Università degli Studi di Padova

J. Pazzini
PADOVA UNIVERSITY, INFN

4 - FILE SYSTEMS

Management and Analysis of Physics Datasets - Module B

Physics of Data

A.A. 2023/2024

HOW TO STORE DATA

Three main solutions to store/manage our datasets:

Files → Blocks of data stored in volumes managed by a File System
⇒ Extension to Distributed FS

DataBases→ Data stored and managed via dedicated systems
⇒ Extension to Distributed DB (mostly NoSQL)

ObjectStores → Arbitrary data (files, blocks, etc…) stored in “buckets”
⇒ Extensively used in Cloud Storage

FILE SYSTEMS

The FS manages and tracks the location of data in terms of a **file hierarchy**

- Data is **physically stored in blocks** (e.g. 512 Bytes) scattered within the available storage devices (HDD/…)
- The **abstraction** provided by the FS allows to identify data as files/directories by their name, path, and metadata
- The FS also provides the Operating System (OS) the interface for **low level operations** with the storage blocks as files/directories:
  - OPEN
  - CLOSE
  - READ
  - WRITE
  - SEEK
  - …

  } a file (i.e. scattered data blocks on the hardware device) → The common standard interface across most OSs for such FS operations is *POSIX*

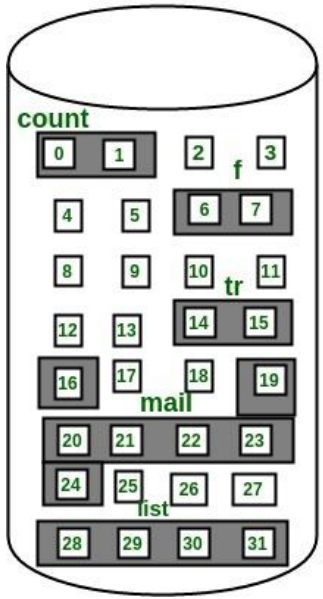
Depending on the FS, files can be accessed, and disk space be allocated, in a number of ways, including:

File Access:

1. Sequential Access
2. Direct/Random Access
3. Indexed Access

Space Allocation:

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation



Directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

□ = block (e.g. 4 kB)

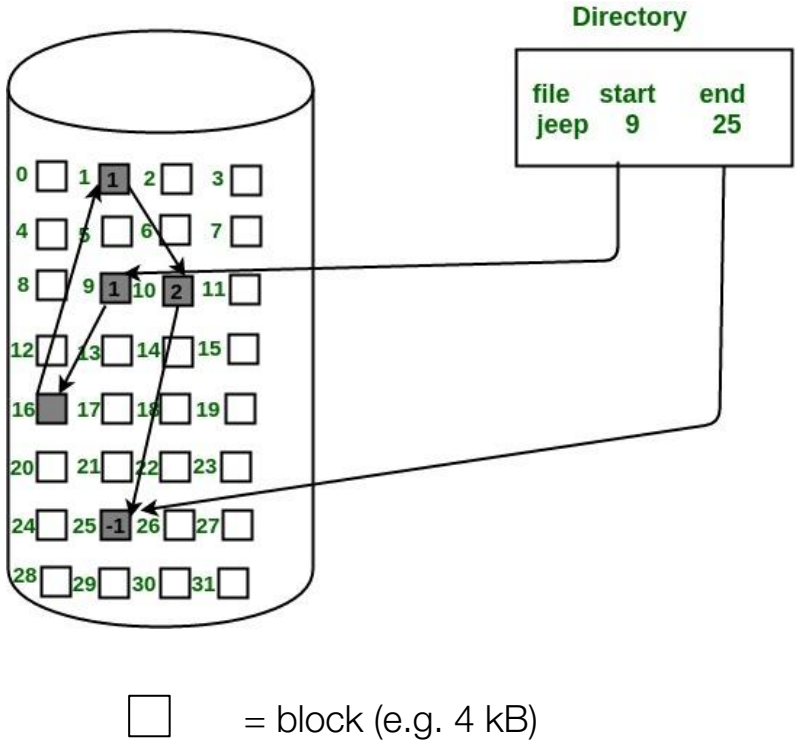
Depending on the FS, files can be accessed, and disk space be allocated, in a number of ways, including:

File Access:

1. Sequential Access
2. Direct/Random Access
3. Indexed Access

Space Allocation:

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation



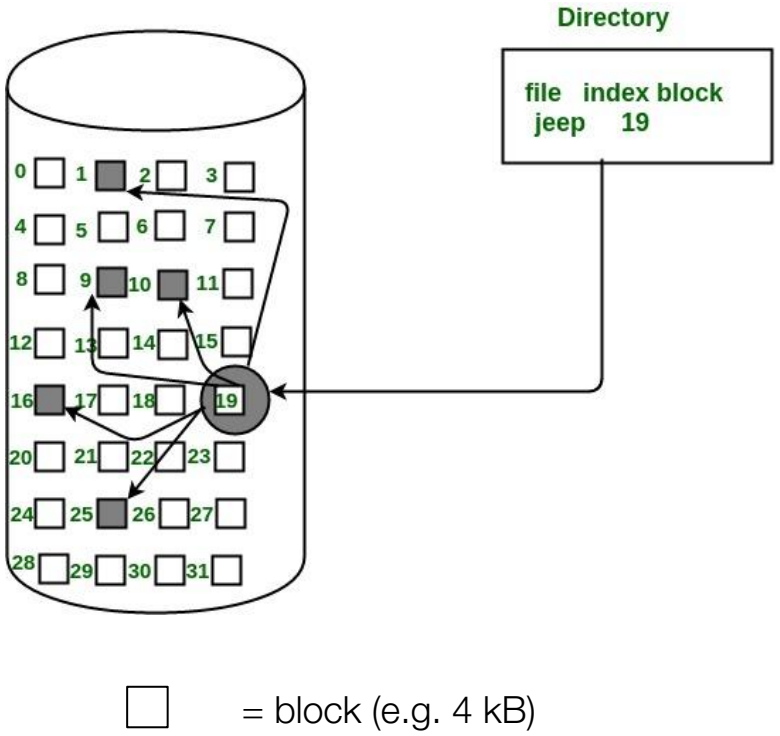
Depending on the FS, files can be accessed, and disk space be allocated, in a number of ways, including:

File Access:

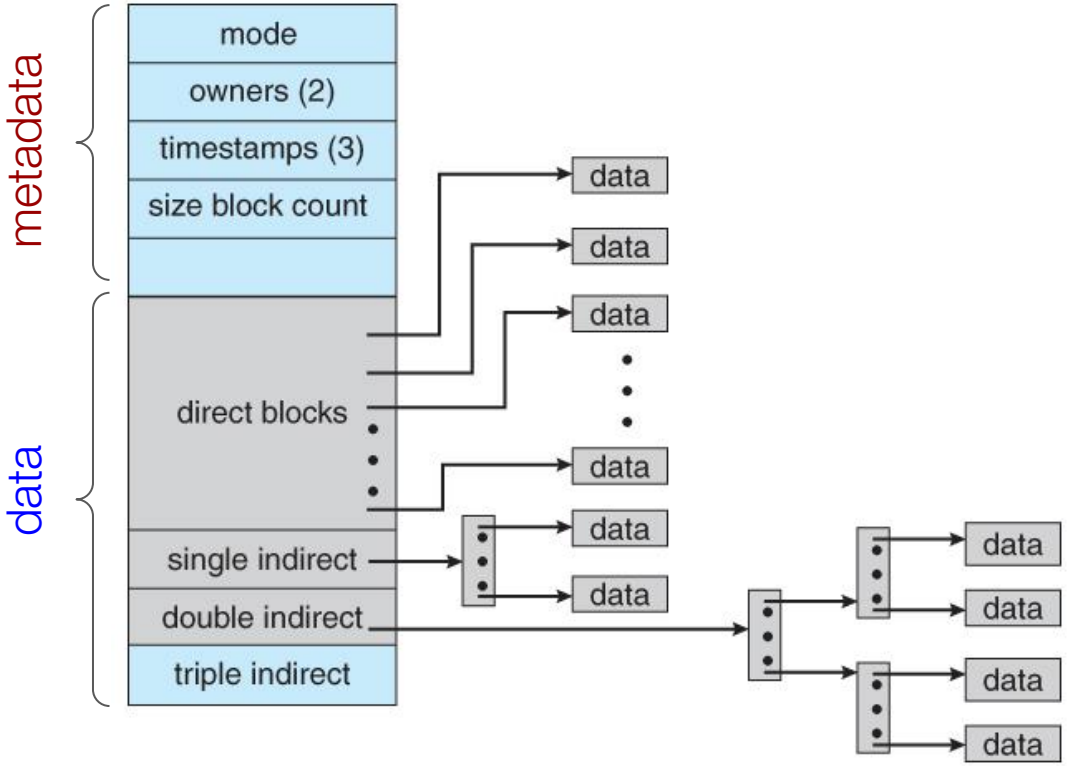
1. Sequential Access
2. Direct/Random Access
3. Indexed Access

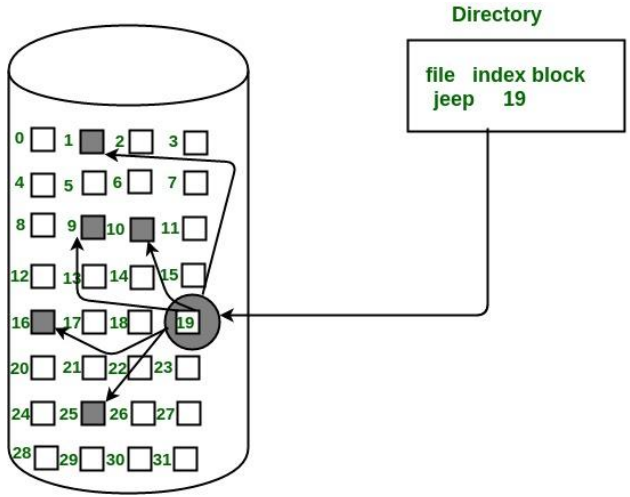
Space Allocation:

1. Contiguous Allocation
2. Linked Allocation
3. Indexed Allocation



In an indexed FS, each file is described by its own block, which in turn stores the indices ("pointers") to all other blocks containing the file raw data



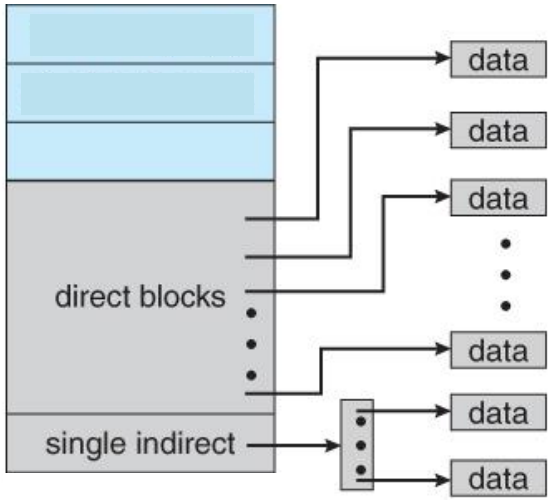


INDEXED FILE SYSTEMS

Let's assume we have an indexed FS with 4kB blocks; each indirect pointers has an index of 4B. In total, the FS has 12 direct blocks, and 1 single indirect block.

