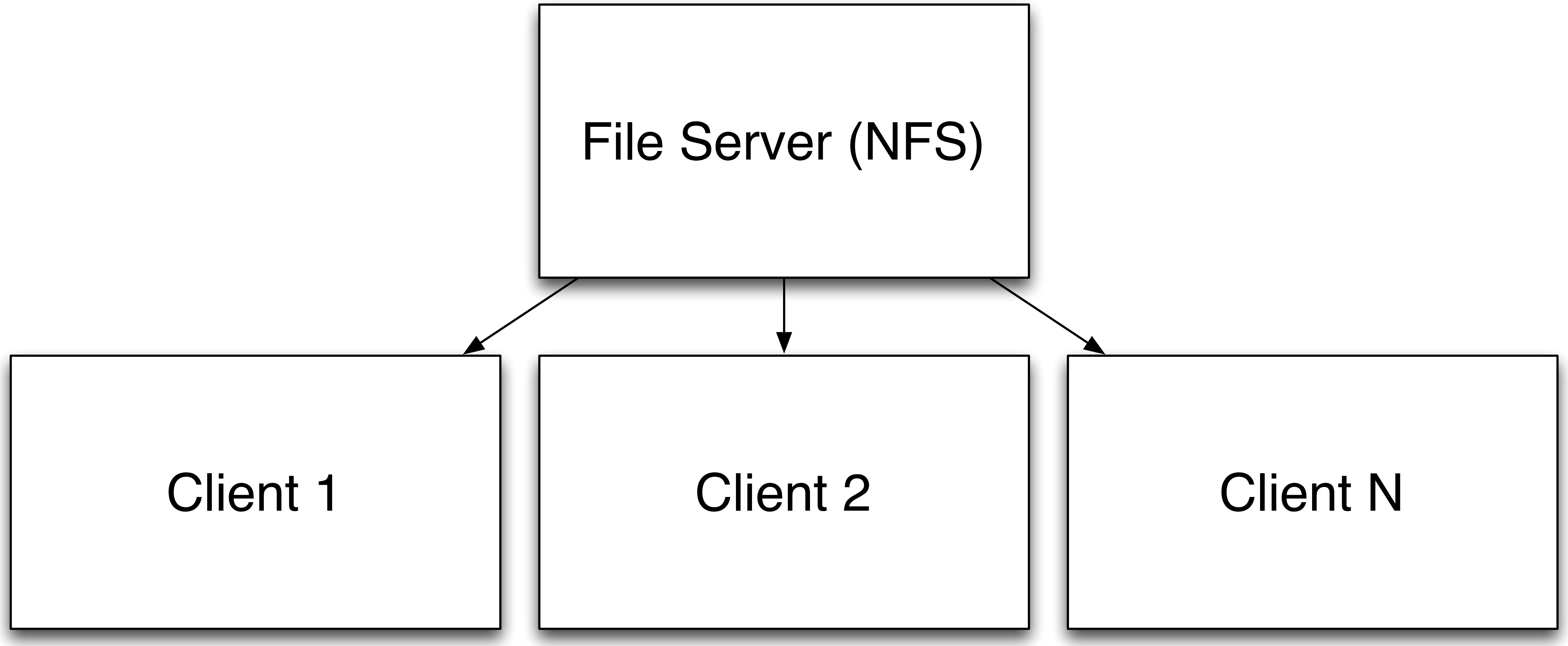
Hadoop Distributed Filesystem

28-01-2016
Jeroen Schot - jeroen.schot@surfsara.nl
Mathijs Kattenberg - mathijs.kattenberg@surfsara.nl
Machiel Jansen - machiel.jansen@surfsara.nl

Let’s store data like its 1999



Traditional shared storage in clusters

Problems with storage server

- File server is a single point of failure
- Scale-up has hard limits (both in capacity and IO)
- Scale-out complex due to filesystem semantics (editing, locking)

