

Cloud storage

Learning objectives

The following is a checklist that students can use during their learning in order to self-assess their mastery of the material.

- a. Can you describe the limitations of traditional (local) file systems?
- b. Can you explain what object storage is?
- c. Can you explain what the benefits of object storage are?
- d. Can you contrast or relate object storage with block storage, a file system, and the key-value model?
- e. Can you explain the three different ways, on the physical level, to deal with more data (scale up, scale out, write better code)?

- f. Can you explain why scaling out is less expensive than scaling up?
- g. Can you explain what aspects of the design of object storage enable scaling out and why?
- h. Do you know what a data centre is made of (racks, server nodes, storage nodes, etc.)?
- i. Do you know the difference between storage, memory, CPU and network and how the three are paramount in a cluster?
- j. Do you know rough, typical numbers (per-node storage capacity, memory, number of cores, etc.)?
- k. Do you know how storage and memory technologies (HDD, SSD and RAM) compare in terms of capacity and throughput?

- l. Can you sketch the (key-value) data model behind object storage? Can you spot instances of the key-value model in several other technologies seen in the lecture?
- m. Do you know the difference between data and metadata?
- n. Do you know the order of magnitude that can be achieved in terms of number of objects, and object size?
- o. Can you name a few big players for cloud-based object storage (vendors, consumers)?
- p. Can you describe the features of S3 and Azure Blob Storage on a high level? Do you know what a bucket and object are? What block blob storage, append blob storage and page blob storage are and how they work?

- q. Can you describe what the most important SLA (Service Level Agreement) parameters mean (e.g., latency, availability, durability) as well as their typical range?
- r. Can you describe a typical use case for object storage?
- s. Can you explain what a REST API is, what resources and methods are? Do you know the most important status codes (200, 404, 500, ...)? Do you know the most important methods? Do you know what a URI is? (Note that the difference between URI, IRI, URN, URL has become irrelevant lately and for the purpose of the course)
- t. Do you know what each letter stands for in CAP?
- u. Can you explain why, for large amounts of data, CAP becomes relevant over ACID?

Distributed file systems

In the previous chapter, we saw that cloud storage services provide a global service that is able to store billions, trillions of objects. We also saw that the size of these objects is limited, for example in Amazon S3, an object cannot exceed 5 TB. In other words, they can store huge amounts of large files.

What about large amounts of huge files? Is it possible to store individual objects bigger than a single machine? The answer is yes. It requires a shift in paradigm that:

- brings back the file hierarchy natively;
- supports block-based storage natively.

Main requirements of a distributed file system

Data goes through a lifecycle: there is the raw data that is directly collected from the real world (e.g., sensors, logs, etc). Later in the pipeline, this data gets processed, analyzed and turned into derived datasets. These datasets need to be written back to a large-scale storage backend. In turn, they might be processed again leading to more derived datasets, etc.

Such a storage backend must first be resilient to failure. While a single hard drive might occasionally fail, as some of us probably already experienced with their laptop, the nodes in a cluster with 1,000 machines and more is *guaranteed* to fail. Not once, not twice, all the time. Thus, the storage technology must be capable of

- monitoring itself;
- detecting failures;

- automatically recovering;
- being, in the end and as a whole, fault tolerant.

The access mode is also an important feature of the storage backend. On a laptop hard drive, random access is paramount: any part of the disk can be read and written at any time, and in any order. Of course, accessing data randomly and back and forth will be slower than reading a file stored contiguously on disk, or writing a file as a stream and in one go. But laptop hard drives are *designed* to be practically usable for random access.

For distributed data storage though, and for the use case at hand where we read a large dataset, analyze it, and write back the output as a new dataset, random access is not needed. Rather, we need to be able to:

- efficiently scan a large file (in its entirety) – for data analysis
- append efficiently new data at the end of an existing large file – particularly for logging and sensors

Furthermore, this must be supported even with hundreds of concurrent users reading and writing to and from the same file system.

Going back to our capacity-throughput-latency view of storage, a distributed file system is designed so that, in cruise mode, its bottleneck will be the data flow (throughput), not the latency. This aspect of the design is directly consistent with a full-scan pattern, rather than with a random access pattern, the latter being strongly latency-bound (in the sense that most of the time is spent waiting for data to arrive).

We saw that capacity increased much faster than throughput, and that this can be solved with parallelism.

We saw that throughput increased much faster than latency decreased, and that this can be solved with batch processing.

Distributed file systems support both parallelism and batch processing natively, forming the core part of the ideal storage system accessed by MapReduce or Apache Spark.



The origins of such a system come back to the design of GoogleFS, the Google File System. Later on, an open source version of it was released as part of the Hadoop project, initiated by Doug Cutting at Yahoo, and called HDFS, for Hadoop Distributed File System. The Hadoop projects further includes other open source releases of MapReduce (with the same name as Google's original MapReduce) and HBase (corresponding to Google's BigTable).

Between 2006 and 2016, in just ten years, the cluster size running HDFS successfully went from 188 nodes to almost 42,000 nodes (and by now probably more), storing hundreds of Petabytes of data.

The model behind HDFS

In almost every chapter of this course, we will carefully introduce the logical level separately from its physical implementation. Thus, we start by describing HDFS on the logical level.

File hierarchy

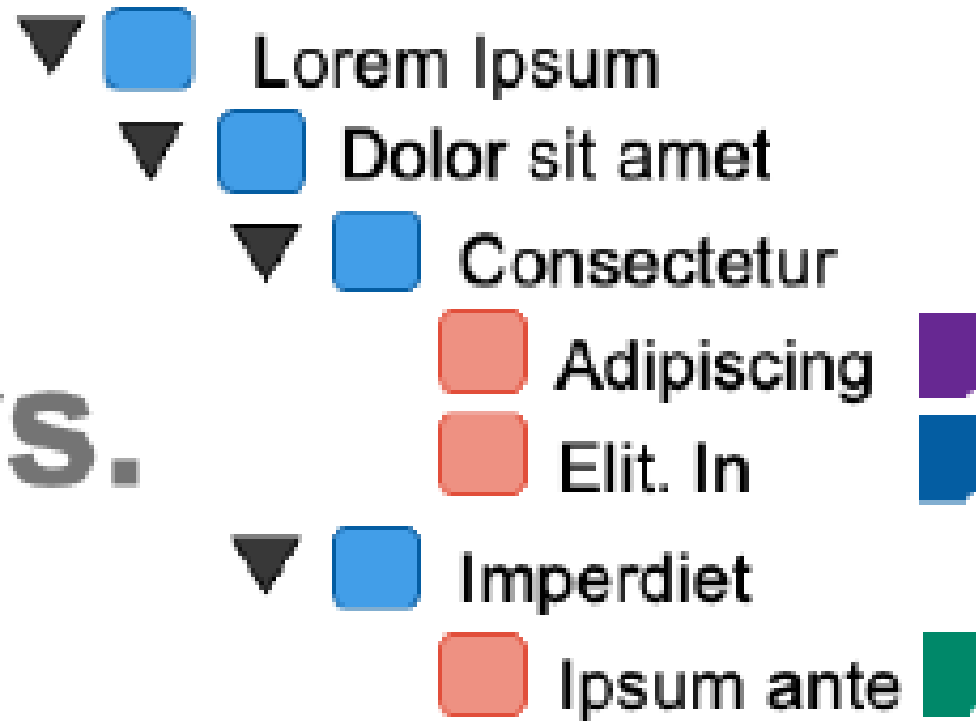
First, in HDFS, the word “file” is used instead of “object”, although they correspond to S3 objects logically.

HDFS does not follow a key-value model: instead, an HDFS cluster organizes its files as a hierarchy, called the file namespace. Files are thus organized in directories, similar to a local file system.

Lorem Ipsum	
Dolor sit amet	
Consectetur	
Adipiscing	
Elit. In	
Imperdiet	
Ipsum ante	

Key-Value Model

VS.



File Hierarchy

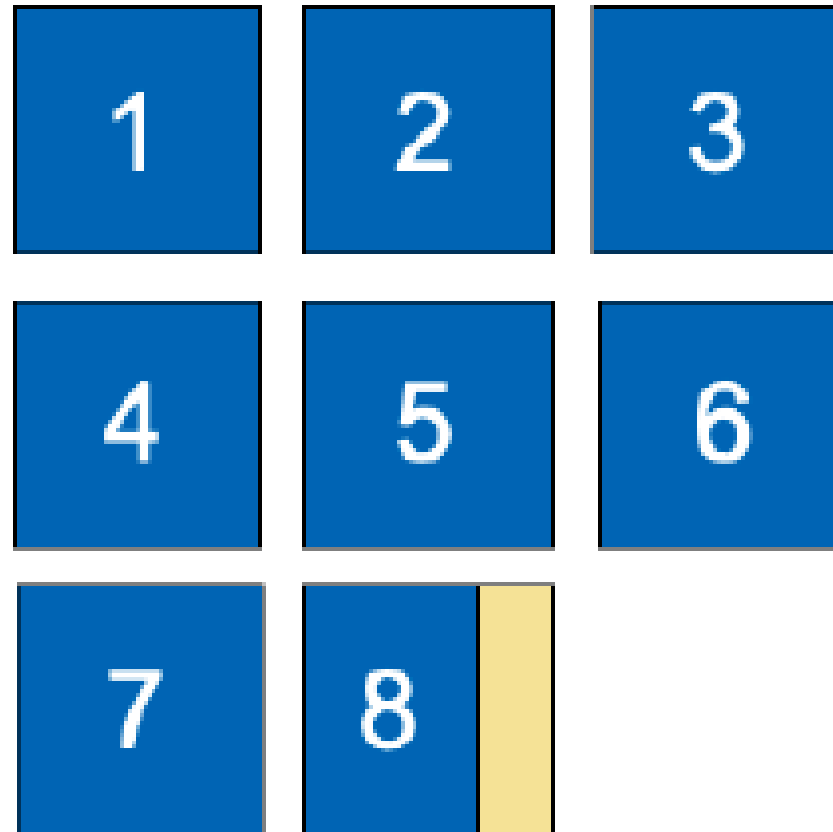
Blocks

Unlike in S3, HDFS files are furthermore not stored as monolithic black-boxes, but HDFS exposes them as lists of blocks – also similar to a local file system.