These are for example numbers close to the ones of an older file system, the Berkeley UNIX FFS (Fast File System).

How much data can we store using this FS?



INDEXED FILE SYSTEMS

Let's assume we have an indexed FS with 4kB blocks; each indirect pointers has an index of 4B. In total, the FS has 12 direct blocks, and 1 single indirect block.

These are for example numbers close to the ones of an older file system, the Berkeley UNIX FFS (Fast File System).

How much data can we store using this FS?

- Each direct data block contains 4 kB ⇒ **12 x 4kB = 48kB**

INDEXED FILE SYSTEMS

Let's assume we have an indexed FS with 4kB blocks; each indirect pointers has an index of 4B. In total, the FS has 12 direct blocks, and 1 single indirect block.

These are for example numbers close to the ones of an older file system, the Berkeley UNIX FFS (Fast File System).

How much data can we store using this FS?

- Each direct data block contains 4 kB ⇒ **12 x 4kB = 48kB**
- Each singly-indirect data block contains 4kB/4Bytes pointers
  - Each pointer points to a 4 kB data blocks
  - Thus each singly-indirect data block describes 1000 x 4 kB = 4 MB

⇒ **1 x 4MB = 4MB**

INDEXED FILE SYSTEMS

Let's assume we have an indexed FS with 4kB blocks; each indirect pointers has an index of 4B. In total, the FS has 12 direct blocks, and 1 single indirect block.

These are for example numbers close to the ones of an older file system, the Berkeley UNIX FFS (Fast File System).

How much data can we store using this FS?

- Each direct data block contains 4 kB ⇒ **12 x 4kB = 48kB**
- Each singly-indirect data block contains 4kB/4Bytes pointers
  - Each pointer points to a 4 kB data blocks
  - Thus each singly-indirect data block describes 1000 x 4 kB = 4 MB

⇒ **1 x 4MB = 4MB**

This very old File System would be finally capable to store **4048kB**
(out of which only ~1% is due to direct block allocation)

INDEXED FILE SYSTEMS

Let's assume we have an indexed FS with 4kB blocks; each indirect pointers has an index of 4B. In total, the FS has 12 direct blocks, and 1 single indirect block.

These are for example numbers close to the ones of an older file system, the Berkeley UNIX FFS (Fast File System).

What if there was also a 1 extra doubly-indirect block available?

- Each doubly-indirect data block contains 1000 pointers to indirect data blocks further containing 1000 pointers each
  - Thus, each doubly-indirect data block describes 1000 x 1000 x 4 kB = 4 GB

⇒ **1 x 4GB = 4GB**

Given the block and index sizes of this particular example, this File System would be finally capable to store:

$$n_{\text{direct blocks}} \times 4\ \text{kB} + n_{\text{singly-indirect blocks}} \times 4\ \text{MB} + n_{\text{doubly-indirect blocks}} \times 4\ \text{GB}$$

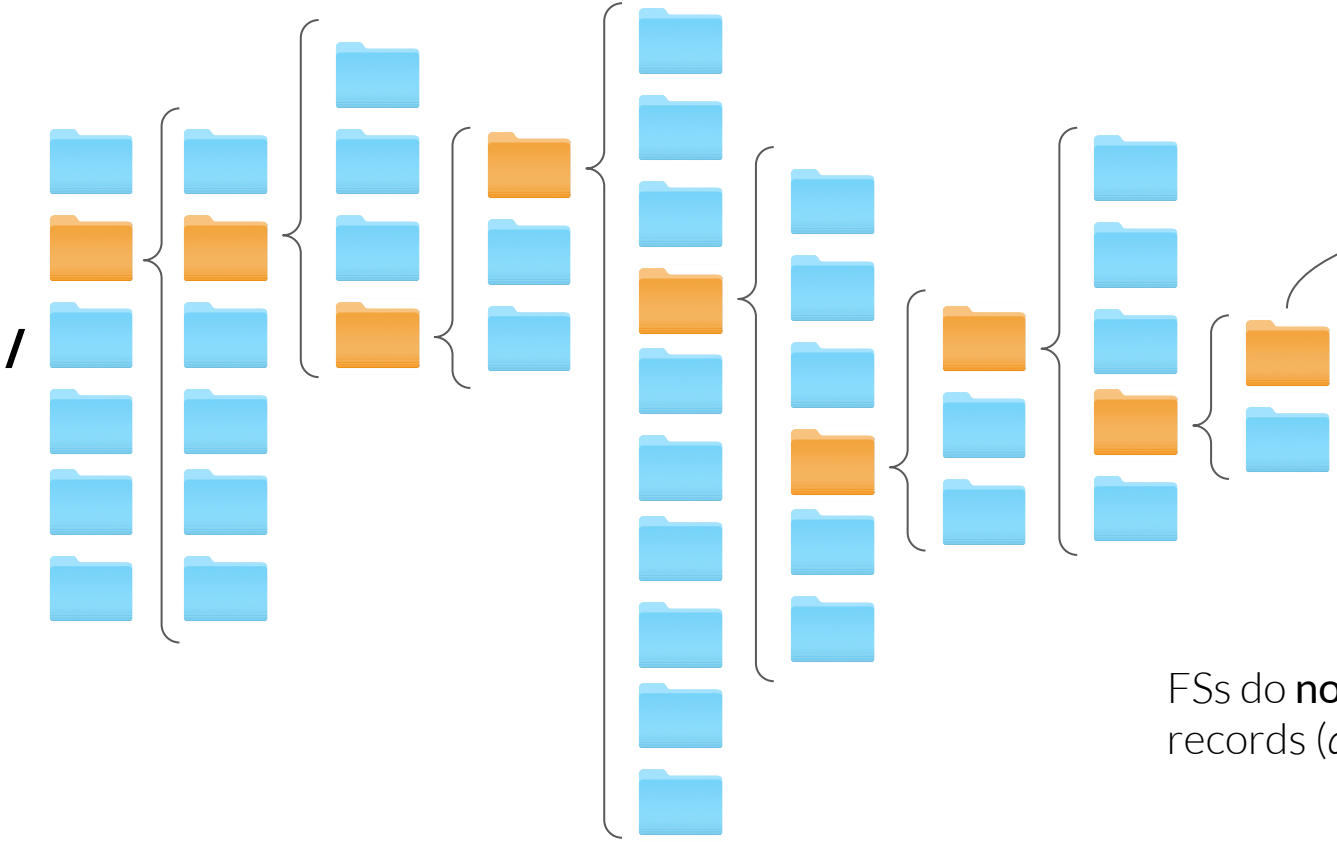
FILE SYSTEMS

	Individual file size limit	Single volume size limit	Filename limit	Metadata File Owner	Metadata File Permissions
FAT32	4 GiB	16 TiB	255 UCS-2 characters	No	No
NTFS	16 EiB	16 EiB	255 characters	Yes	Yes
exFAT	16 EiB	64 ZiB	255 UTF-16 characters	No	No
HFS	2 GiB	2 TiB	31 bytes	No	No
HFS+	8 EiB	8 EiB	255 UTF-16 characters	Yes	Yes
EXT2,3	16 GiB to 2 TiB	32 TiB	255 bytes	Yes	Yes
EXT4	16 GiB to 16 TiB	1 EiB	255 bytes	Yes	Yes

FILE SYSTEMS

FS provide a hierarchical tree-like structure for directory and file allocation

```
/eos/cms/store/data/Run2012D/SingleMu/RAW-RECO/ZMu-15Apr2014-v1/00001/*
```




```
BC1BA084-96CD-A328-002590D60134.root
A20DF339-E311-B5D3-001EC9D8A8A4.root
08FEFA88-BDCD-E311-E0CB4E19F995.root
…
```

+ other 84(!)

FSs do **not** provide any information on records (*datum*) contained inside files

FILE SYSTEMS

Pros:

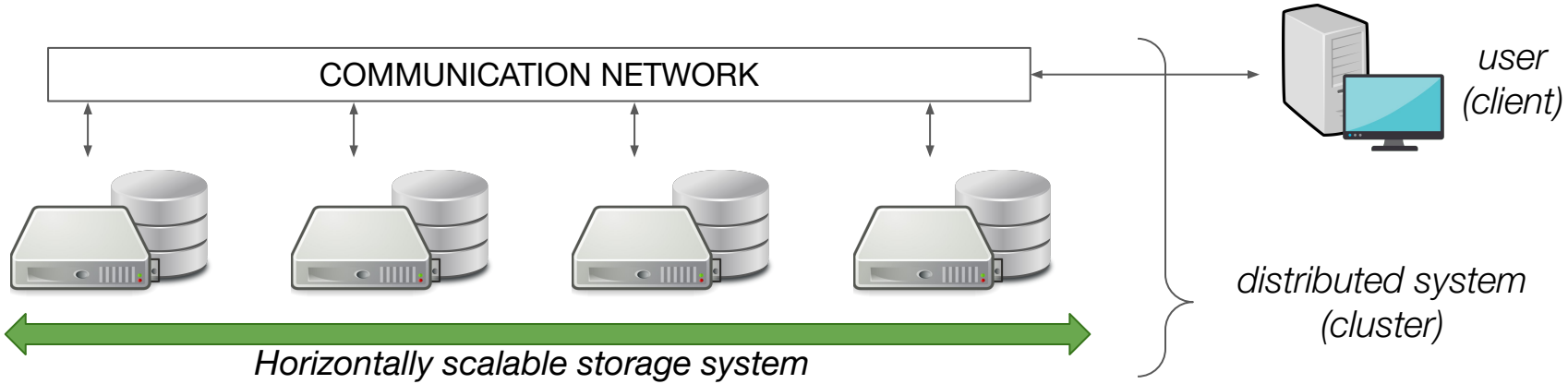
- Can manage huge amounts of data (often EB)
- `read/write/open/close/`... operations (POSIX) are FAST
- Offers a hierarchical view of all data
- Single entities (*files)* can be easily transferred-copied-shared-processed