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of

1. INTRODUCTION

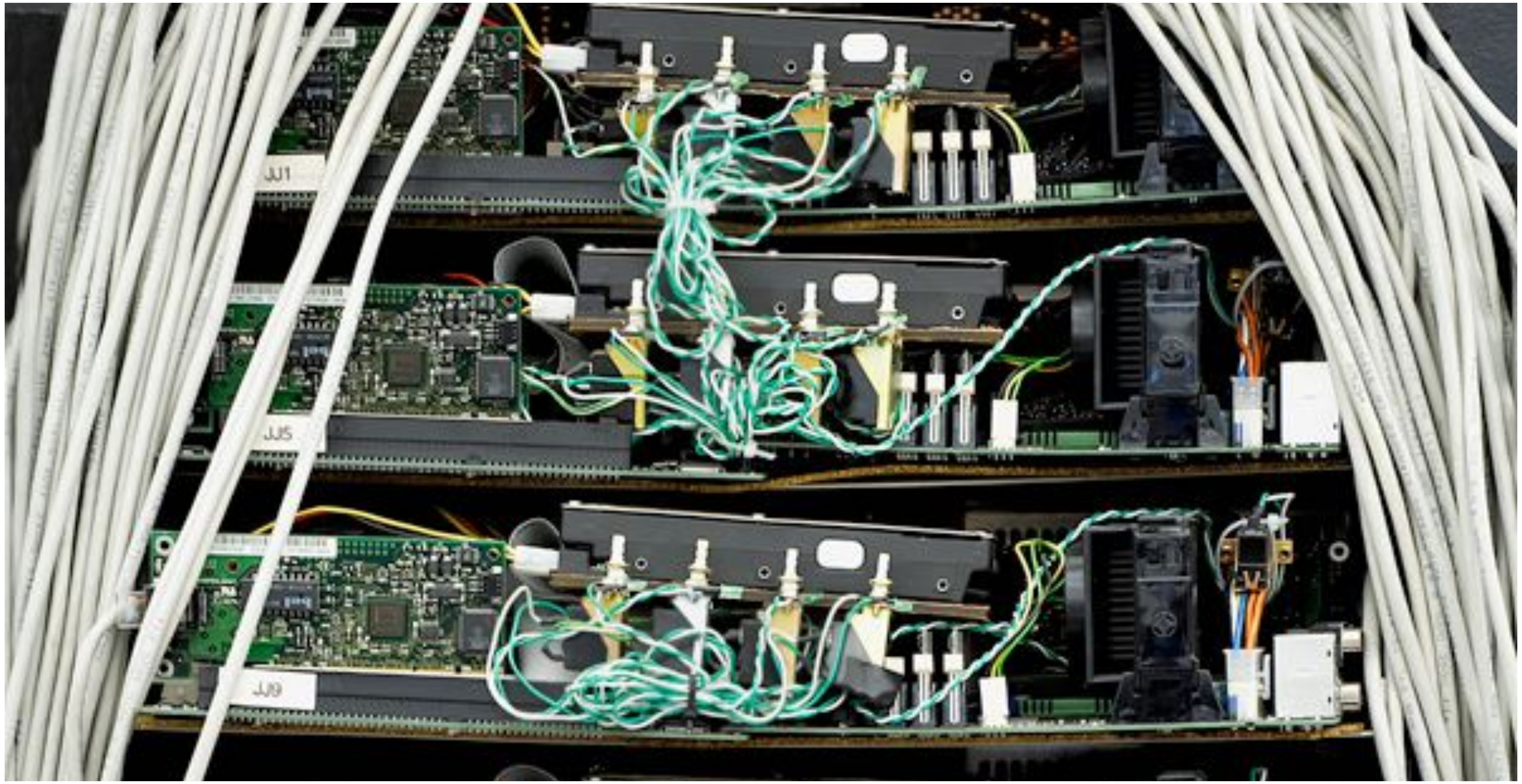
We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client

GFS - design overview

- Handles failure of individual nodes
- Optimised for large (100+MB) files
- Optimised for sequential reads
- Favours high-throughput over low-latency

