Centre de Calcul
de l’Institut National de Physique Nucléaire
et de Physique des Particules

DU Data Science 2020
Storage overview

Loïc Tortay, tortay@cc.in2p3.fr

Units for data storage & transfer

- Bytes for data storage, ISO/IEC 80000-13:2008 standard (& EN 60027-2:2007 derived from IEC 60027):
  - one kilobyte (**k**B) = 1000 bytes
    one megabyte (MB) = 1000 kB, etc. up to yottabyte (1 YB = 1000 ZB = 10^6 EB = 10^9 PB = 10^{12} TB = 10^{15} GB = 10^{18} MB = 10^{21} kB = 10^{24} bytes)
  - 1024 bytes = 1 kibibyte (**Ki**B)
    1 mebibyte (MiB) = 1024 KiB, etc. up to yobibyte (1 YiB = 1024 ZiB = 2^{20} EiB = 2^{30} PiB = 2^{40} TiB = 2^{50} GiB = 2^{60} MiB = 2^{70} KiB = 2^{80} bytes)
  - *En Français : 1024 octets = 1 kibioctet (**Ki**o), etc.*
- Bits (mostly) for data transfer:
  - one megabit (Mbit) = 1000 kilobit (kbit) = 10^6 bits, etc.
    1 Mbit/s, 1 Gbit/s, etc. (bit/sec & bps also common)

Local storage: basics

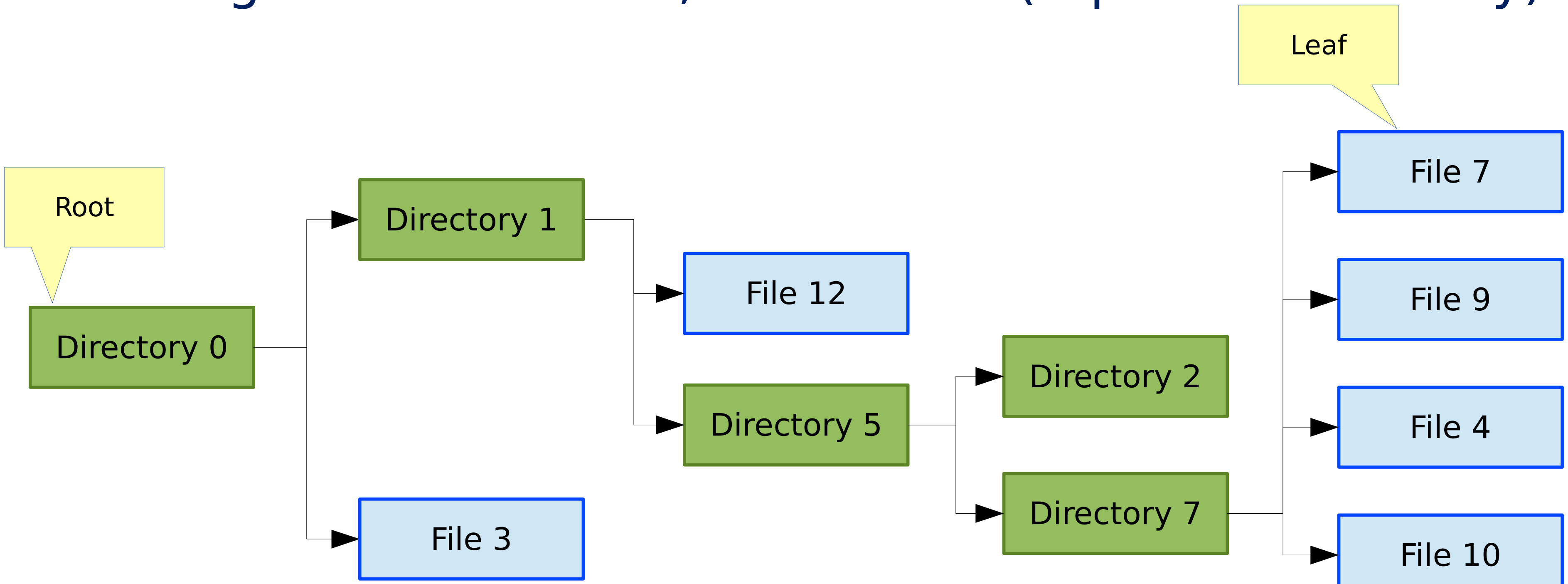
- File: sequences of bytes, one dimensional array, index in the array called *offset*
- Directory (a.k.a. folder): file containing a list of other file(s) name(s)
- Link: file which contains a *reference* to another file

Local storage: files

- Most modern operating systems (Windows, Linux, macOS, ...) do not care about the actual content of files
- Files content can be:
  - *structured* (e.g. *binary* files like JPEG image, NumPy or ROOT data file, extreme case is database table), usually includes some sort of headers with parameters
  - *unstructured* (e.g. text file, like source code), in many cases even unstructured data has some structure (e.g. HTML/XML, JSON, Pickle, CSV, ...)
- Operating systems & programming languages often provide different access functions for *text* (line-oriented) and *binary* (byte-adressed) files