Cons:

- Complex to define, maintain and scale a good data hierarchy for large datasets
- The files are the “atomic entity”, no underlying structure is accessible
- No way to group-select-filter the data based on file content

FROM LOCAL TO DISTRIBUTED FS

Distributed FS are ideally similar to Local FS, but are extended to environments where both data and processing may be spread out across several nodes (i.e. machines with attached storage)



The DFS resides on multiple nodes, but offers a coherent and uniform view of data stored on all disks

DISTRIBUTED FS

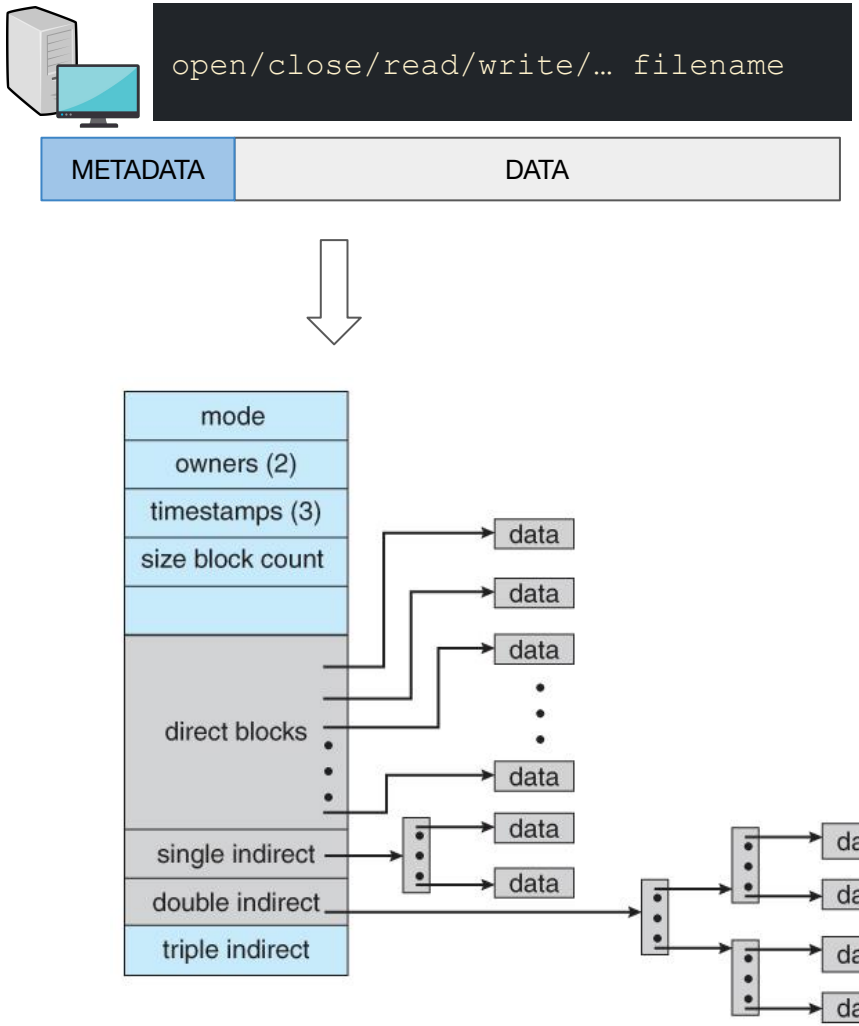
A number of DFS available, with very different implementations and applications:

- **NFS** → Network File System (1984)
- **AFS** → Andrew File System (1986)
- **Lustre** → Widely used in HighPerformanceComputing (HPC) (2003)
- **EOS** → Developed at CERN (2010)
- **GFS** → Google File System (2003)
- **HDFS** → Hadoop Distributed File System (2006)
- …

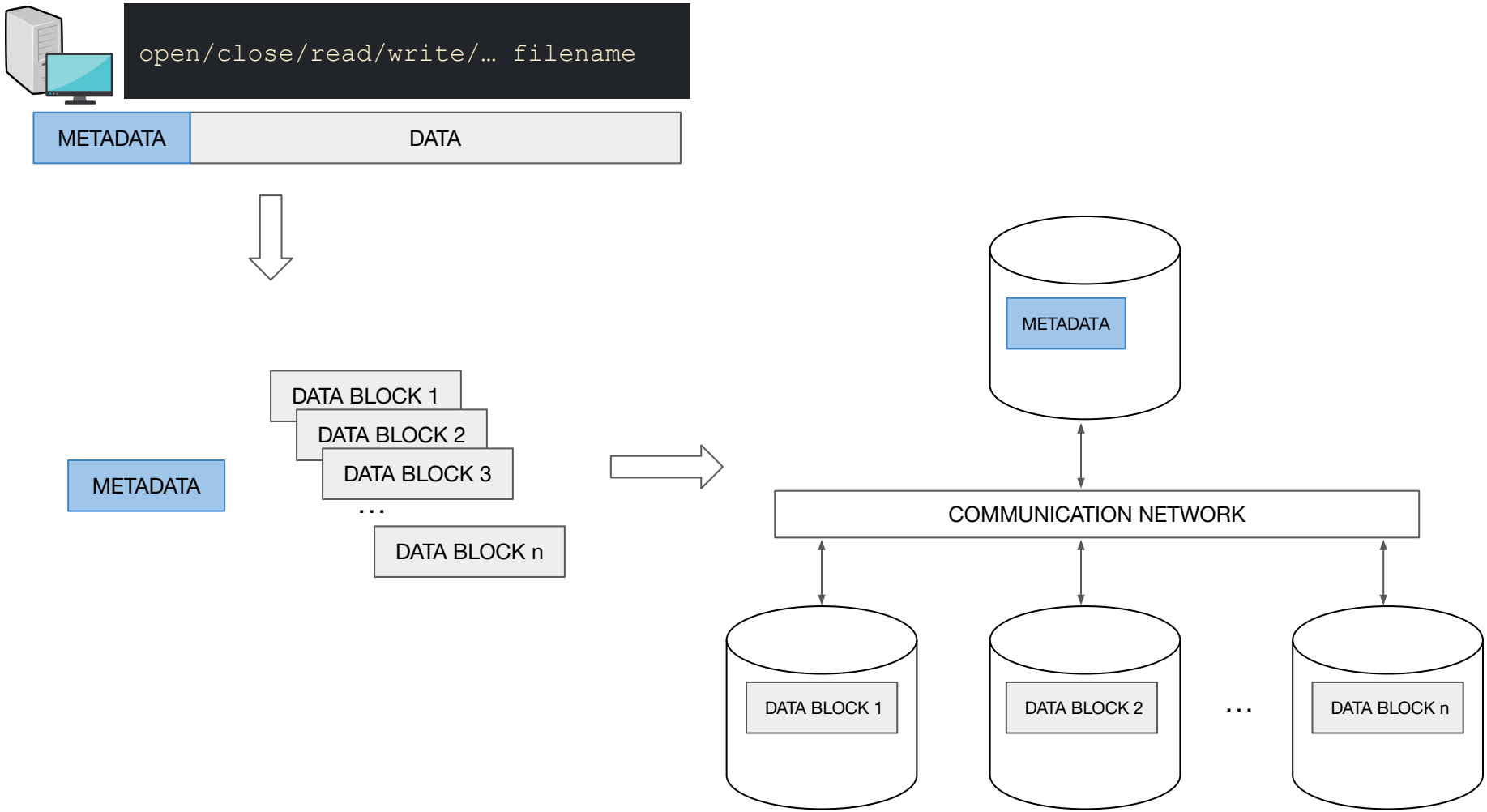
FROM FS TO DFS

```
open/close/read/write/… filename
```

FROM FS TO DFS



FROM FS TO DFS



IDEAL DFS REQUIREMENTS

- **Transparency**: users should access the system regardless where they log-in from, be able to perform the same operations on DFSs and local FS, and should not been able to experience potential failures of part of the system

  ⇒ *work with distributed data as if it was "local"*

- **Fault tolerance**: the distributed system should not halt in case of failures (even temporary) of parts of its infrastructure

  ⇒ *be resilient against network/server failures, data integrity issues, …*

- **Scalability**: the system should efficiently leverage from the usage of large numbers of servers (potentially dynamically) added into the system