Inexpensive components…



SURF SARA

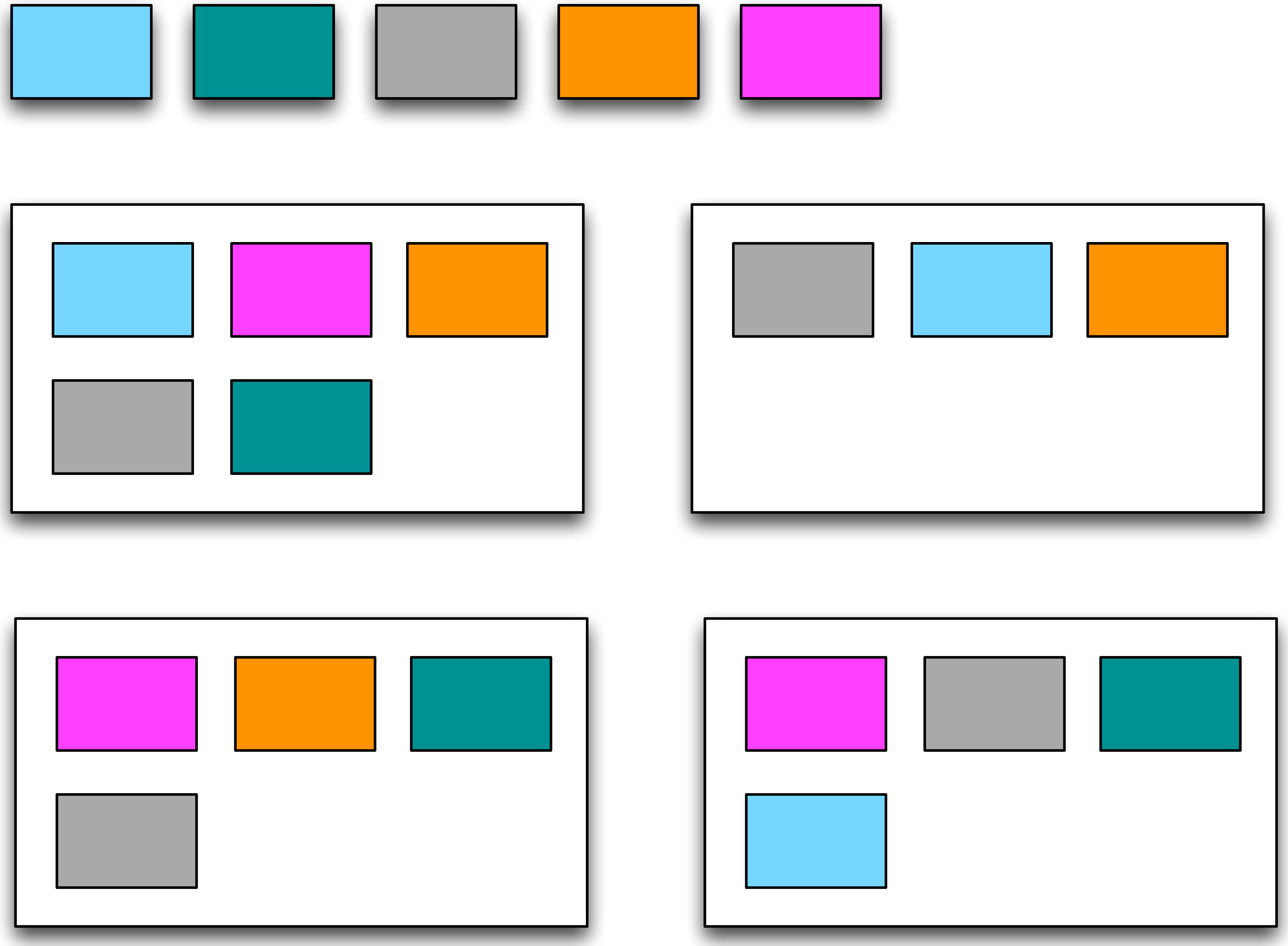
GFS - architecture

- Files are split in 128MB blocks
- Blocks are stored on many datanodes
- Each block is stored 3 times (on different nodes)
- Single namenode handles metadata (namespace, block locations)
- Clients connect directly to datanodes

Hadoop Distributed File System

- Storage component of Apache Hadoop
- First released in 2005
- HDFS started as a open-source implementation of GFS