Object Storage

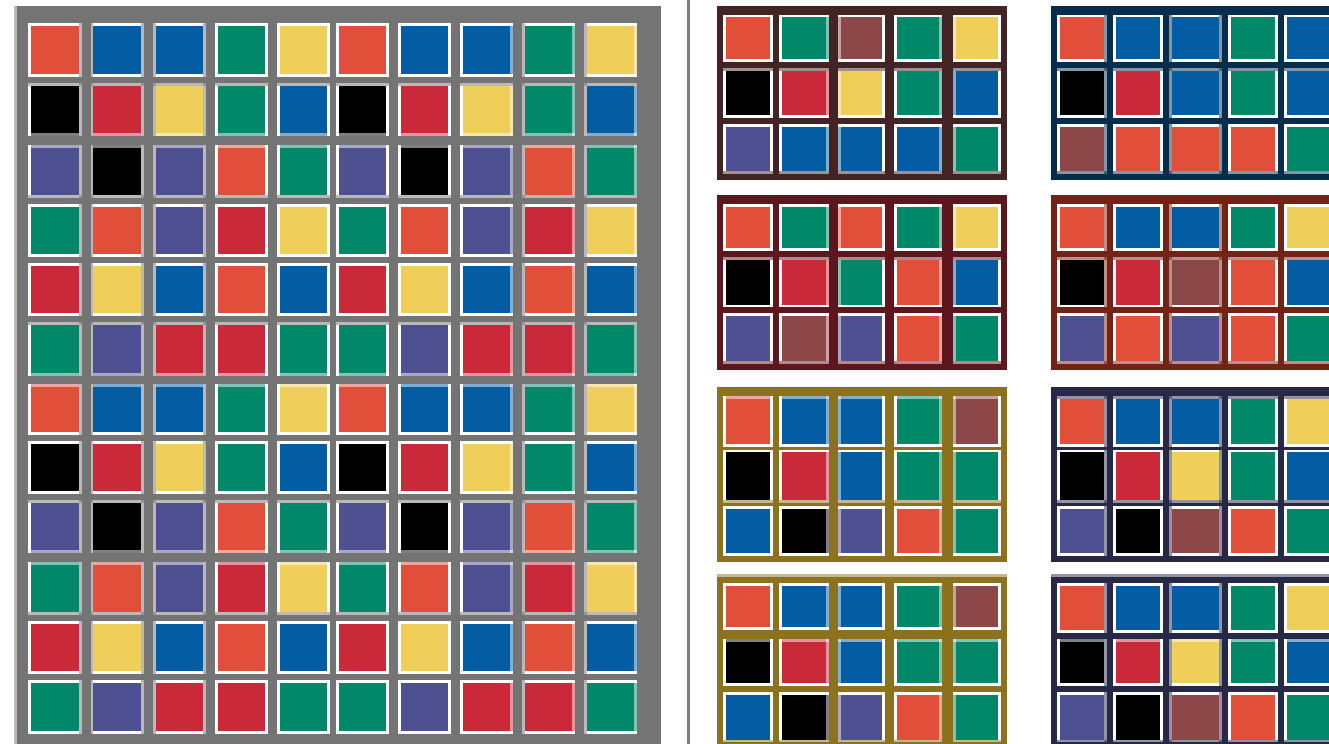
VS.



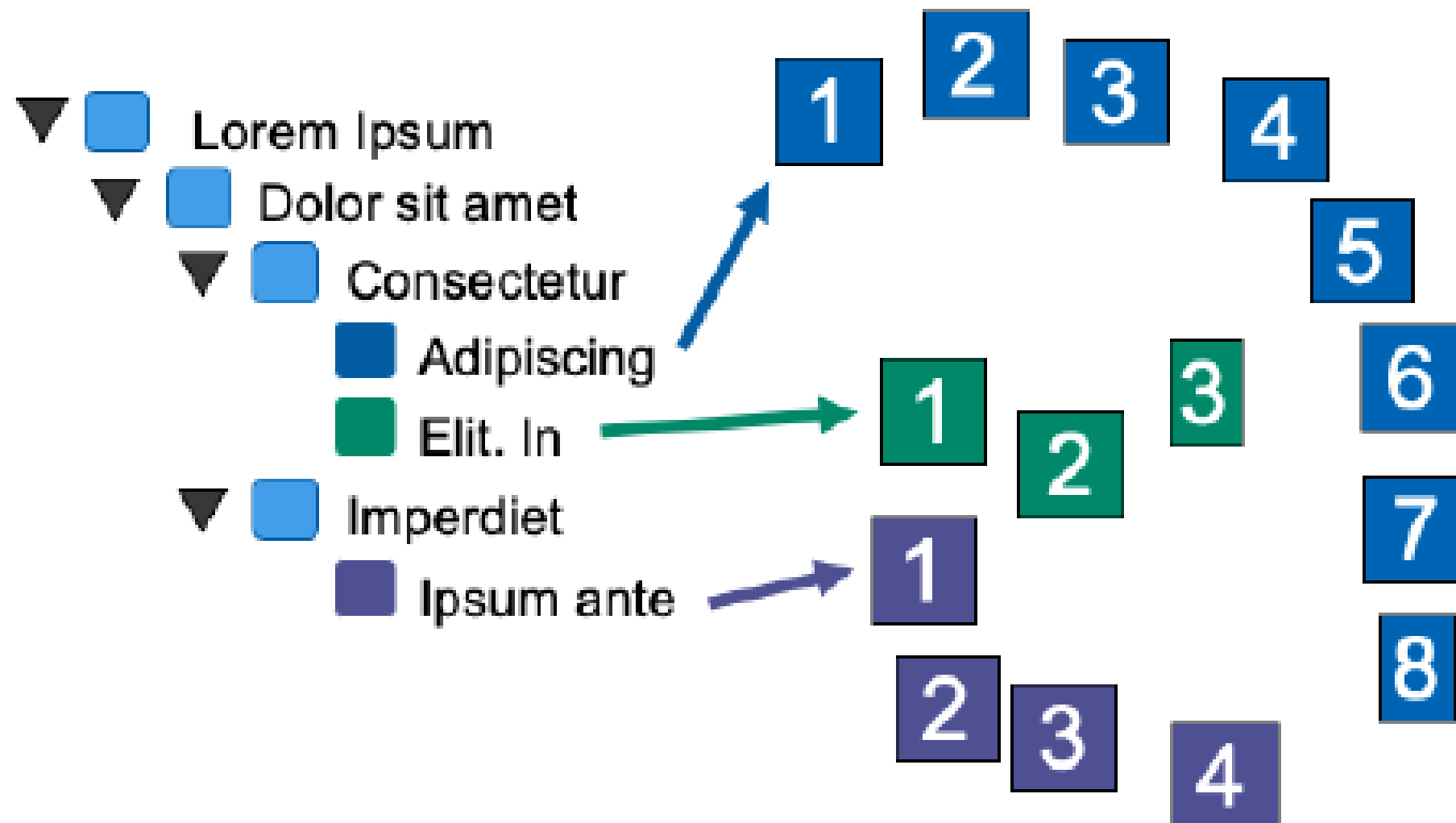
Block Storage

In Google's original file system, GFS, blocks are called chunks. In fact, throughout the course, many words will be used, in each technology, to describe partitioning data in this way: blocks, chunks, splits, shards, partitions, and so on. It is more important to understand that they are almost interchangeable, while it is secondary to learn “by heart” which technology uses which word.

Why does HDFS use blocks?



Putting it all together, the following picture summarizes best the logical model of an HDFS cluster:



The size of the blocks

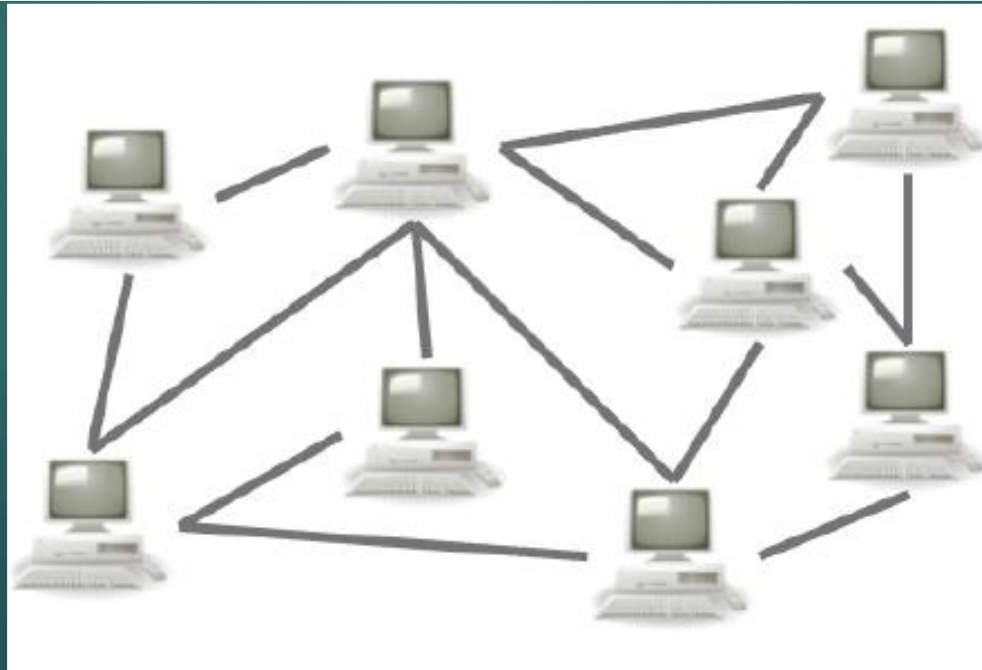
As for the block size: HDFS blocks are typically 64 MB or 128 MB large, and are thus considerably larger than blocks on a local hard drive (around 4 kB). This is because of the prerequisites of HDFS: first, it is not optimized for random access, and second, blocks will be shipped over the network.