  ⇒ *be extensible with the increase of storage nodes*

A DFS EXAMPLE: HDFS

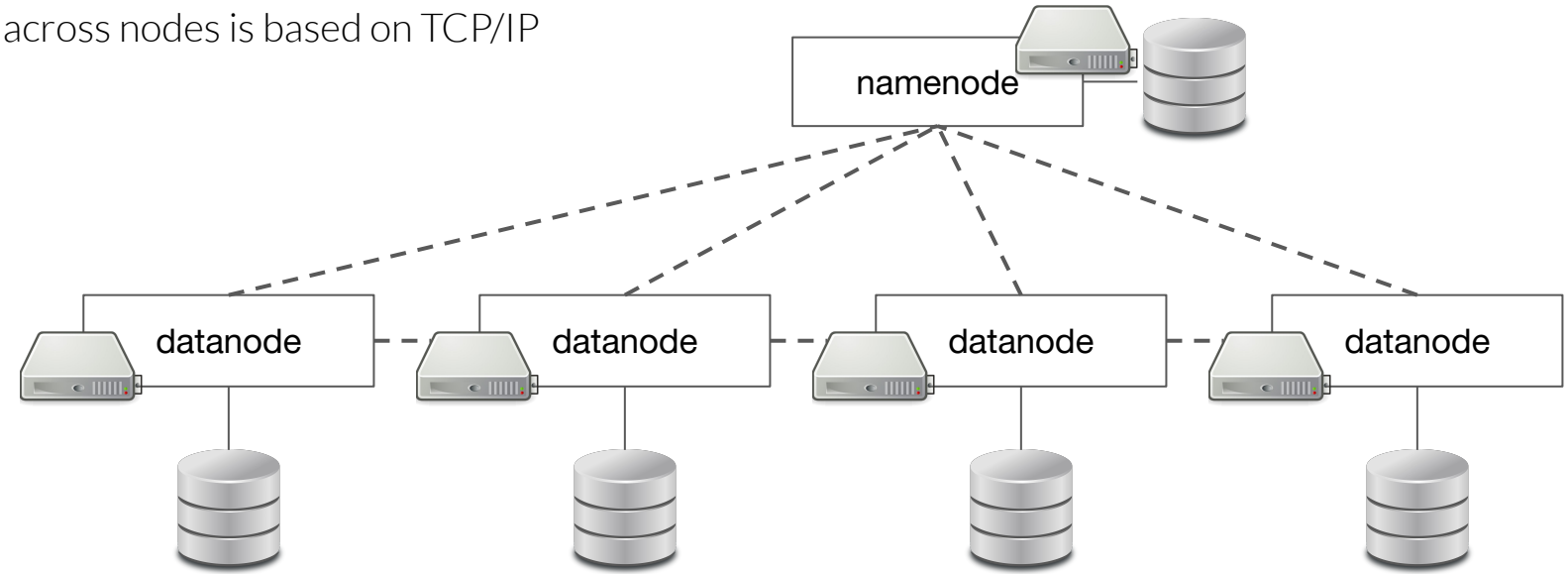
DFS designed to run on commodity hardware for large data applications

2 main type of nodes:

Namenode → 1 node hosting the metadata (file names, blocks, ...) and taking all decisions
Datanodes → a number of nodes hosting the data in blocks

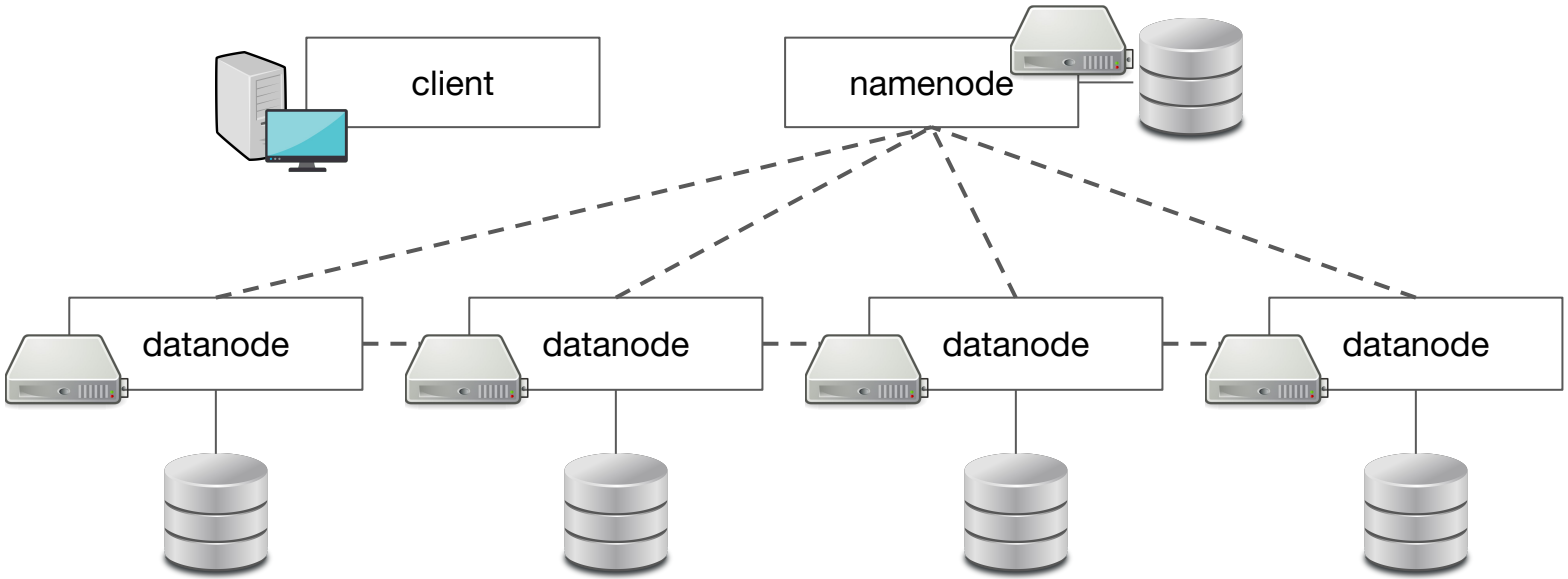
A typical block size in HDFS is 128 MB (designed for large file applications)

Communication across nodes is based on TCP/IP



A DFS EXAMPLE: HDFS

Client requests (read/write/…) are directed only to the namenode.
The namenode will reply and redirect the client to the appropriate block locations.

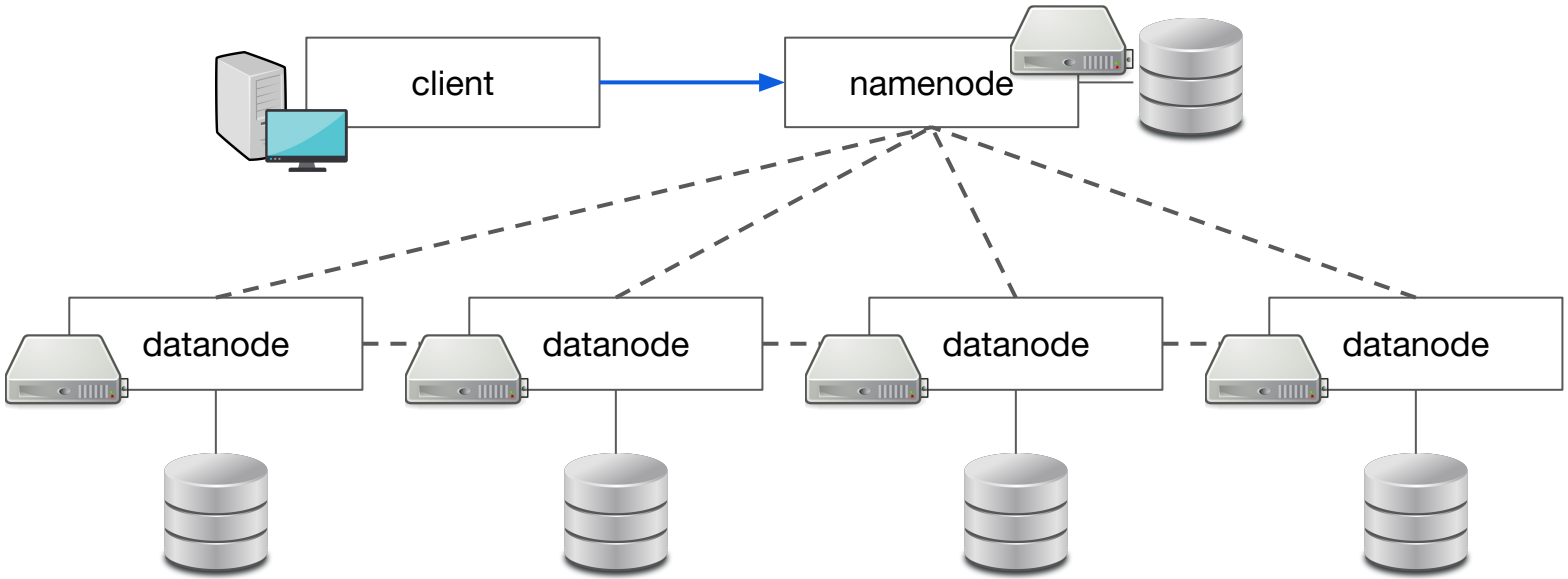


A DFS EXAMPLE: HDFS

Client requests (read/write/…) are directed only to the namenode.
The namenode will reply and redirect the client to the appropriate block locations.

Simplified example flow of **`write`** request:

1. The client issues the request to the namenode

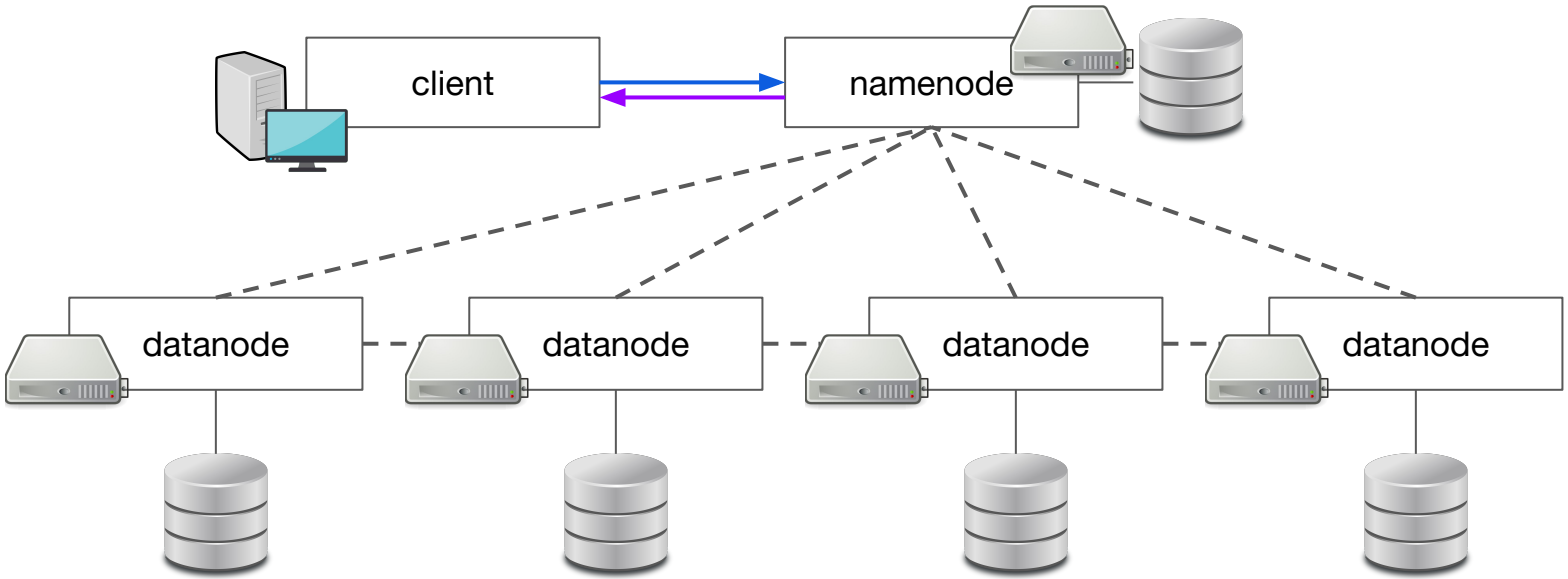


A DFS EXAMPLE: HDFS

Client requests (read/write/…) are directed only to the namenode.
The namenode will reply and redirect the client to the appropriate block locations.

