to the standby node that you are now the Active node.

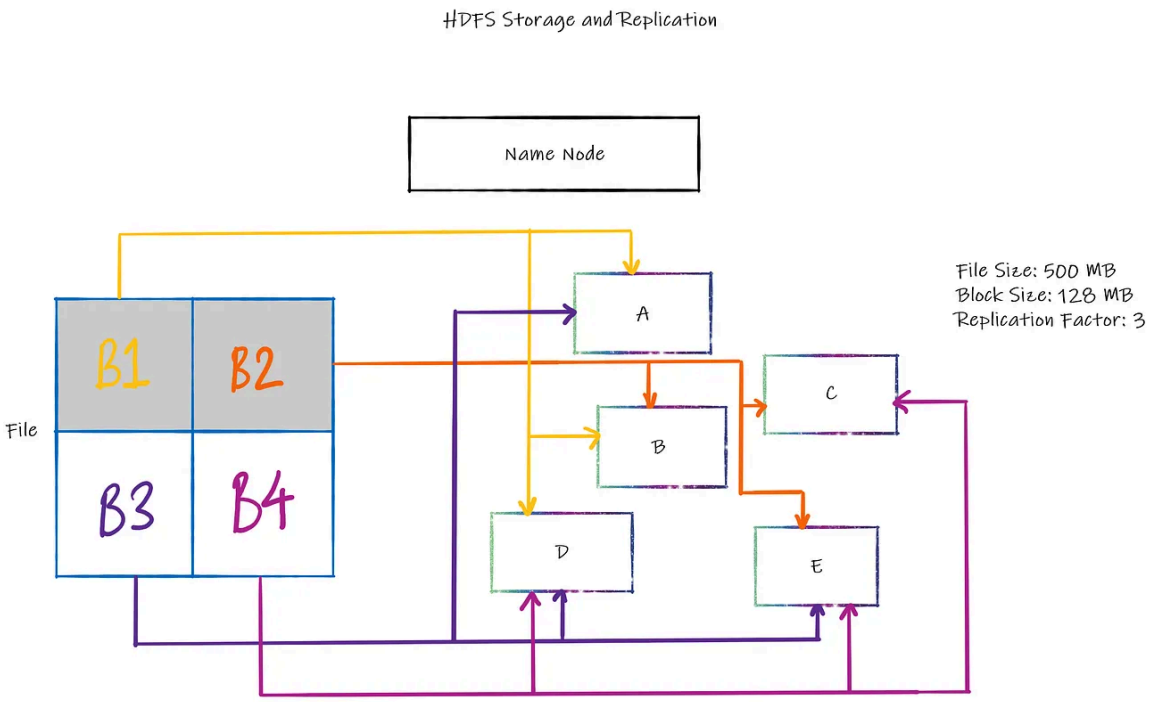


Replication Factor:

HDFS is a fault-tolerant and resilient system,In order to achieve this, data stored in HDFS is automatically replicated across different nodes.

How many copies are made? This depends on the “replication factor”. By default, it is set to 3 i.e. 1 original and 2 copies.

Storage and Replication Architecture:

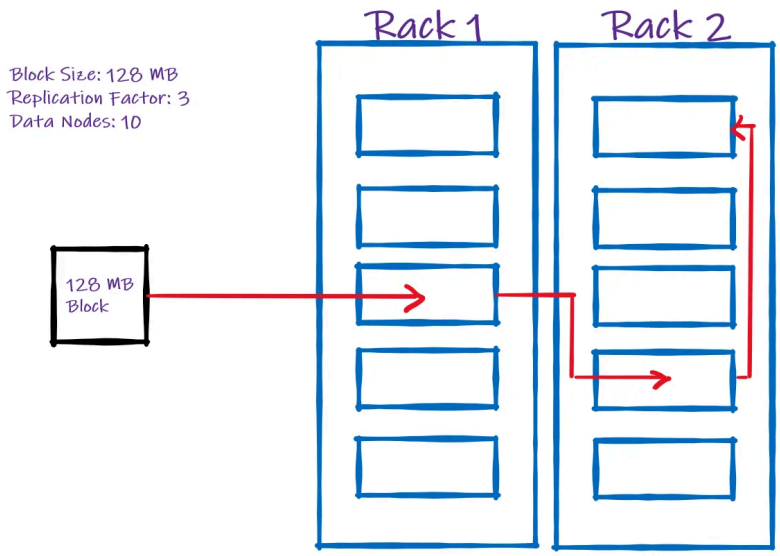


With HDFS' default block size of 128 MB, this file is broken into 4 blocks B1 — B4. Please note that A — E are our Data Nodes. With HDFS' default replication factor of 3, the blocks are replicated across our 5 node cluster. Block B1 (in yellow) is replicated across Nodes A, B and D and so on and so forth (follow the coloured lines).

Here, the Name Node maintains the metadata, i.e. data about data. Which replica of which block of which file is stored in which node is maintained in NN — replica 2 of block B1 of file xyz.csv is stored in node B.