Thus, blocks of 128 MB are large enough that time is not lost in latency waiting for a block to arrive, i.e., access to the HDFS cluster will be throughput-bound during a full-scan analysis with MapReduce or Spark. And at the same time, they are small enough for a large file to be conveniently spread over many machines, allowing for parallel access, and also small enough for a block to be sent again without too much overhead in case of a network issue.

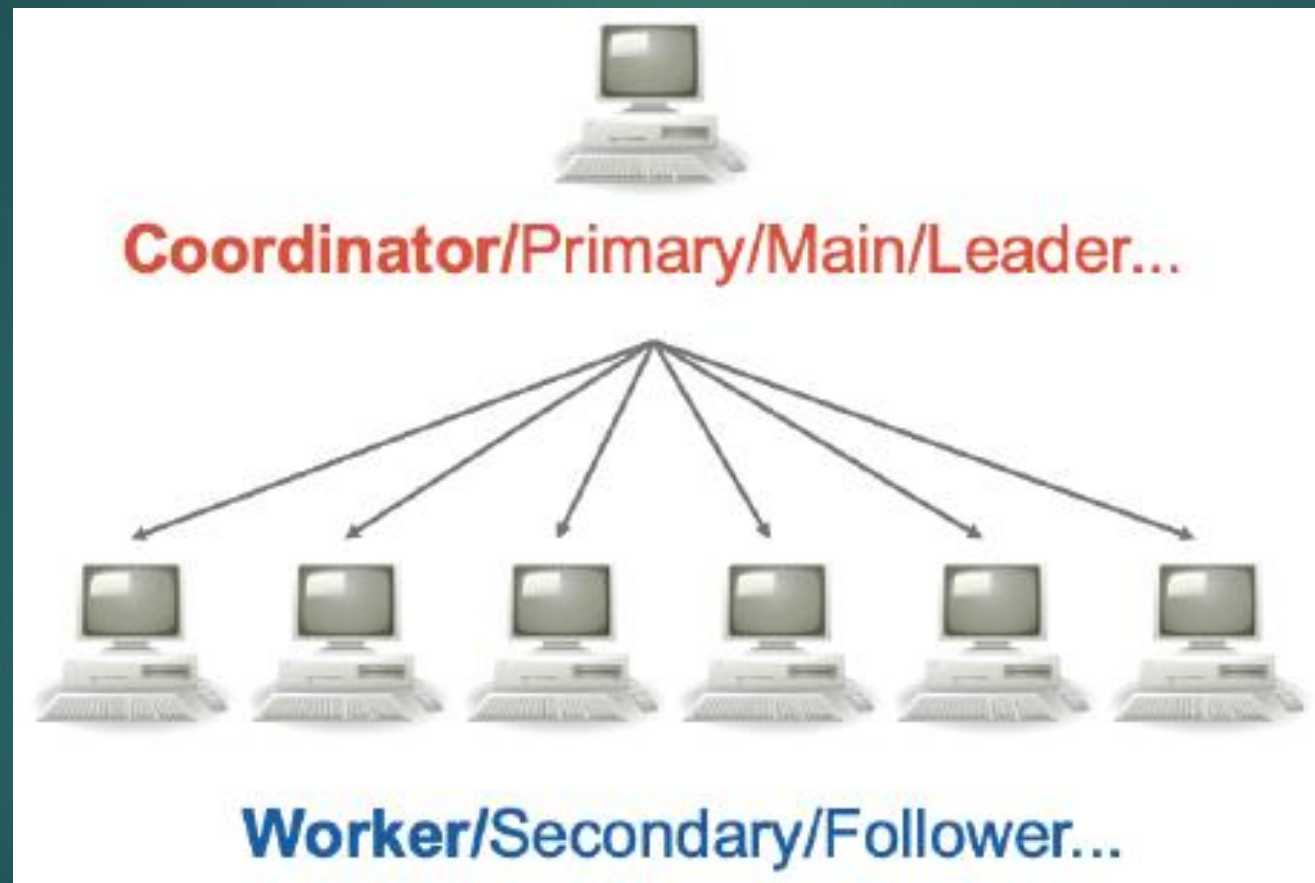
Physical architecture

HDFS is designed to run on a cluster of machines. The number of machines can range from just a handful of them to thousands of machines. How are they connected?

One way to connect machines is called a peer-to-peer, decentralized network. In a peer-to-peer network, each machine talks to any other machine. This architecture, for example, is used in the Bitcoin blockchain and was also popular in the 1990s with the Napster network.



By contrast, HDFS is on the contrary implemented on a fully centralized architecture, in which one node is special and all others are interchangeable and connected to it.

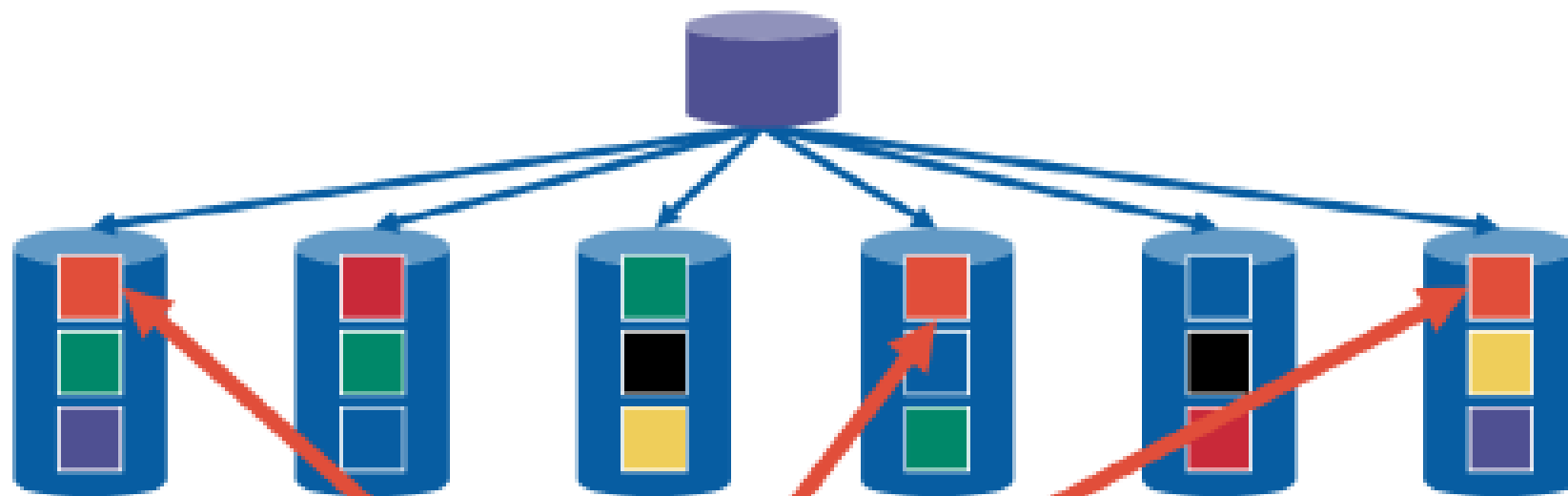


The terminology used to describe the nodes in a centralized network architecture varies in the literature. In the case of HDFS, the central node is called the NameNode and the other nodes are called the DataNodes. In fact, more precisely, the NameNode and DataNodes are processes running on these nodes, and the CamelCase notation often used to write their names down corresponds to the Java classes implementing these processes. By metonymy, we also use these names to describes the nodes themselves.

Now that we have the NameNode and DataNodes in place, how is the logical HDFS model implemented on top of this architecture?

As we saw before, every file is divided into chunks called blocks. All blocks have a size of exactly 128 MB, except the last one which is usually smaller (indeed, what are the odds that the file size would be exactly a multiple of 128 MB?).

Namenode



... replicated for fault tolerance



...divided into 128MB chunks...

File...