Simplified example flow of **`write`** request:

1. The client issues the request to the namenode
2. The namenode checks for authentication/access privileges/etc and provides the location where to write the data blocks

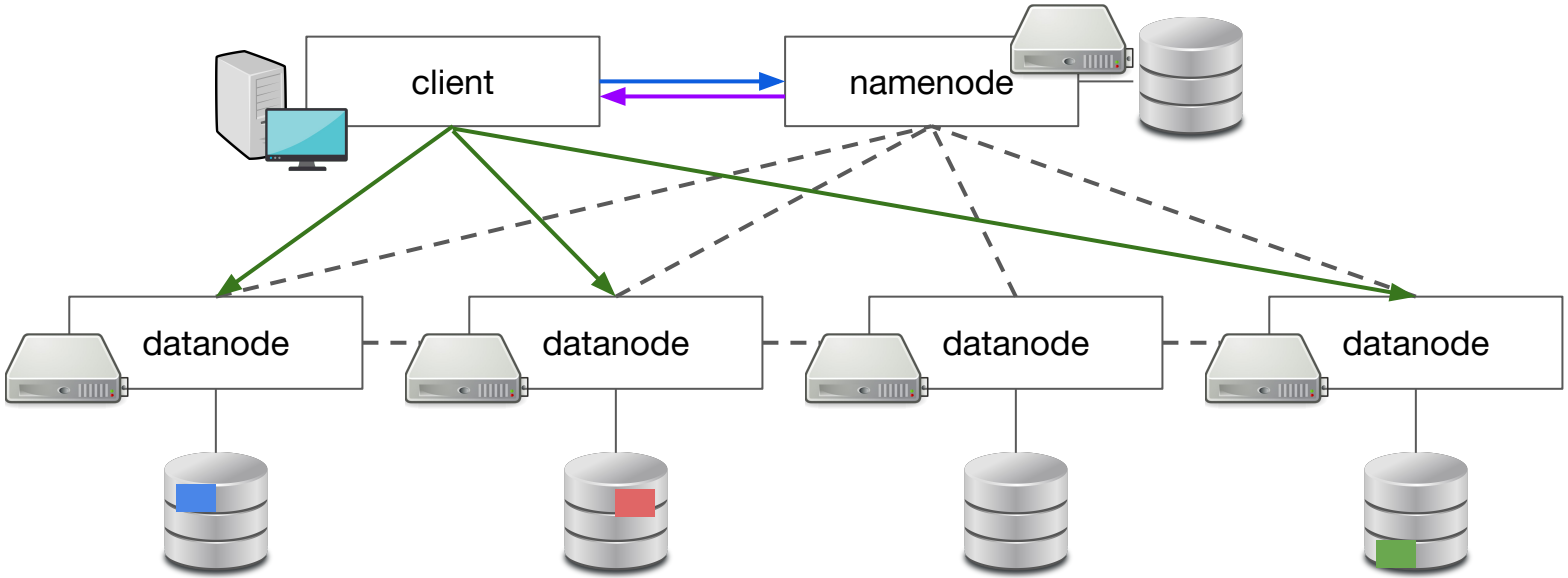


A DFS EXAMPLE: HDFS

Client requests (read/write/…) are directed only to the namenode.
The namenode will reply and redirect the client to the appropriate block locations.

Simplified example flow of **`write`** request:

1. The client issues the request to the namenode
2. The namenode checks for authentication/access privileges/etc and provides the location where to write the data blocks
3. The client starts writing the data blocks directly to the datanodes w/o passing through the namenode

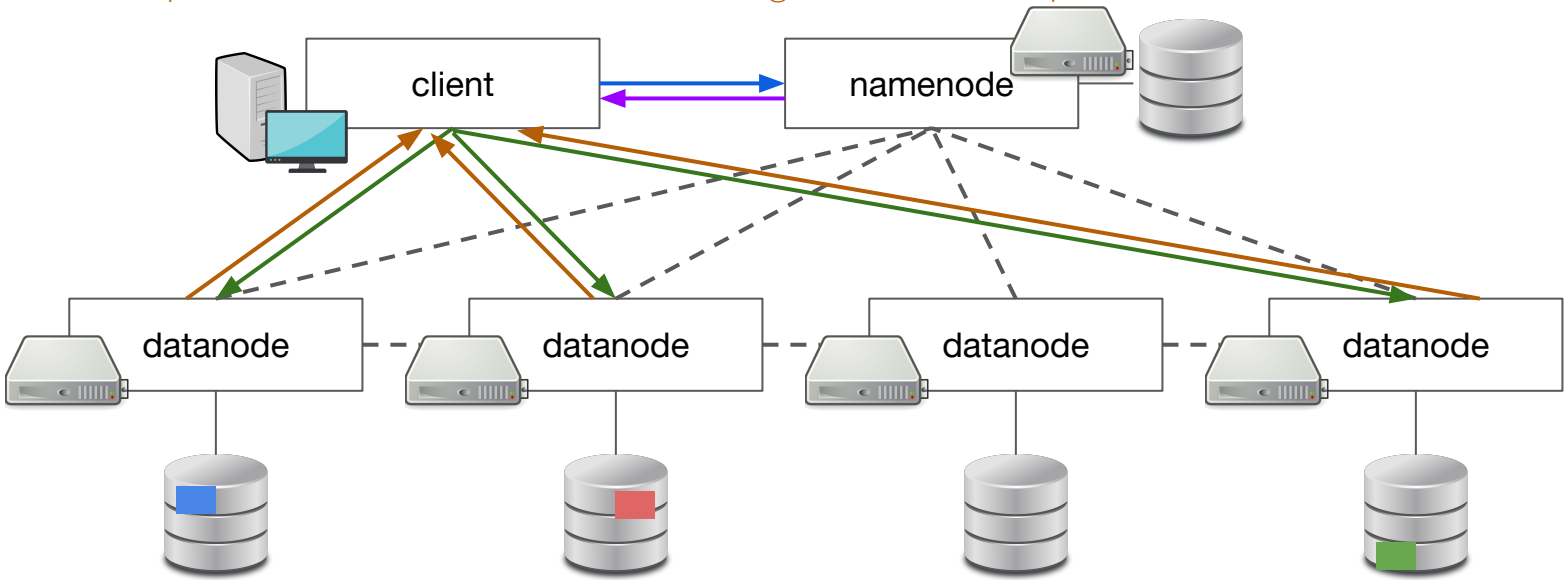


A DFS EXAMPLE: HDFS

Client requests (read/write/…) are directed only to the namenode.
The namenode will reply and redirect the client to the appropriate block locations.

Simplified example flow of **`write`** request:

1. The client issues the request to the namenode
2. The namenode checks for authentication/access privileges/etc and provides the location where to write the data blocks
3. The client starts writing the data blocks directly to the datanodes w/o passing through the namenode
4. The datanodes will report back to the client to acknowledge the write completion



A DFS EXAMPLE: HDFS

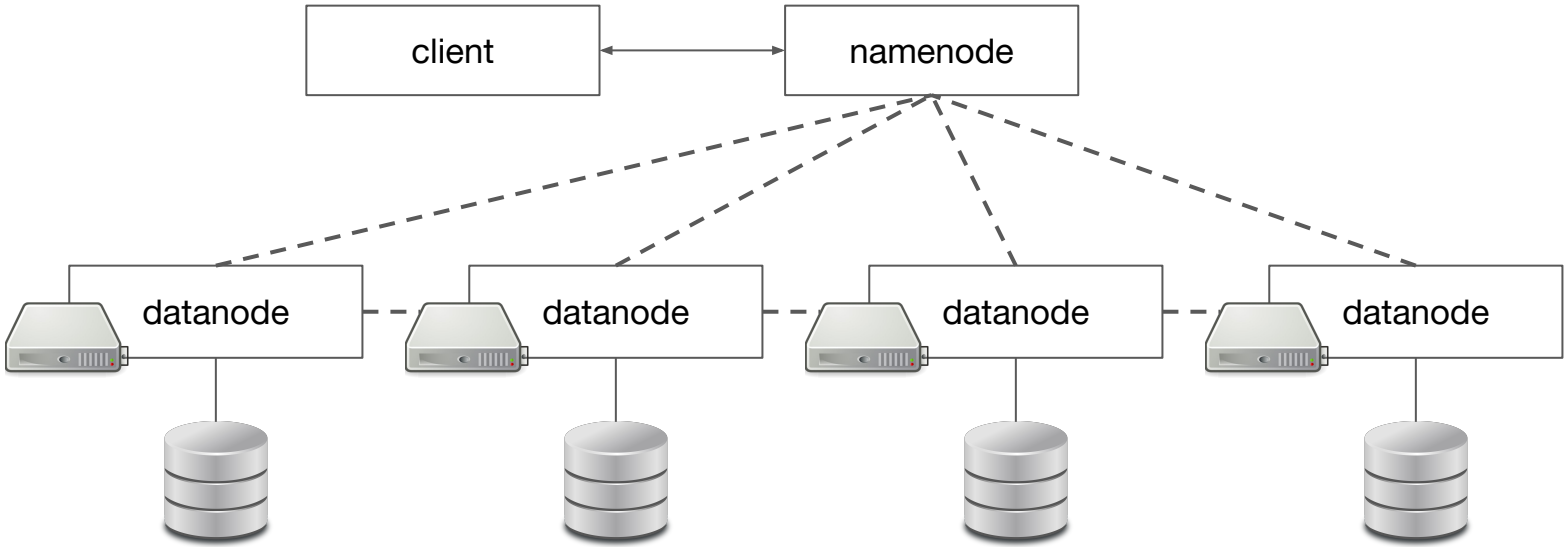
HDFS provides **Transparency** and **Scalability**. But what about **Fault Tolerance**?