So a file of size 500 MB requires a total storage capacity in HDFS of 1500 MB due to its replication. This is abstracted from the end users' perspective and the user can only see 1 file of size 500 MB stored within HDFS.

The Block Replication Algorithm



The algorithm starts by searching for the topology.map file under HDFS' default configuration folder. This .map file contains metadata information on all the available racks and nodes it contains. In our example case in the image above, we have 2 racks and 10 data nodes.

Once the file is divided into blocks, the first copy of the first block is inserted into the rack and data node which is nearest to the client

The copy of this first block is created and moved onto the next available rack i.e. Rack 2 and stored in any available data node.

Another copy is created here and moved onto the next available rack

this is how Replication algorithm works

YARN (Yet Another Resource Negotiator):

Hadoop 1.x consist of only 2 components

- **Mapreduce**: which is used for processing and resource management
- **HDFS**: for storage

Mapreduce started giving issues:

1) It was only used for doing batch processing. But as technology progressed everybody started progressing on real-time data rather than batch processing but Mapreduce was not able to process real-time data.

They solved this issue in Hadoop 2.x by introducing YARN

Hadoop developed tool which can perform resource management separately from Mapreduce so Hadoop 2.x have 3 components:

1) Mapreduce
2) YARN
3) HDFS

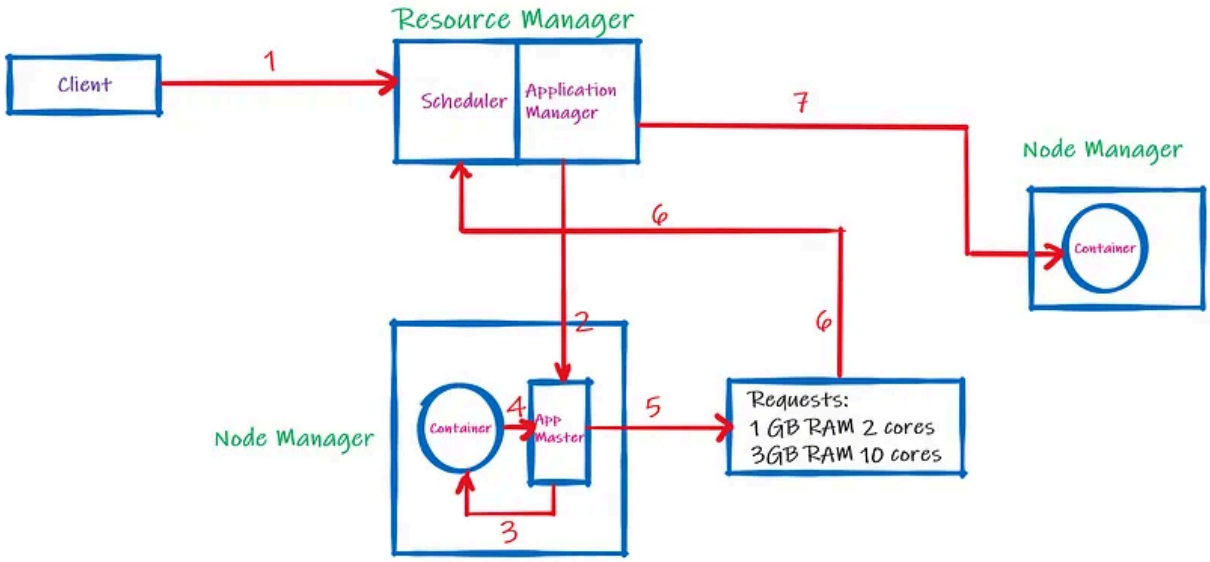
What Exactly is a YARN?

YARN or “Yet Another Resource Negotiator” does exactly as its name says, it negotiates for resources to run a job.

YARN, just like any other Hadoop application, follows a “**Master-Slave**” architecture, wherein the **Resource Manager is the master** and the **Node Manager is the slave**.

The master allocates jobs and resources to the slave and monitors the cycle as a whole. The slave receives the job and requests (additional) resources to complete the job and actually undertakes the execution of the job.

Working of YARN and it’s Architecture:



1. The Client sends a job (jar file) to the Resource Manager (RM).

2. The RM contains two parts, namely, **Scheduler** and **Application Manager** (AM). The Scheduler receives the job request and requests the AM to search for available Node Managers (NM). The selected NM spawns the Application Master (App Master).

   Please note, the RM Scheduler only schedules the job. It cannot monitor or restart a failed job. The AM monitors the end-to-end life cycle of the job and can reallocate resources if a NM fails. It can also restart a failed job in App Master.

3. The App Master checks the resources provided for the job in the container. The job now resides in the App Master. Do note, that it communicates the status of the job to AM.