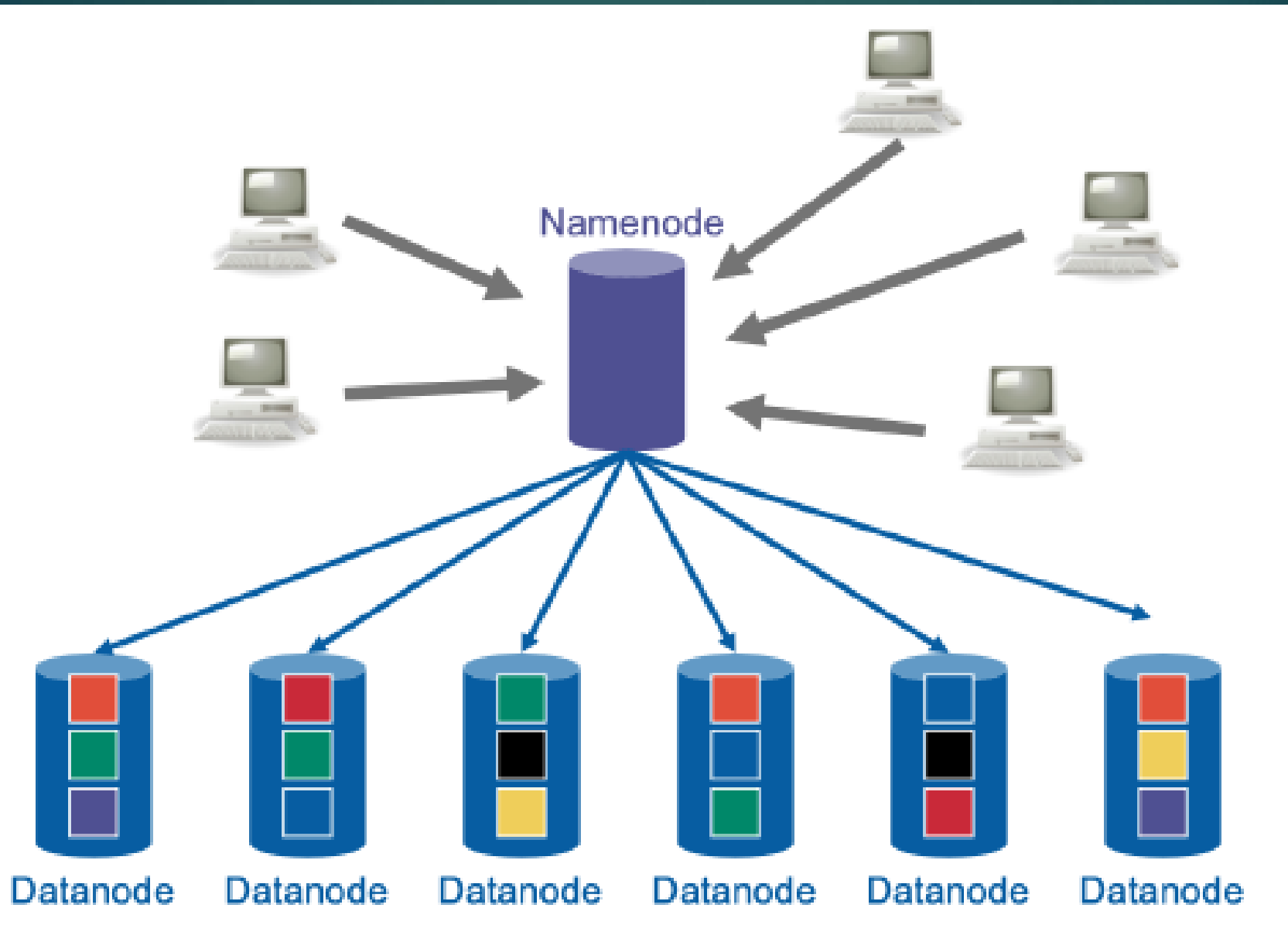


Each one of the blocks is then replicated and stored on several DataNodes. How many times? This is a parameter called the *replication factor*. By default, it is 3, but this can be changed by the user.

There is no such thing as a primary replica: all replicas are on an equal footing. In other words, by default, three replicas of each block are stored by default. Overall, in a production instance, there is a very high number of block replicas and they are spread more or less evenly across the nodes, to have a certain balance.

An HDFS cluster is then accessed concurrently by many clients. Typically, the cluster is owned by one company or organization and centrally managed by a specialized team (e.g., site reliability engineers).

The clients correspond to users within this organization who read from and write to the HDFS cluster.



The responsibilities of the NameNode

The NameNode is responsible for the system-wide activity of the HDFS cluster. It stores in particular three things:

- the file namespace, that is, the hierarchy of directory names and file names, as well as any access control (ACL) information similar to Unix-based systems.
- a mapping from each file to the list of its blocks. Each block, in this list, is represented with a 64-bit identifier; the content of the blocks is not on the NameNode.
- a mapping from each block, represented with its 64-bit identifier, to the locations of its replicas, that is, the list of the DataNodes that store a copy of this block.

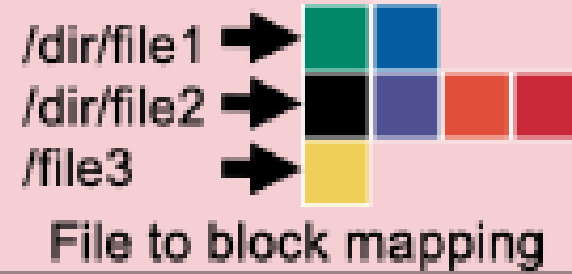
Memory

1

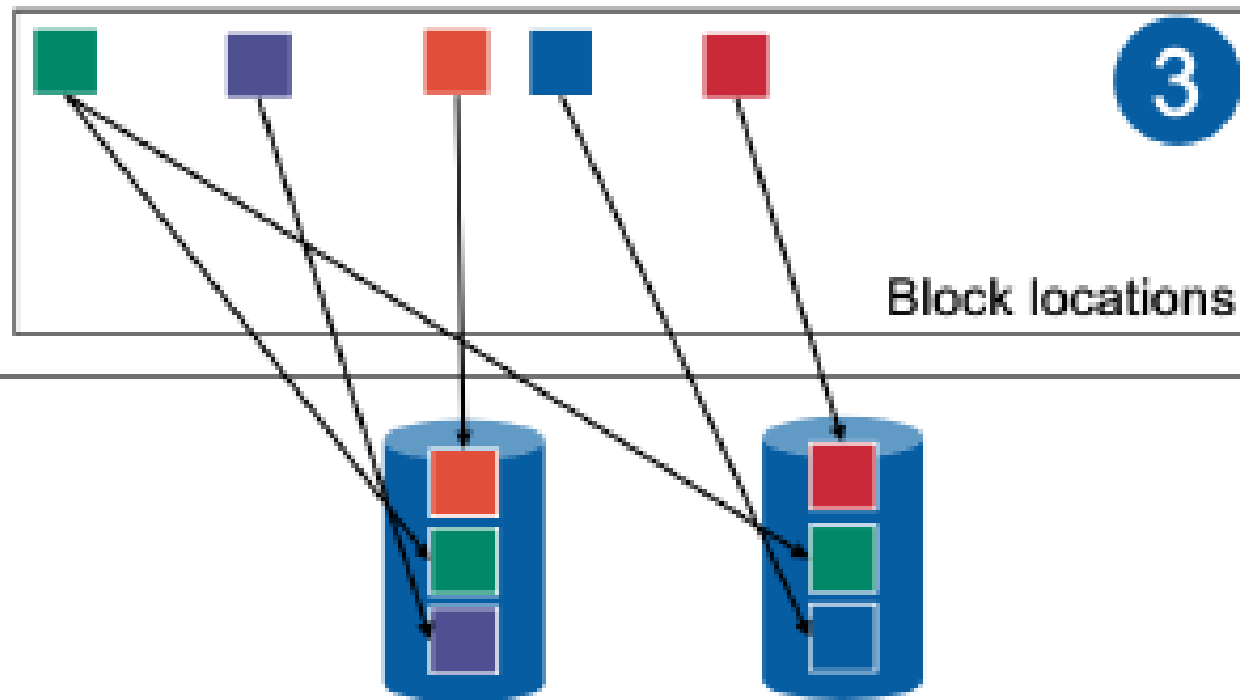


File namespace
(+Access Control)

2



3



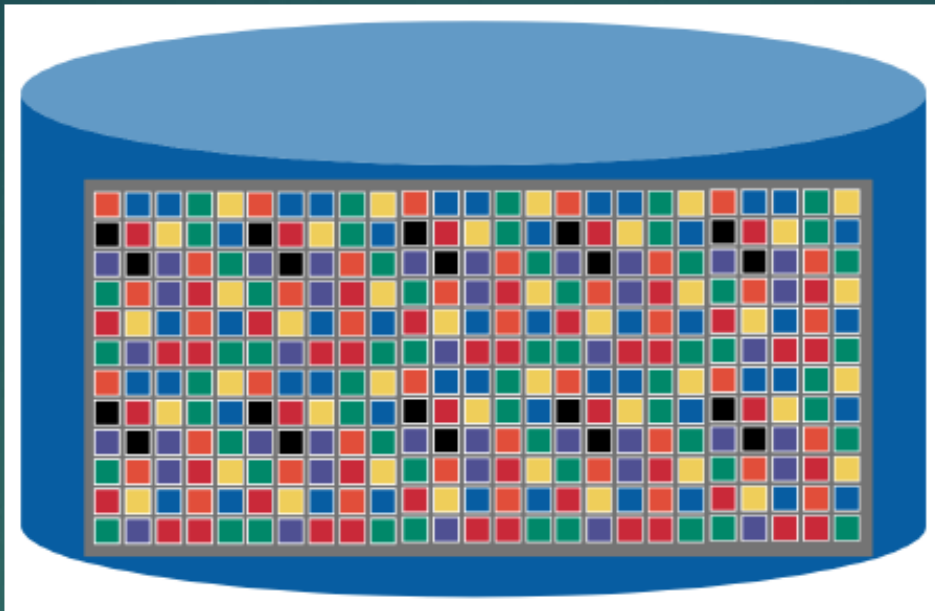
The NameNode updates this information whenever a client connects to it in order to update files and directories, as well as with the regular reports it receives from the DataNodes.

Clients connect to the NameNode via the Client Protocol. Clients can perform metadata operations such as creating or deleting a directory, but also ask to delete a file, read a file or write a new file. In the latter case, the NameNode will send back to the client block identifiers (for reading them), or lists of DataNode locations (for reading and writing them).