Three main issues to be addressed:

1) Network instability

2) Datanode failure

3) Namenode failure



A DFS EXAMPLE: HDFS

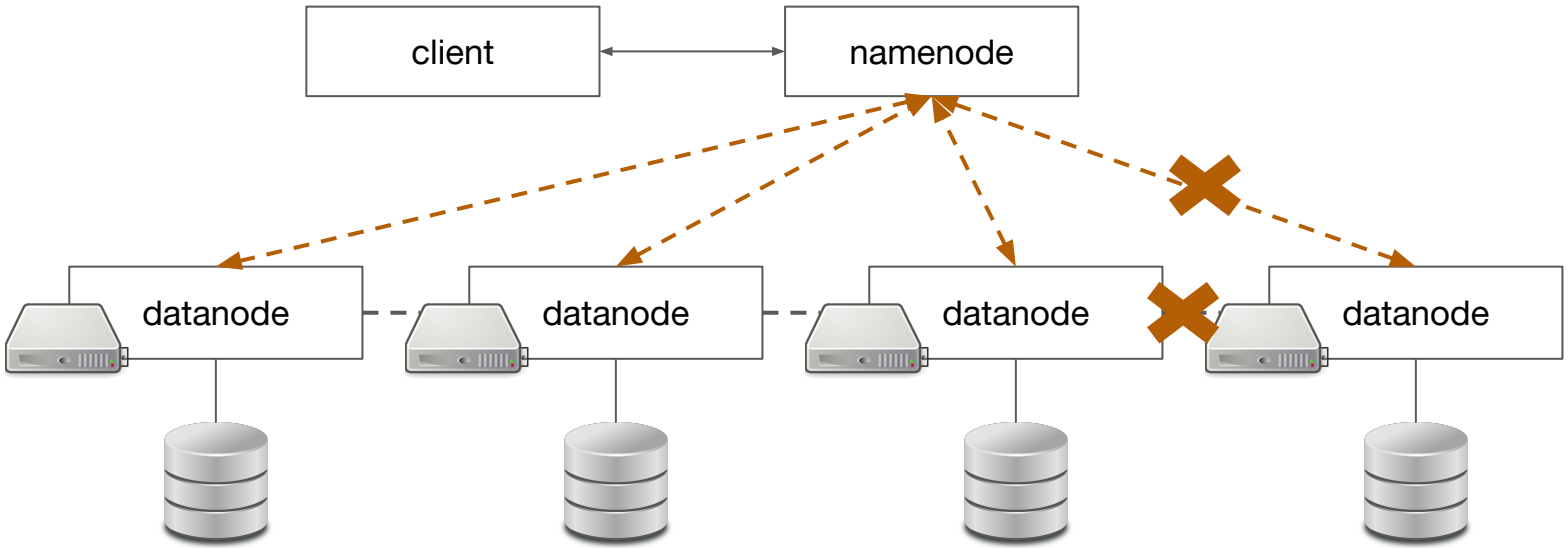
HDFS provides **Transparency** and **Scalability**. But what about **Fault Tolerance**?

Three main issues to be addressed:

1) **Network instability** → the datanodes repeatedly send a "heartbeat" signal to the namenode

2) Datanode failure

3) Namenode failure



A DFS EXAMPLE: HDFS

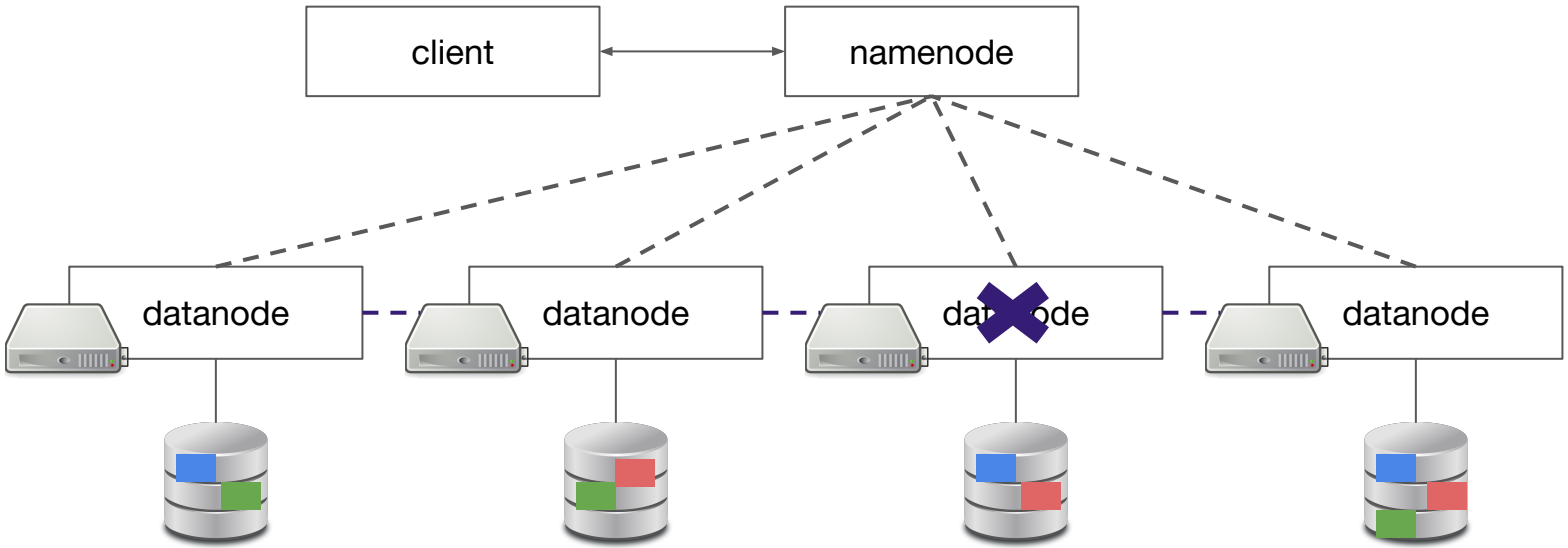
HDFS provides **Transparency** and **Scalability**. But what about **Fault Tolerance**?

Three main issues to be addressed:

1) Network instability → the datanodes repeatedly send a "heartbeat" signal to the namenode

2) Datanode failure → data-blocks are replicated across multiple nodes (default is 3). No need for RAID

3) Namenode failure



A DFS EXAMPLE: HDFS

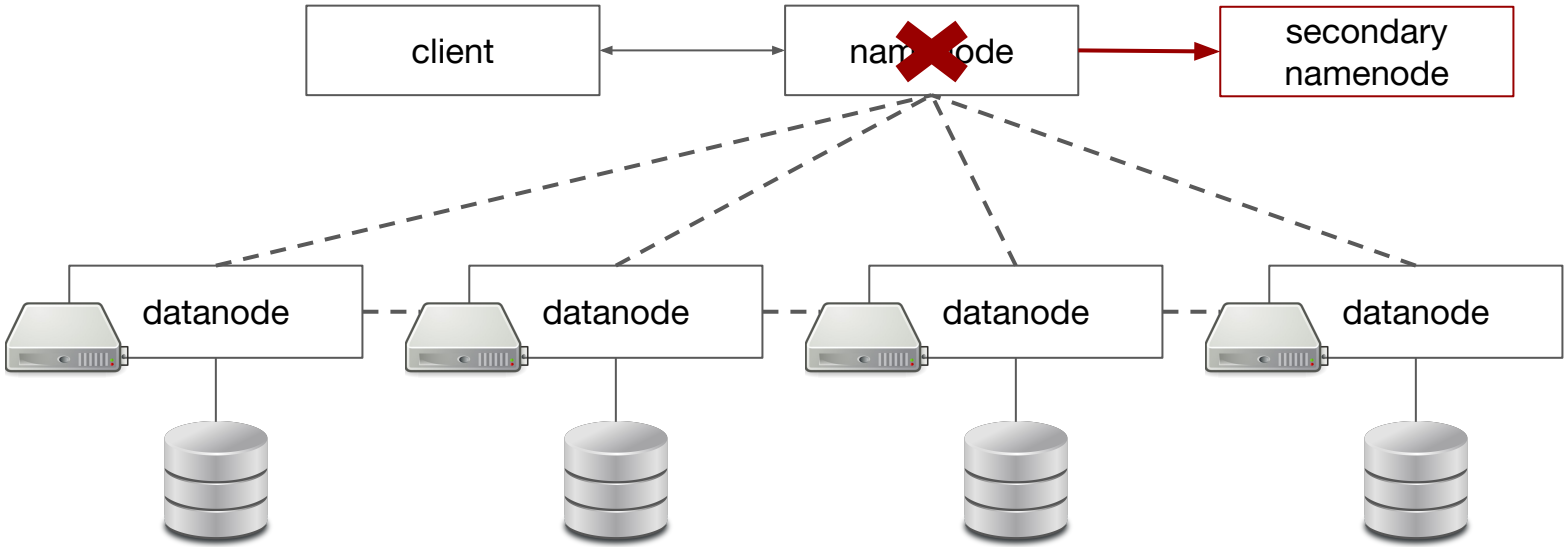
HDFS provides **Transparency** and **Scalability**. But what about **Fault Tolerance**?