4. It is important to note that the Yarn Container can only be used when a Container Launch Context (CLC) certificate is provided by the App Master. It works like a key to unlock the container's resources. This is internal to YARN. The App Master job is now executed in the container. However, if the resources provided are not sufficient then

5. the App Master creates a list of requests and

6. This list is sent directly to the RM Scheduler. RM again requests the AM for more resources.

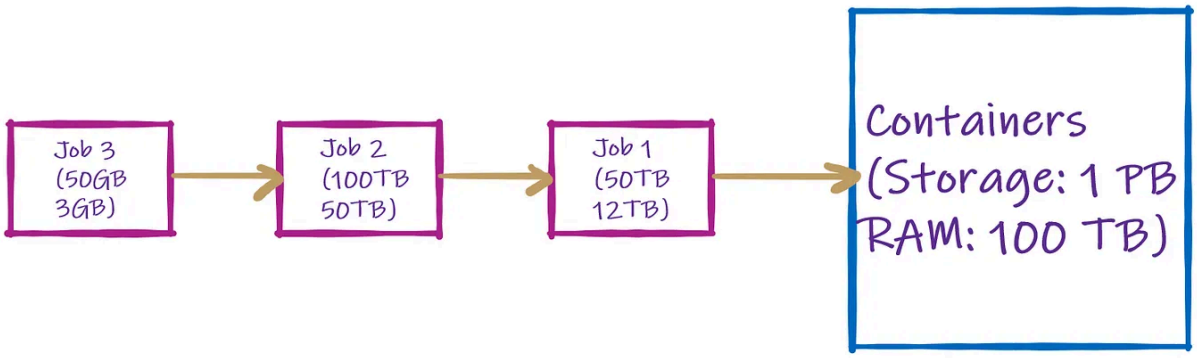
7. A new container is launched through a new NM without an App Master. The job is successfully executed and the resources are released.

Scheduler has an internal policy on How to give resources to the job?

YARN Schedulers:

1) **FIFO Scheduling**
2) **Capacity Scheduling**
3) **Fair Scheduling**

FIFO Scheduling:



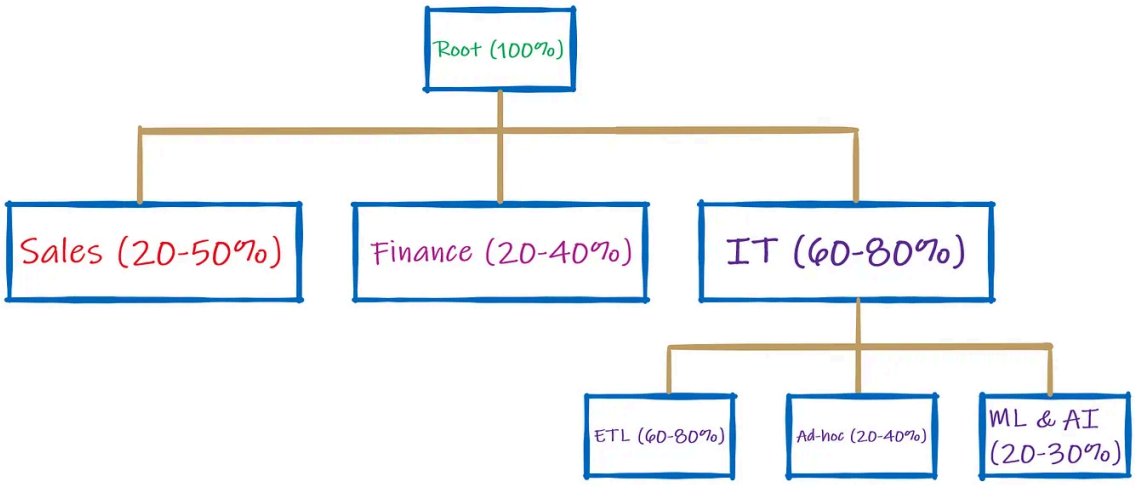
As the name suggests, First in First out or FIFO is the most basic scheduling method provided in YARN.

FIFO places jobs submitted by the client in queues and executes them in a sequential manner on a first-come-first-serve basis.

Although we could run multiple jobs together, in FIFO, they will run sequentially. Such is the waste. Hence, this scheduling methodology is not preferred on a Production/Shared Cluster as it suffers from poor resource utilization.

Capacity Scheduler was introduced to maximize utilization.

Capacity Scheduling:



The central idea is that multiple departments fund the central cluster or “root” denoted as 100% of available resources as seen on top of the above chart. Each department or “leaf” is guaranteed a specific range of capacity to carry out its jobs whenever required without waiting.

Basically, it’ll provide the resources to each job according to its requirement plus it’ll also provide additional runtime if some job is taking more time, like here IT department.

The priority functionality, depending on a case-to-case basis, is a drawback of Capacity Scheduling.

Fair Scheduling:

You need to understand how fair and capacity scheduling works, as Fair scheduling builds upon its drawbacks.

If a single job is run, Fair Scheduling (FS) will throw all its resources at it. If another job is added, it will ensure that a “fair” amount of resources are added to finish that job too.

The main aim is that all jobs receive a “fair” amount of resources over-time to execute successfully. This is especially true when there are 2 jobs and one of them is small. The small job also receives an adequate amount of resources to execute instead of being put on hold until the big job completes