The responsibilities of the DataNode

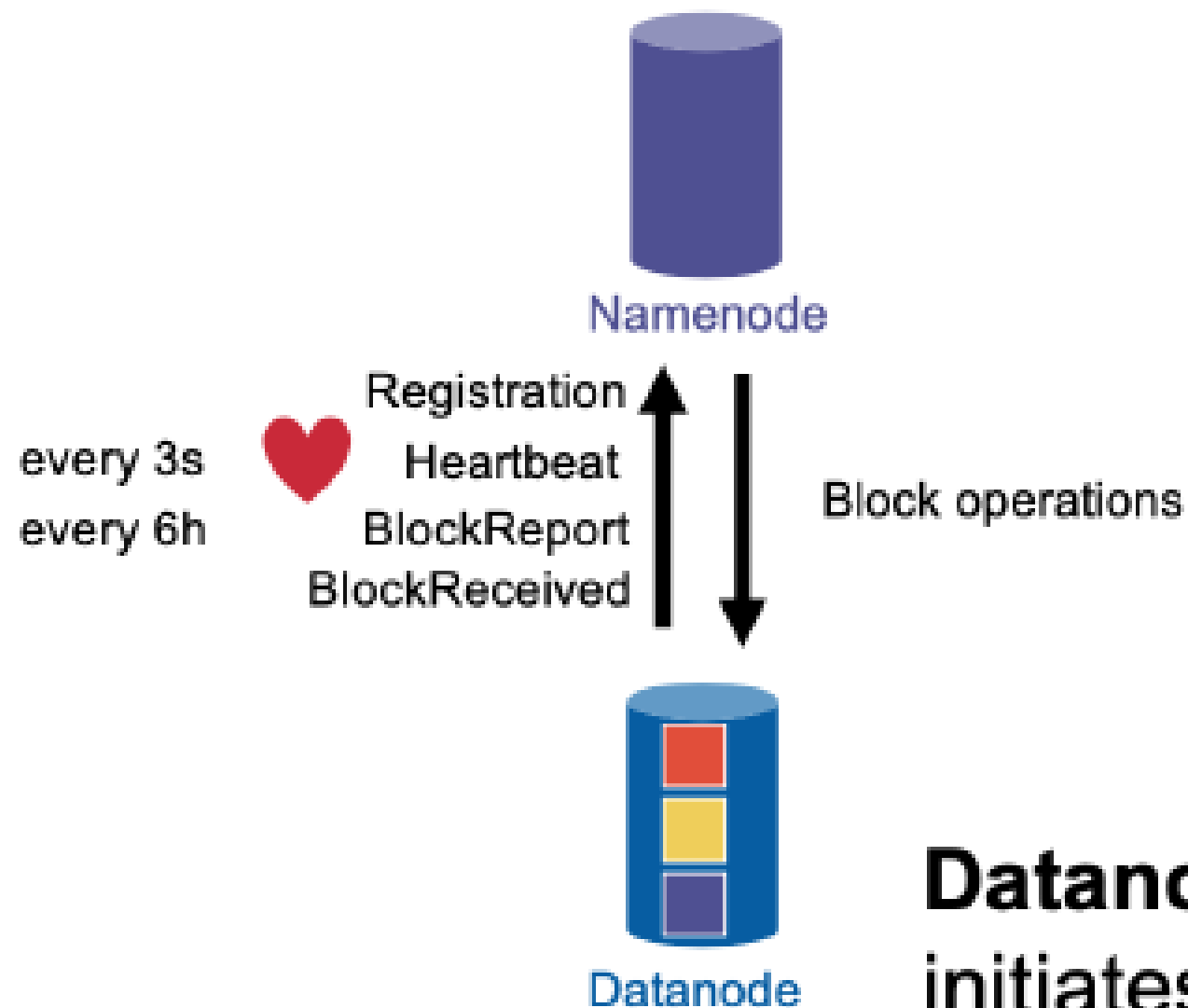
The DataNodes store the blocks themselves. These blocks are stored on their local disks¹. Each block is stored as a 128 MB local, physical file on the DataNode. If the block is the last block of an HDFS file and thus less than 128 MB, then the physical file has exactly the size of the block: there is no waste of space. This is different from the physical blocks (4 kB or so) of a local hard drive, which have a constant size.

¹In a cloud installation, these local disks can also be virtually attached, but this is the same from the perspective of the DataNode process.



ma btejeh bel fahes

DataNodes send regular heartbeats to the NameNode. The frequency of these heartbeats is configurable and is by default a few seconds (e.g., 3s, but this value may change across releases). This is a way to let the NameNode know that everything is alright. The heartbeat may also contain a notification to the NameNode that a new block was received and successfully stored, and that the DataNode can thus be used as a location for this block. Finally, the DataNode also sends, every couple of hours (e.g., 6h, but this value may change across releases), a full report including all the blocks that it contains.

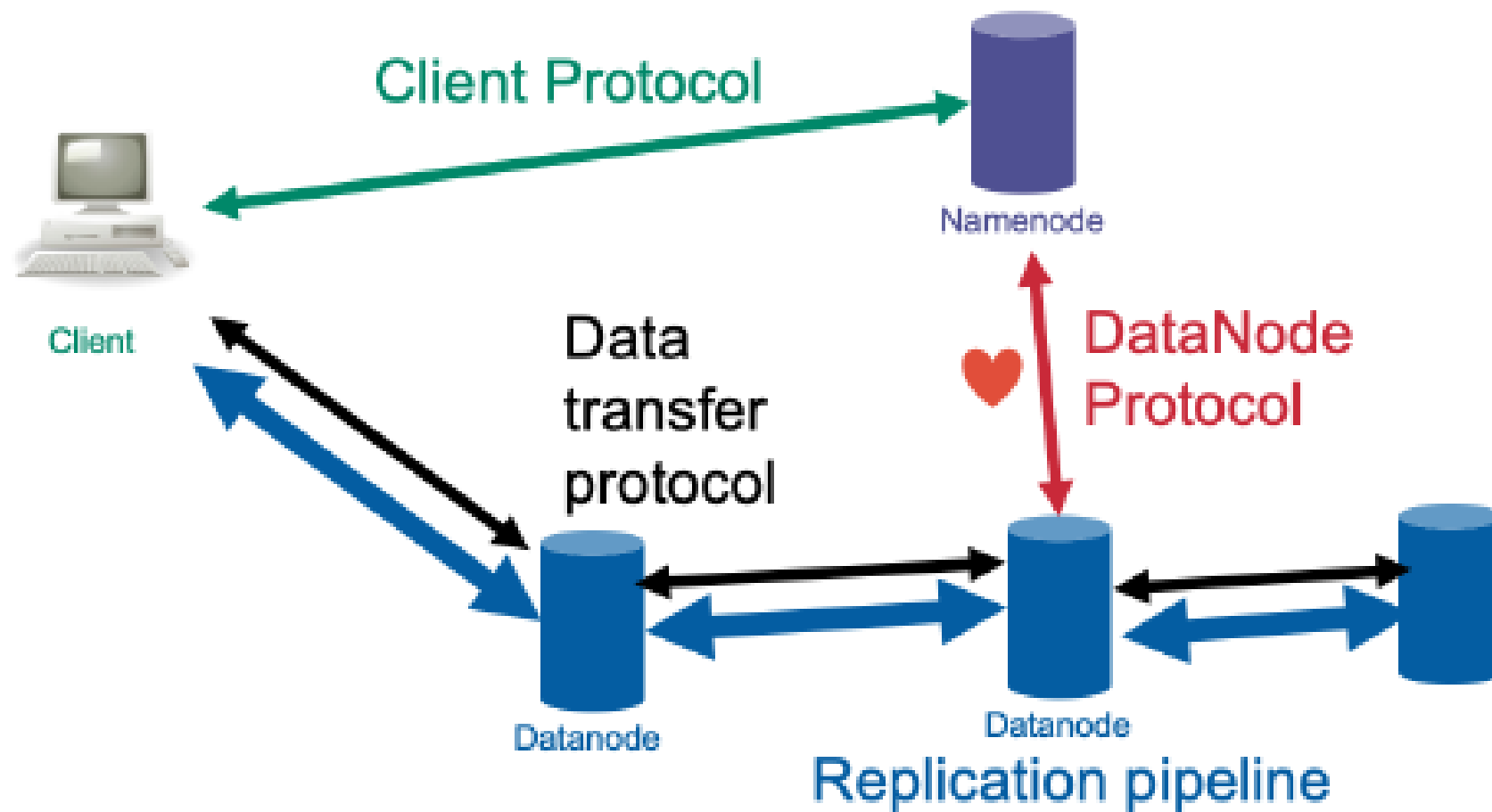


Datanode always
initiates connection!

If there is an issue with the local disk or the node, then the DataNode can report the block as corrupted to the NameNode, which will then ensure, asynchronously, its replication to somewhere else. “Asynchronously,” as opposed to “synchronously,” means that this is done in the background at some later time and the DataNode does not idly wait for this to happen. In the meantime, the block is simply marked as underreplicated.

A NameNode never initiates a connection to a DataNode. If the NameNode needs anything from a DataNode, for example, if it needs to request a DataNode to download an underreplicated block from another DataNode and store a new replica of it, then the NameNode will wait until the next heartbeat, and answer to it with this request.

Finally, DataNodes are also capable of communicating with each other by forming replication pipelines. A pipeline happens whenever a new HDFS file is created. The client does not send a copy of the block to all the destination DataNodes, but only to the first one. This first DataNode is then responsible for creating the pipeline and propagating the block to its counterparts.