Files, Blocks & Replicas



Namenode

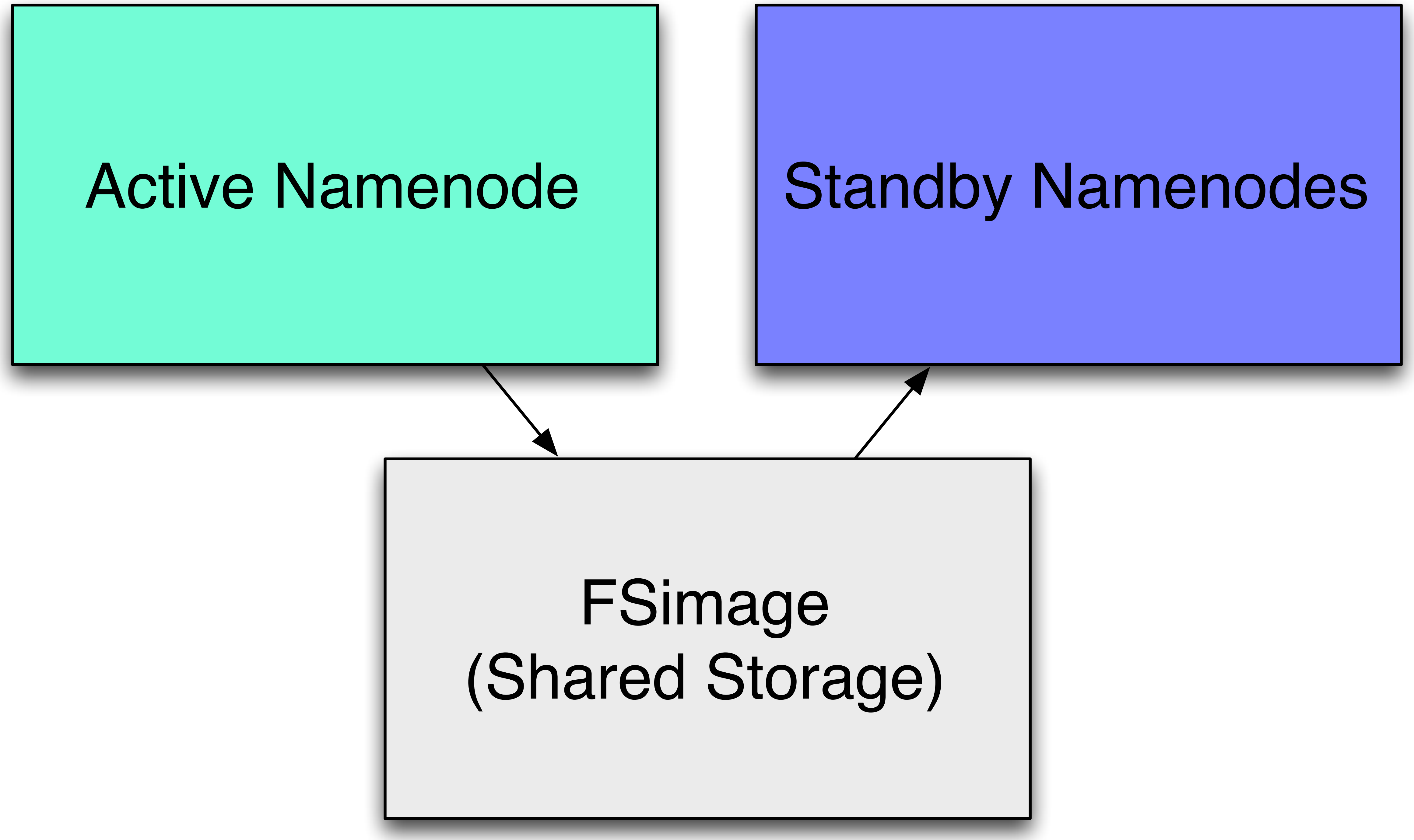
Maintains the mappings of

- files to blocks
- blocks to datanodes

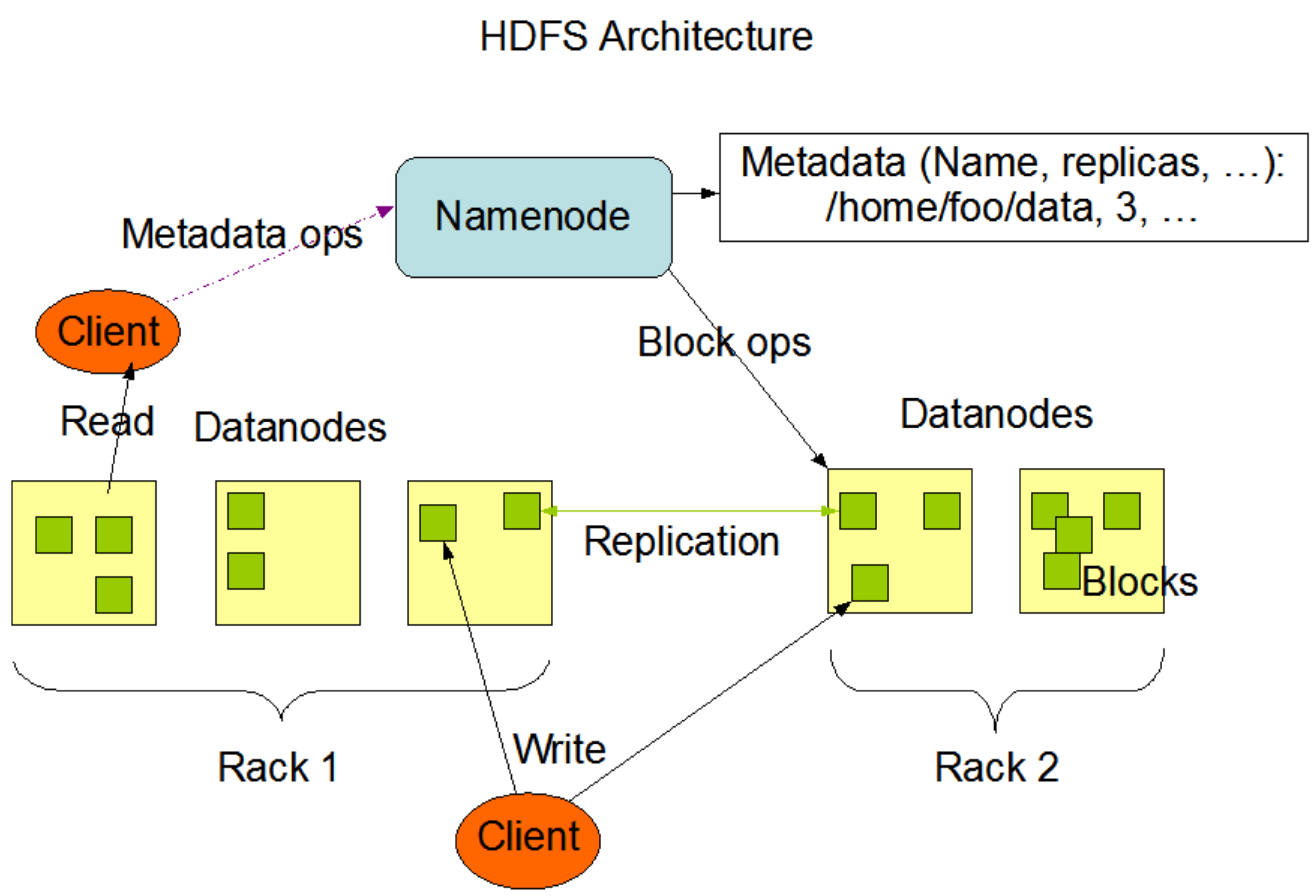
Monitors datanode health

Enforces block replica count

Namenode High Availability



Read/write scaling



Data locality

- In Hadoop the same machines are often used for both storage and compute
- The YARN scheduler takes data location into account: tries to schedule tasks on the same machine as the data

Using HDFS

HDFS is not a filesystem you mount*

Interaction is done via:

- Native Java API
- hdfs command-line utility
- WebHDFS REST API
- Third-party libraries for other languages (Python)

Java API

```
/* Instantiate reference to HDFS filesystem */
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);

/* List files in directory */
FileStatus[] stats = fs.listStatus(new Path("/some/path/"));
for (FileStatus stat : stats) {
  System.out.println(stat.getPath());
}

/* Move a file */
fs.rename(new Path("/old/file/name"), new Path("/new/file/name"));
```

SURF SARA

```
$ hdfs dfs
Usage: hadoop fs [generic options]
  [-cat [-ignoreCrc] <src> ...]
  [-checksum <src> ...]
  [-chgrp [-R] GROUP PATH...]
  [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
  [-chown [-R] [OWNER][:[GROUP]] PATH...]
  [-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