Three main issues to be addressed:

1) Network instability → the datanodes repeatedly send a "heartbeat" signal to the namenode

2) Datanode failure → data-blocks are replicated across multiple nodes (default is 3). No need for RAID

3) Namenode failure → huge issue, as it's a Single Point Of Failure (SPOF)
HDFS implements a secondary namenode to rebuild the metadata if the primary fails

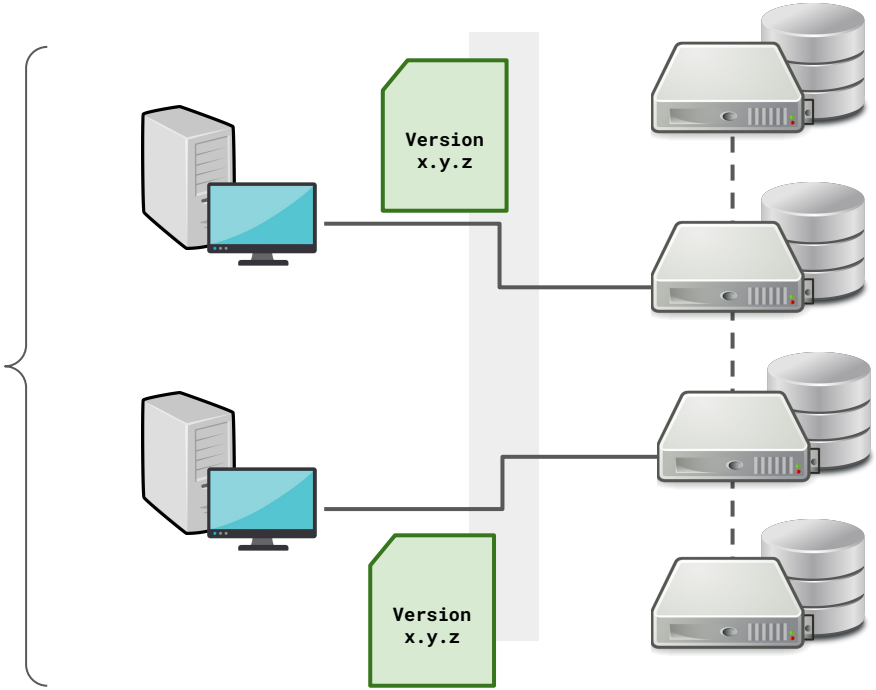


CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

Consistency →Every request made to any data node returns the very same copy of the data (the most recent version)

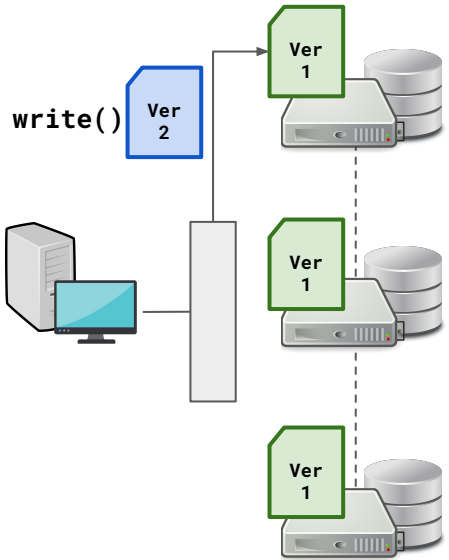
Two clients read the same file. The requests are served by two different nodes and they do get the same version of the data.



CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

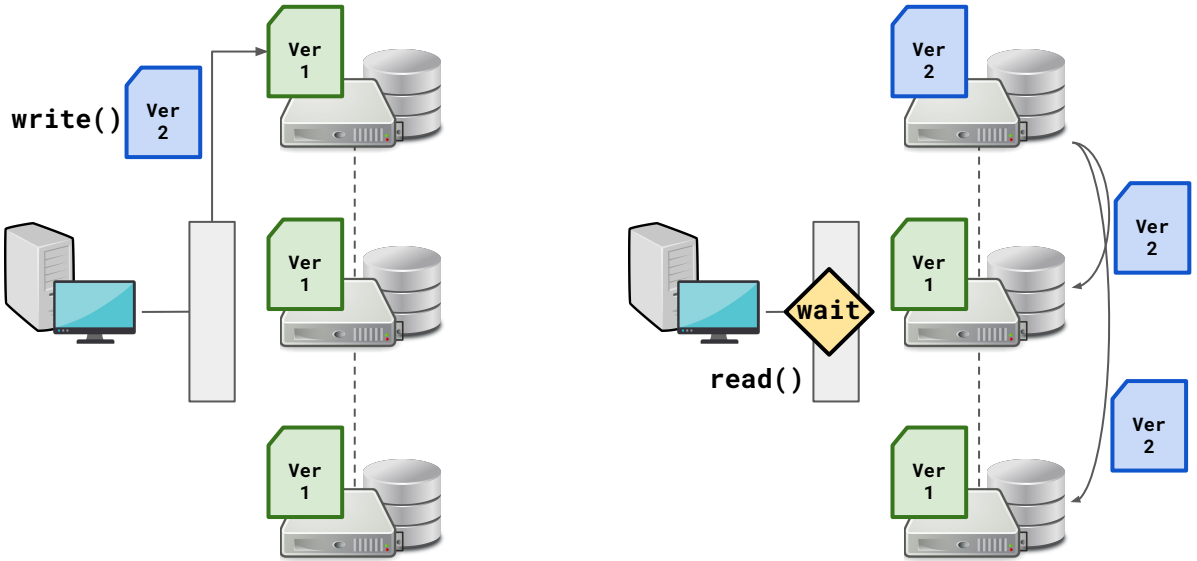
Consistency →Every request made to any data node returns the very same copy of the data (the most recent version)



CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

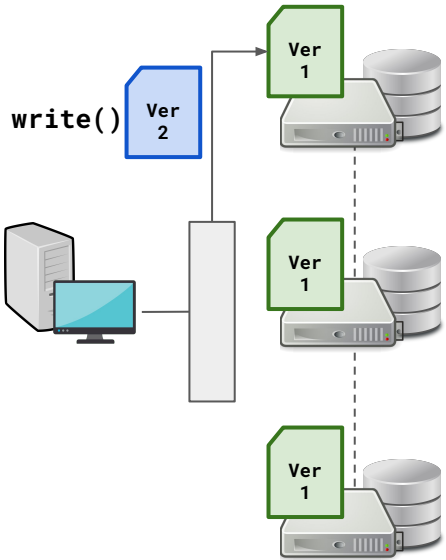
Consistency →Every request made to any data node returns the very same copy of the data (the most recent version)

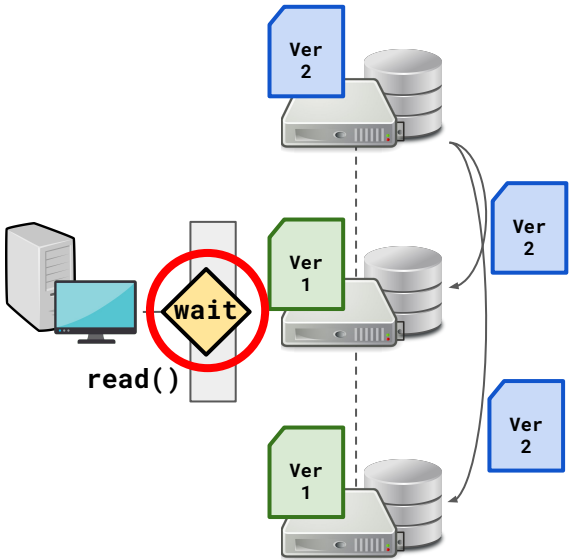


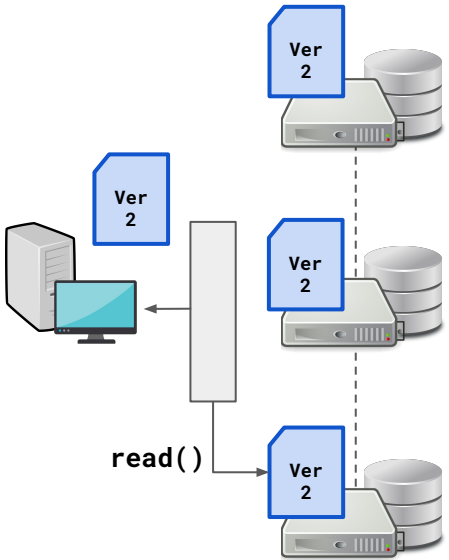
CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

Consistency →Every request made to any data node returns the very same copy of the data (the most recent version)





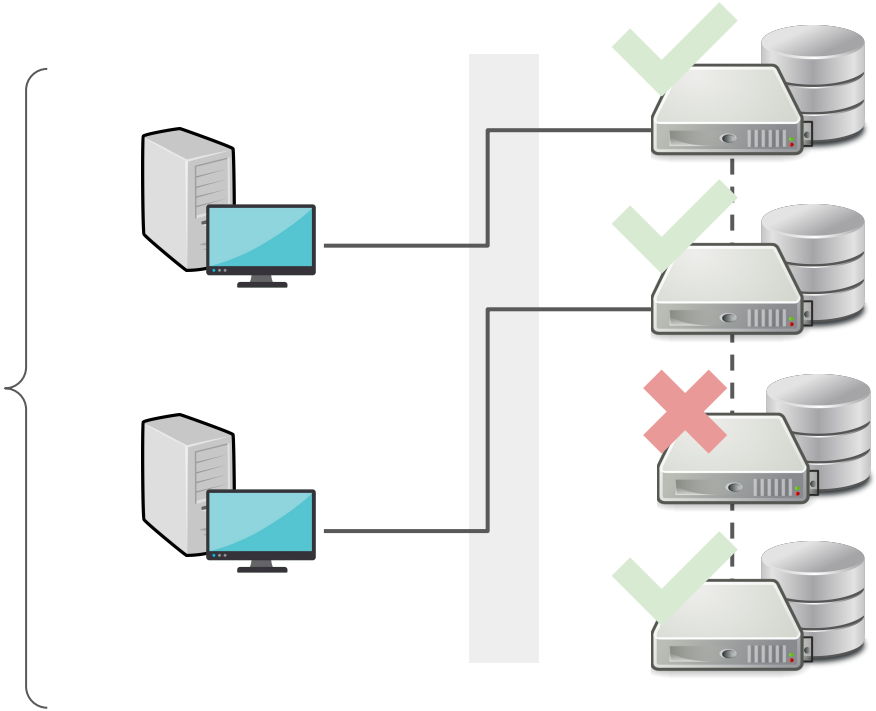


CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

Availability → Every request to access/write/update data must receive a response (regardless of the state of the data) even if one or more nodes are down

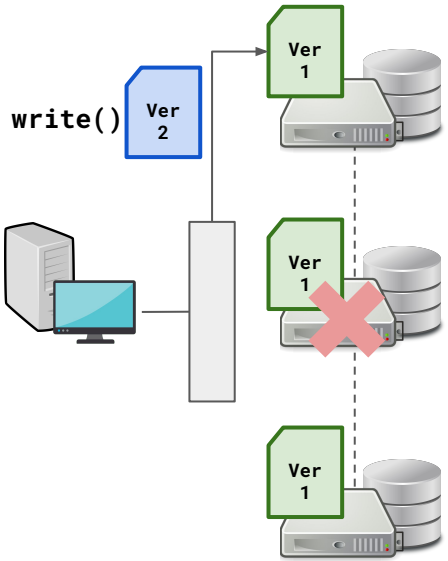
Two clients send requests to the distributed system and they both get an immediate response from the working nodes.



CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

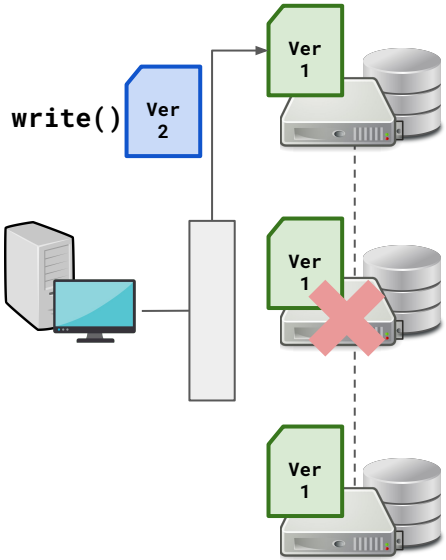
Availability → Every request to access/write/update data must receive a response (regardless of the state of the data) even if one or more nodes are down

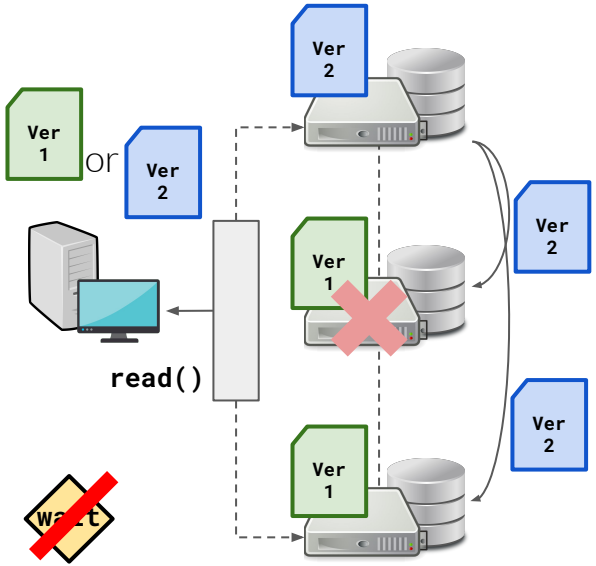


CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

Availability → Every request to access/write/update data must receive a response (regardless of the state of the data) even if one or more nodes are down



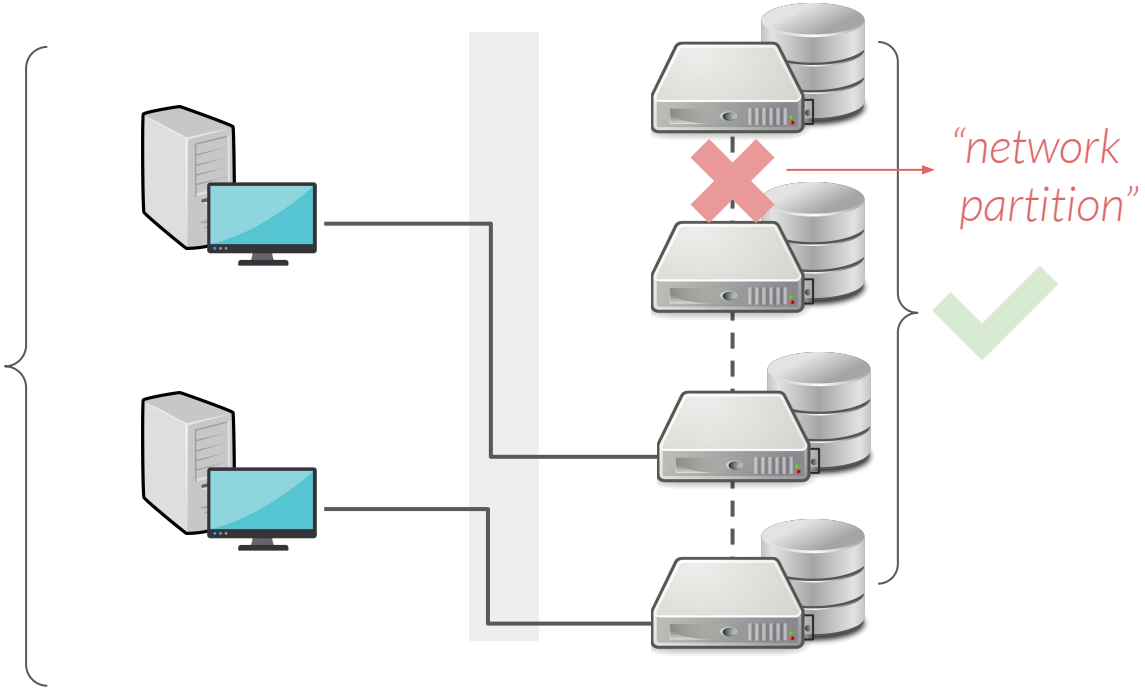


CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

Partition tolerance → The system continues to operate despite failures in the communication between nodes (known as *network partitions*)

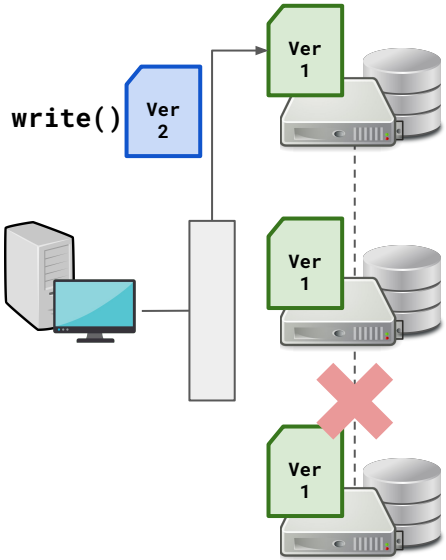
Two clients send requests to the distributed system. Despite an arbitrary number of network partitions the distributed system continues to operate.

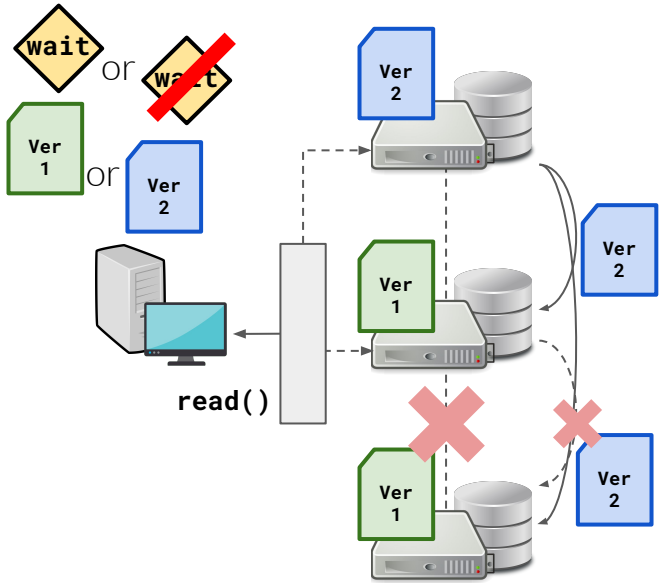


CAP THEOREM

In computer science three concepts are crucial when dealing with distributed systems (both for data storage and processing):

Partition tolerance → The system continues to operate despite failures in the communication between nodes (known as *network partitions*)

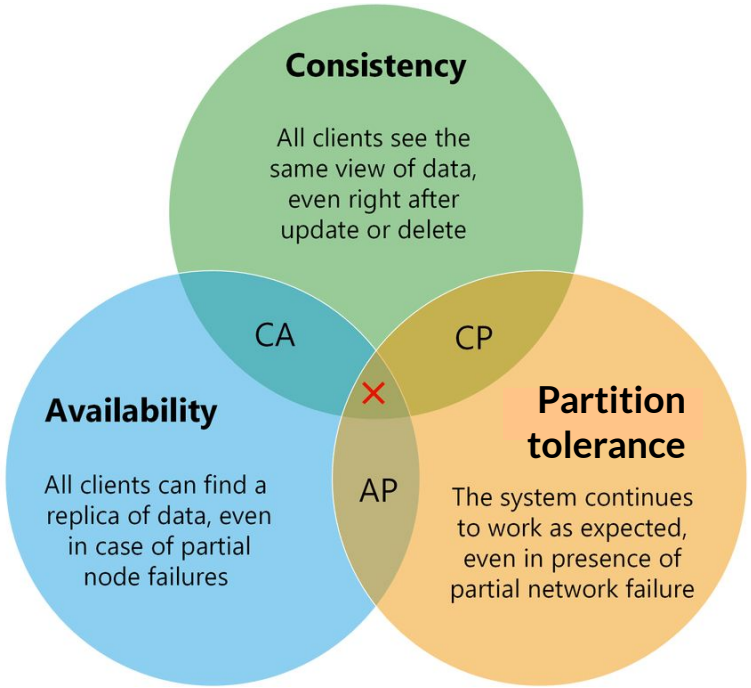




CAP THEOREM

In computer science, the **CAP theorem** (also Brewer's theorem), states that ***it is impossible for a distributed system to simultaneously provide more than two out of the three C/A/P guarantees***

In distributed systems, there is no way to completely avoid Network Partitions, so to keep the cluster operational we must choose between Consistency and Availability



CAP THEOREM

In computer science, the **CAP theorem** (also Brewer's theorem), states that ***it is impossible for a distributed system to simultaneously provide more than two out of the three C/A/P guarantees***

In distributed systems, there is no way to completely avoid Network Partitions, so to keep the cluster operational we must choose between Consistency and Availability

AP →Get access to the data at any time, but no guarantee that the data will be the most recent value for each read

CP →Get the most recent value of data at any reads, but no guarantee that the system is available even in the case of a network partition

AC →Both consistent and available, but cannot deliver fault tolerance in the case of a partition

[not really an option in a *distributed* data management system BUT common choice for vertically scalable systems]