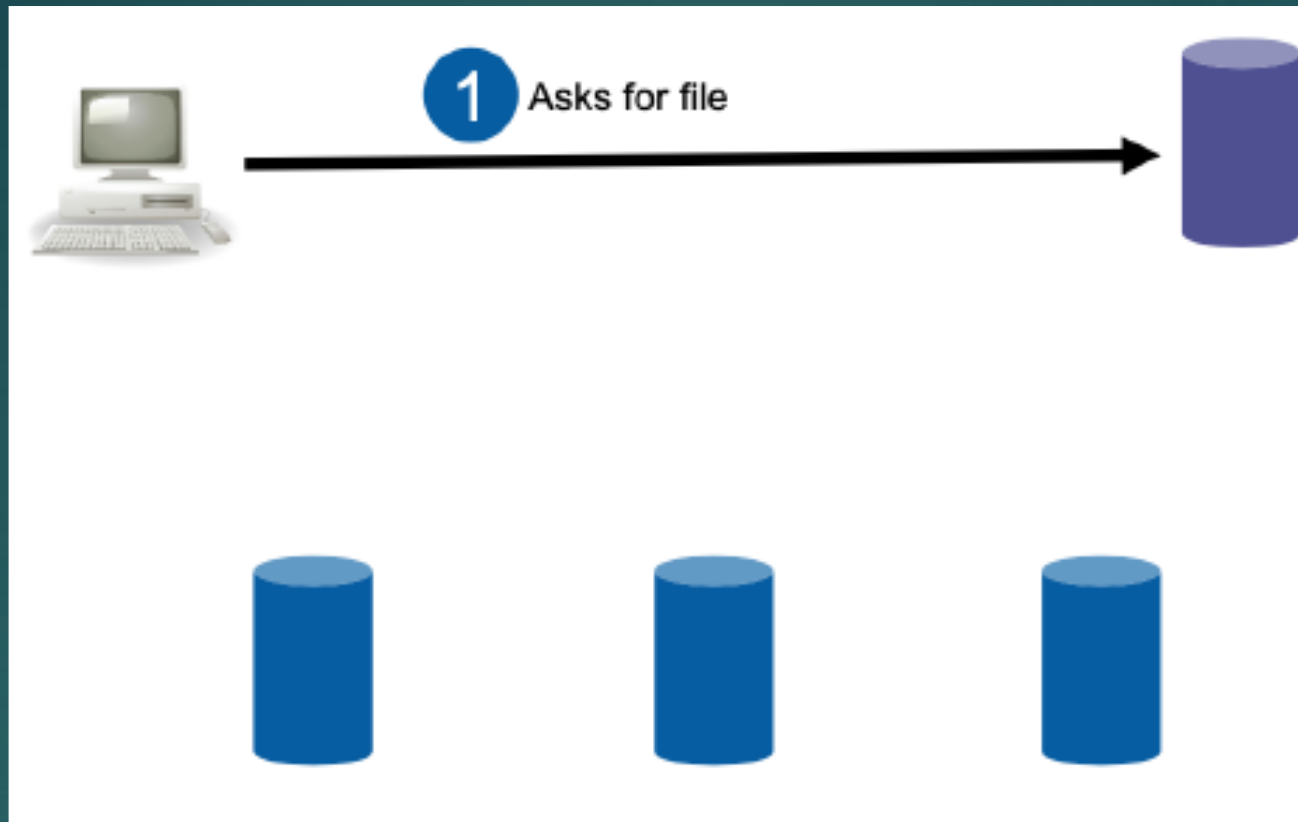
When a replication pipeline is ongoing and a new block is being written to the cluster, the content of the block is not sent in one single 128 MB packet. Rather, it is sent in smaller packets (e.g., 64 kB) in a streaming fashion via a network protocol. That way, if some packets are missing, the client can send them again. The client receives the acknowledgements in a streaming fashion as well.

File system functionality

HDFS exposes to the client, via the NameNode, an API that allows typical operations available on a file system: creating a directory, deleting a directory, listing the contents of a directory, creating a file, appending to a file, reading a file, deleting a file.

Many of these involve the NameNode only. However, reading, writing and deleting a file involves communication across the cluster, which we now detail.

Let us start with reading a file. The client first connects to the NameNode to initiate the read and request info on the file.



Then, the NameNode responds with the list of all blocks, as well as a list of DataNodes that contains a replica of each block. The DataNodes are furthermore sorted by increasing distance of the client (which is typically itself one of the nodes in the cluster, we will come back to this).



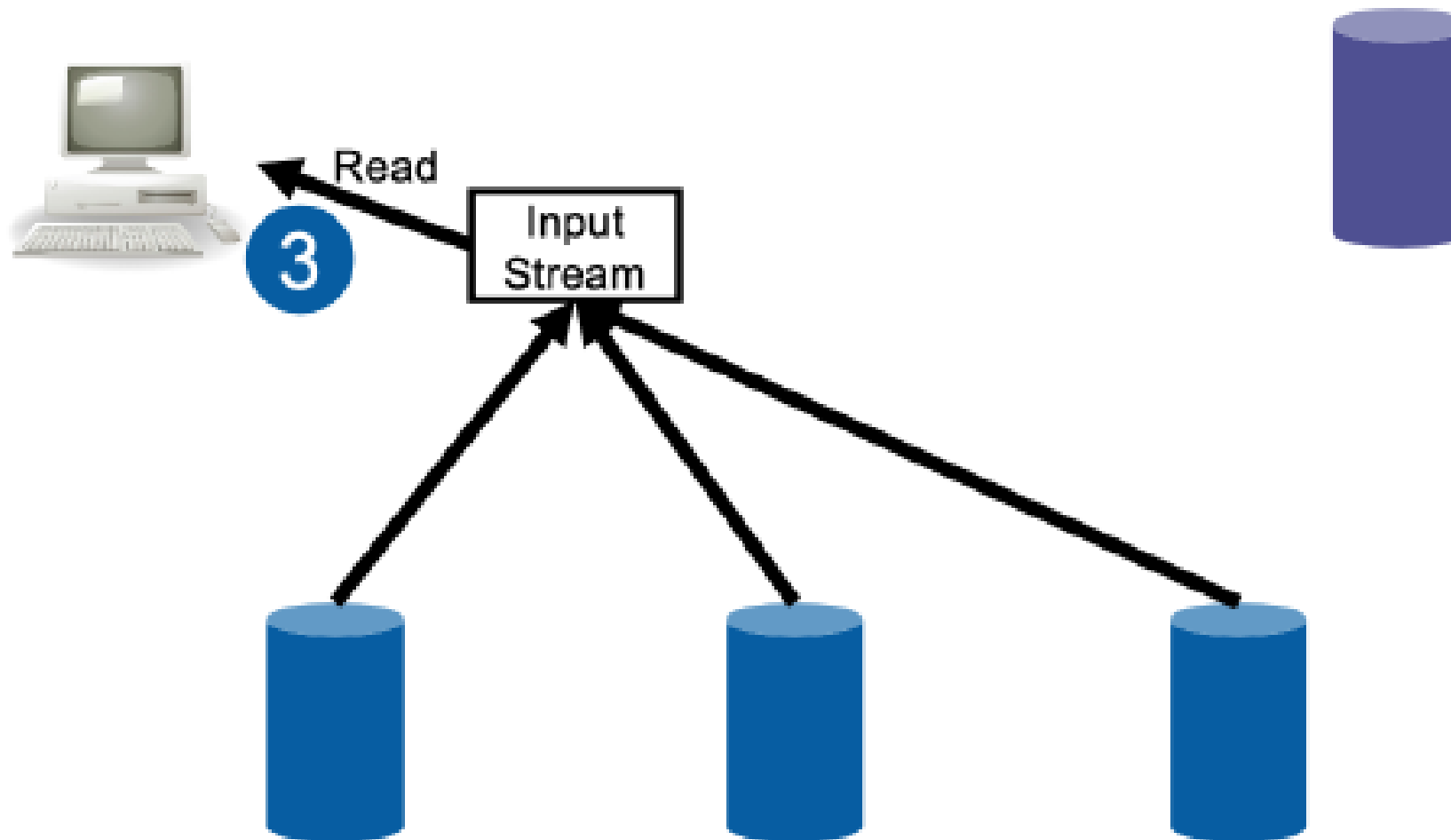
2



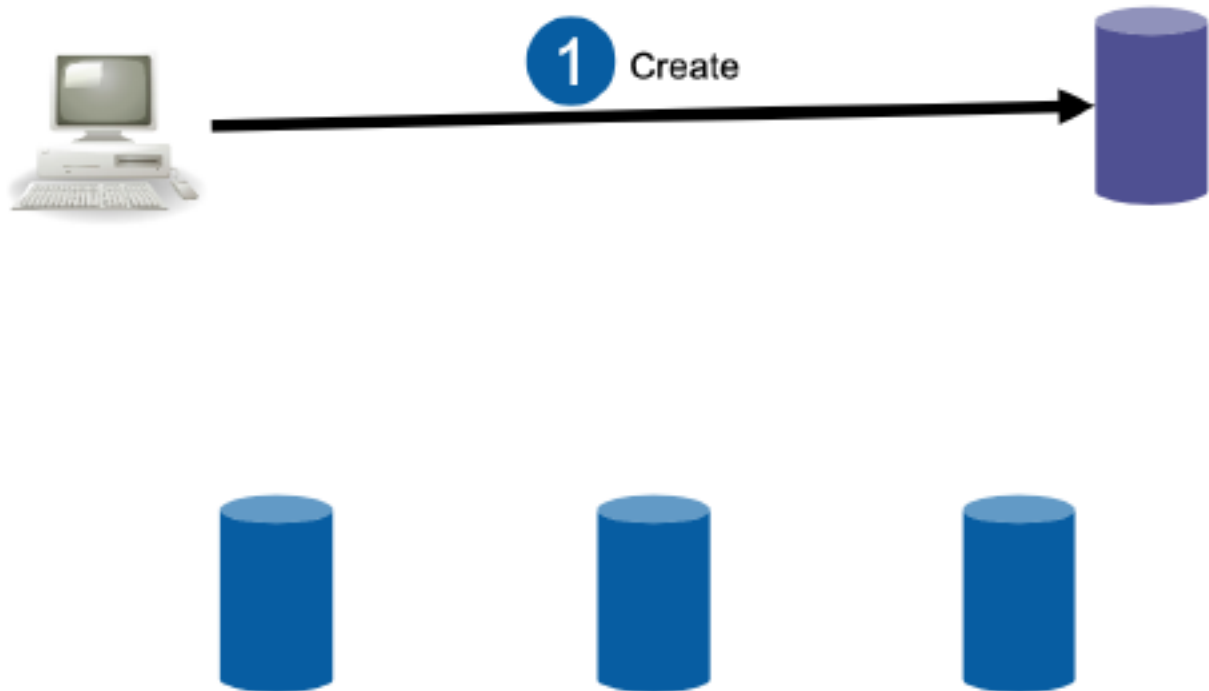
Get block locations
Multiple DataNodes for each block,
sorted by distance