  [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
  [-count [-q] [-h] <path> ...]
  [-cp [-f] [-p | -p[topax]] <src> ... <dst>]
  [-find <path> ... <expression> ...]
  [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
  [-help [cmd ...]]
  [-ls [-d] [-h] [-R] [<path> ...]]
  [-mkdir [-p] <path> ...]
  [-mv <src> ... <dst>]
  [-put [-f] [-p] [-l] <localsrc> ... <dst>]
  [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
  [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
  [-setrep [-R] [-w] <rep> <path> ...]
  [-tail [-f] <file>]
  [-test -[defsz] <path>]
  [-text [-ignoreCrc] <src> ...]
```

SURF SARA

Object stores

- Scalable storage even simpler than GFS/HDFS
- Access via HTTP REST interface
- Put/get/delete, no edit/append
- Many different implementations (Amazon S3, OpenStack Swift, Azure Blob Storage, Google Cloud Storage, …)

Flat namespace

- No filesystem hierarchy with directories and subdirectories
- Only containers and objects:

http://swift.example.com/v1/myaccount/mycontainer/someobject

http://swift.example.com/v1/myaccount/mycontainer/anotherobject

- Filesystem-like view faked with object names (“some/object/name”)

SURF SARA

Fully distributed

- Storage location is derived from the container/object name
- No central namenode needed (better availability)
- But sometimes a client reads stale data (eventual consistency)