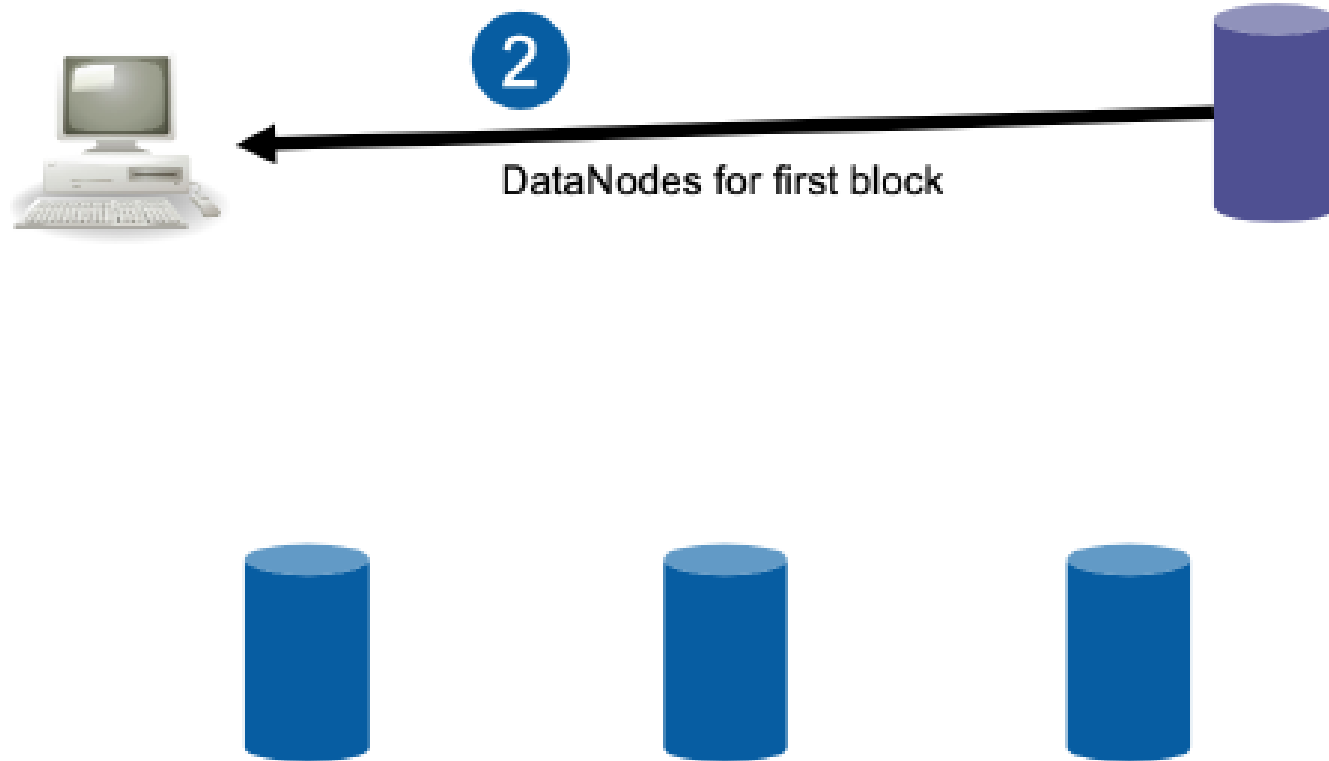
The client then connects to the DataNodes in order to download a copy of the blocks. It starts with the first (closest) DataNode in the list provided by the NameNode for each blocks, and will go down the list if a DataNode cannot be reached or is not available. In the simple case that the client wants to stream its way through an HDFS file, bit by bit, it will download each block in turn. However, this functionality is typically nicely encapsulated in additional, client-side Java libraries that expose this as a "single big input stream" with the InputStream classes familiar to Java programmers. Switching between the DataNodes is hidden and the user just sees a stream of bits flowing through. Note that with streaming, it is possible to process files larger than the working memory, because older blocks can be thrown away from the memory of the client once processed. This is something we will come back to when we look into the architecture of querying engines. Alternatively, the client can also download a multi-block HDFS file and store it as a single big file on its local drive as long as it fits.



Let us now write a new file to HDFS. This can be either a simple upload of a large local file, or it can be from a stream of bits created on the fly by a program. As for reading, the client first connects to the NameNode formulating its intent to create a new file.



The NameNode then responds with a list of DataNodes to which the content of the first block should be sent. Note that, at that point, the file is not yet guaranteed to be available for read for other clients², and it is locked in such a way that nobody else can write to it at the same time.

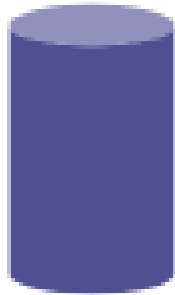
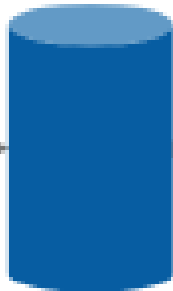
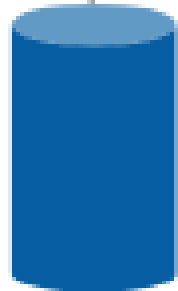


The client then connects to the first DataNode and instructs it to organize a pipeline with the other DataNodes provided by the NameNode for this block.

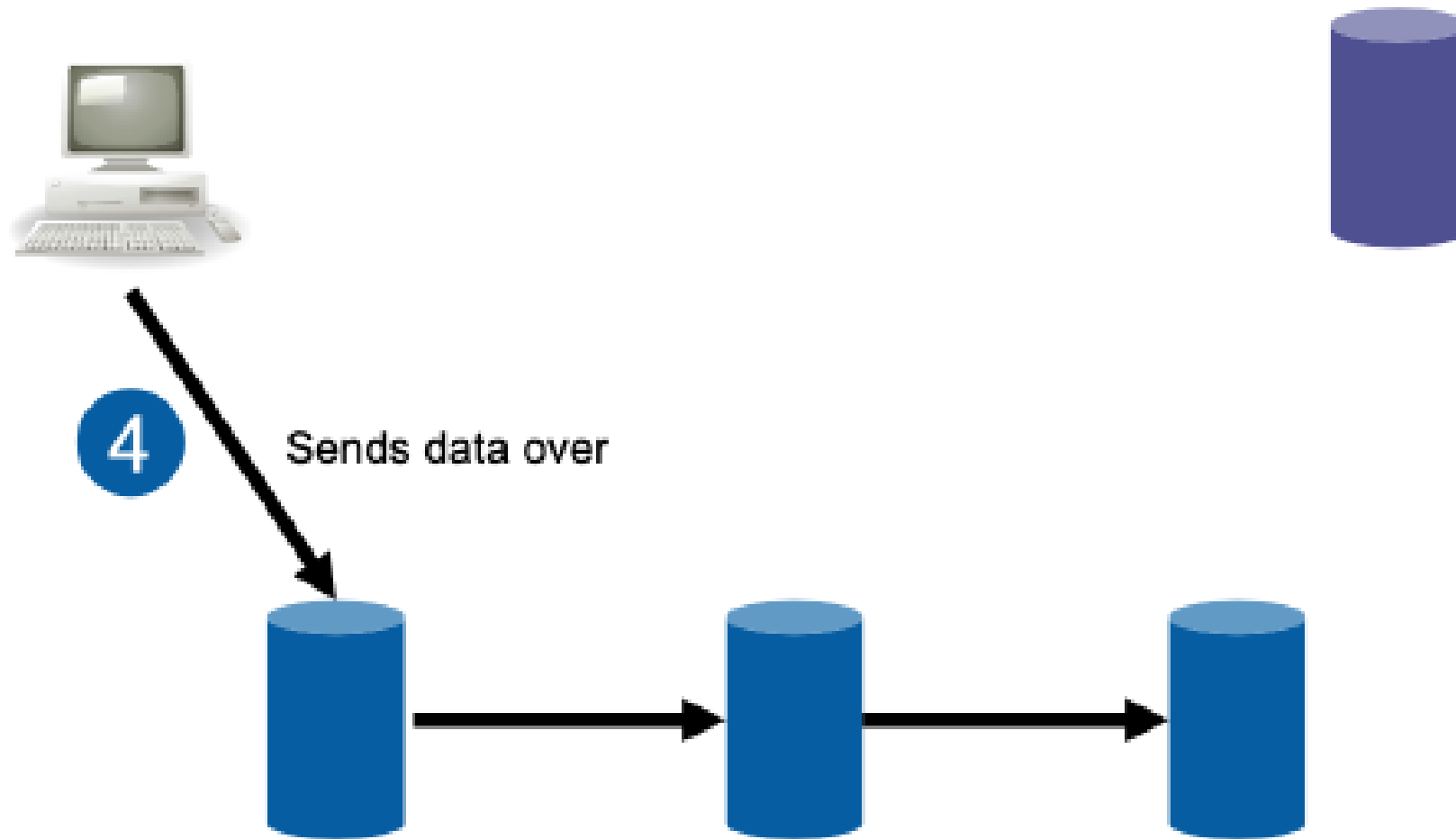


3

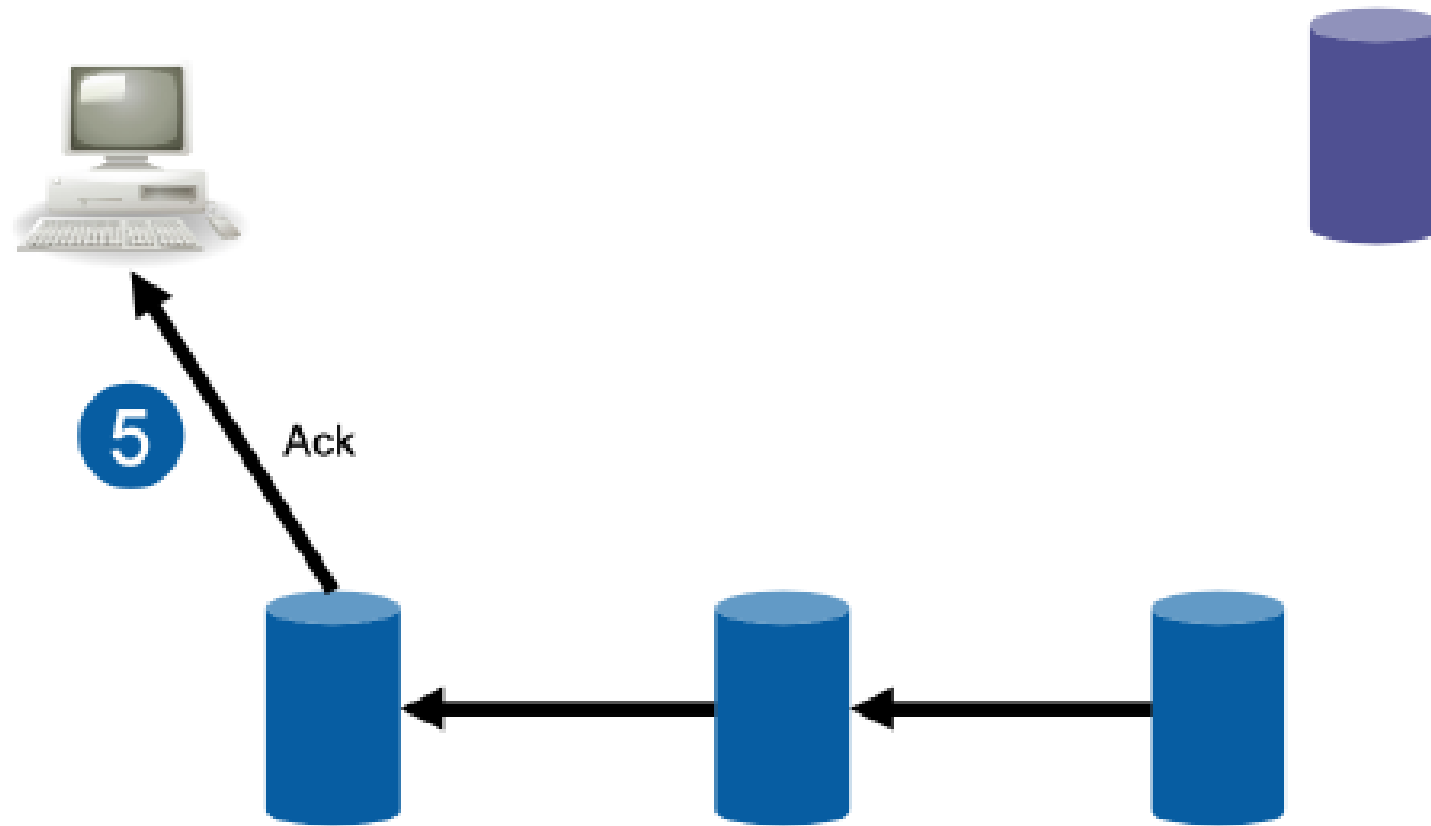
Organizes pipeline



The client then starts sending through the content of that block, as we explained earlier. The content will be pipelined all the way to the other DataNodes.



The client receives regular acknowledgements from the first DataNode that the bits have been received.



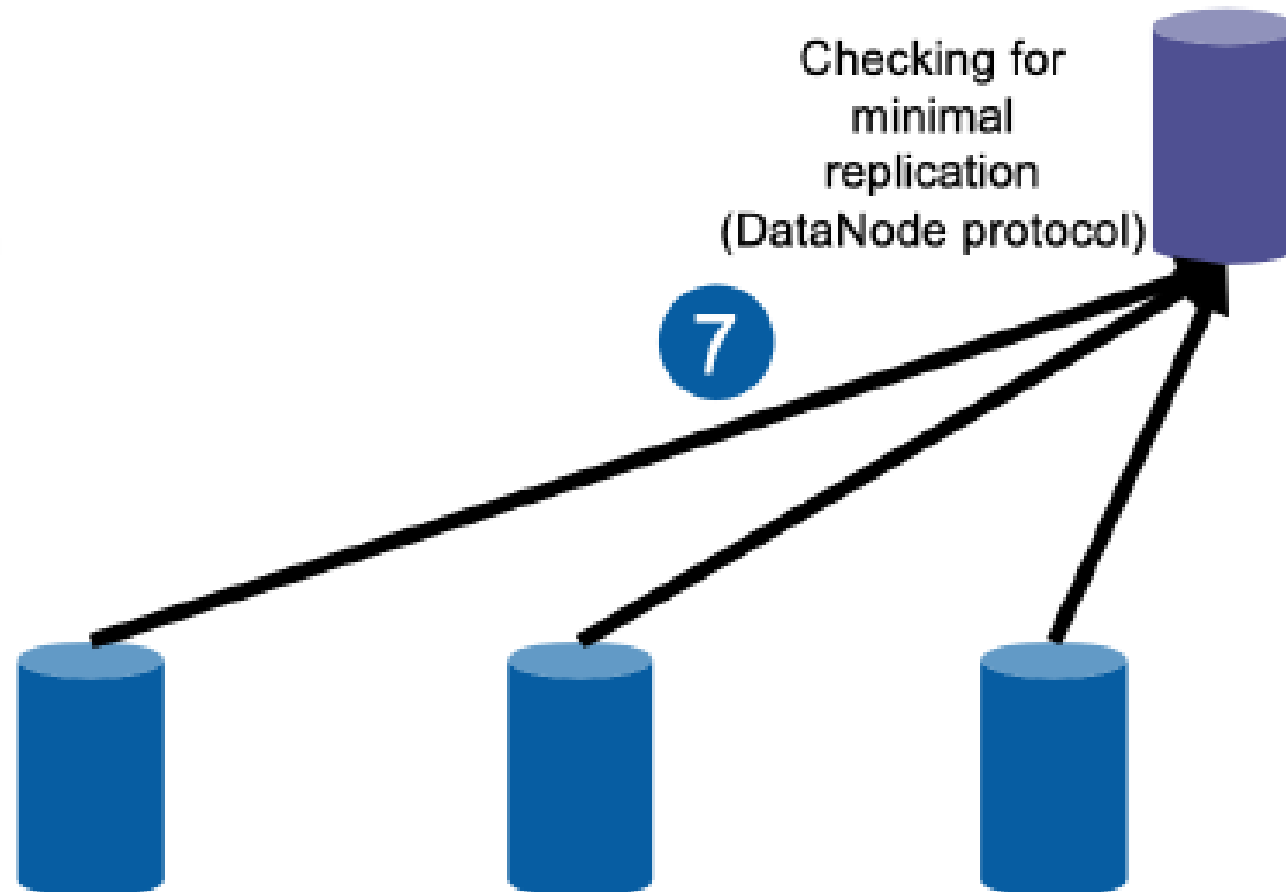
When all bits have been acknowledged, the client connects to the NameNode in order to move over to the second block. Then, the same steps (2, 3, 4, 5) as before are repeated for each block.



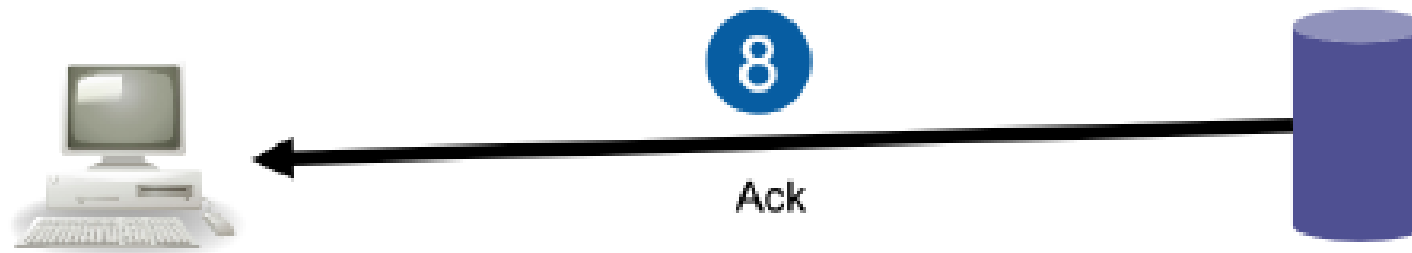
Once the last block has been written, the client informs the NameNode that the file is complete, and asks to release the lock.



The NameNode then checks for minimal replication through the DataNode protocol.



... and gives its final acknowledgement to the client.



From now on, and separately (this is called “asynchronous”), the NameNode will continue to monitor the replicas and trigger copies from DataNode to DataNode whenever necessary, that is, when a block is underreplicated.



replicates further asynchronously

9