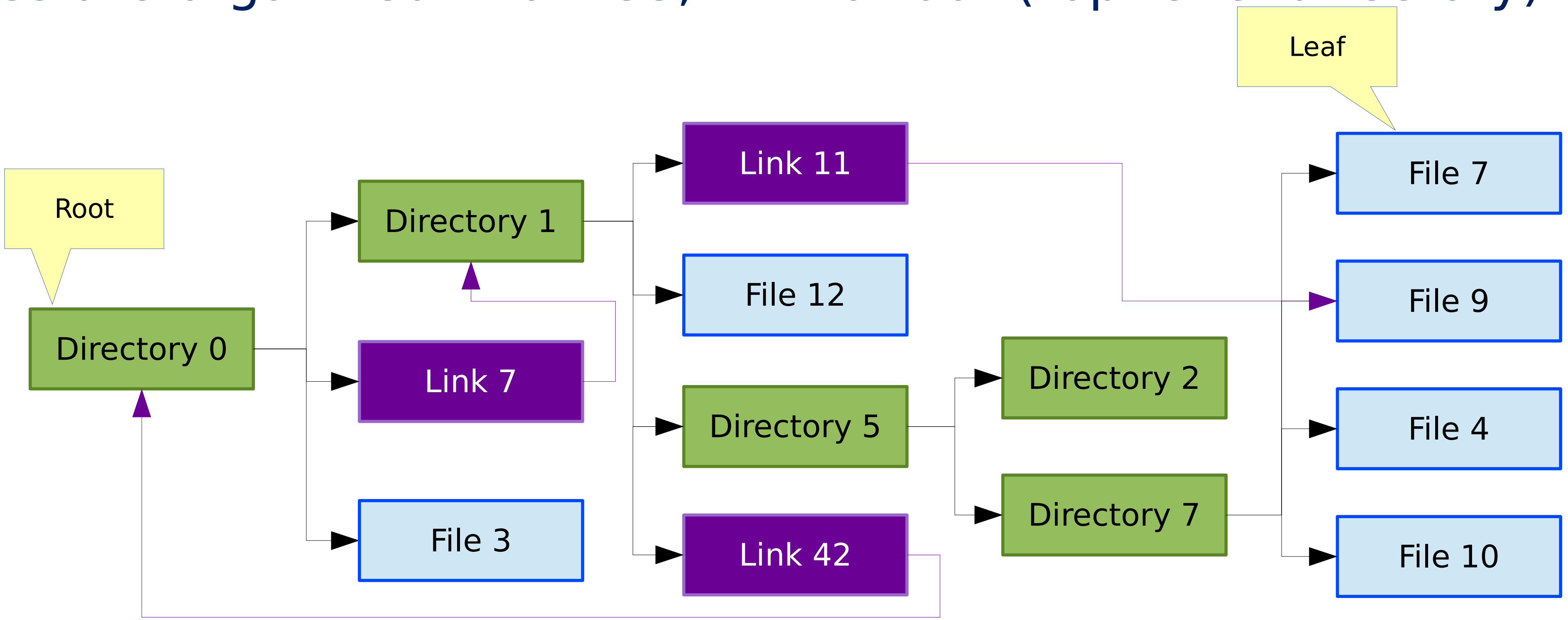
Local storage: namespace [1/2]

- Names are organized in a *tree*, with a *root* (top-level directory):



- Characters allowed in filenames:
  - Unix/Linux: everything **but** `/` and `\0` (ASCII NUL)
  - Windows: everything **but** `\0` and `["\/<>?*:|]`

- Names are organized in a *tree*, with a *root* (top-level directory):



- Characters allowed in filenames:
  - Unix/Linux: everything **but** `/` and `\0` (ASCII NUL)
  - Windows: everything **but** `\0` and `["\/<>?*:|]`

Local storage: names

- (File)names can be *relative* (to a directory) or *absolute* (include the names of all the directories from the root of the filesystem up to the filename)
- Qualified names are sequences of component names and separators:
  - on Unix/Linux, the separator is `/`
  - on Windows the usual separator is `\` but in many cases `/` works too
- Absolute filename examples: `/home/myaccount/.bashrc` or `C:\Windows\Notepad.exe`
- Relative filename example: `../src/plop.py`
- On Unix/Linux, directories names do **not** require a trailing `/`, which is **not** part of the name:

`directory/` ≡ `directory/.` ≡ `directory`

Local storage: filesystems [1/3]

- Filesystem (a.k.a. file system): persistent data structure (on storage media) binding human readable names, data position on media, space allocation and often other attributes
- Filesystems are the product of drive/media formatting
- In order to be accessed, a filesystem must be *mounted* through a *mountpoint* (often a directory or a reserved name, like `C:` on Windows)
- Most filesystems allow (require) *attributes* to be defined for a file, commonly:
  - owner
  - size
  - last access (read) & modification (write) times
  - access permissions

Local storage: filesystems [2/3]

- Filesystems often have constraints in terms of global size, file size, filename length, available attributes, resilience features, ...
- The namespace provided by a filesystem is usually *consistent* for all programs running on the computer hosting the filesystem
- The names, attributes & position information are called the filesystem *metadata*
- Access permissions can often be defined in two ways:
  - basic permissions: Unix permissions, DOS/FAT, ...
  - access control lists *(ACL)*: finer control, Windows (NTFS & ReFS), Unix (ext3/ext4/XFS/ZFS/...), ...
- Extensive documentation on permissions is available, e.g.
  https://en.wikipedia.org/wiki/File_system_permissions

Local storage: filesystems [3/3]

- Many filesystems provide a case sensitive namespace (`myclass.C` ≠ `myclass.c`), on Unix/Linux almost all filesystems do so
- Other file attributes, often called *extended attributes* (EA), allow users to define their own metadata for files:
  - on Unix/Linux: in most cases simple key/value pairs (often with constraints on value size and key name)
  - on Windows & macOS: *forks* (respectively Alternate Data Streams & resources), parallel namespace tied to a single object possibly with its own files, directories, access permissions, …

Local storage: API *(Application programming interface)* [1/3]

- All general purpose operating systems and programming languages support similar basic file operations:
  - open & close a file *(open, close)*
  - read & write a specified amount of bytes at some offset in the file *(read, readline, write, pread, pwrite, ...)*
  - get/set the default/current read/write offset in the file *(tell, ftell, seek, lseek, fseek)*
  - remove a file *(unlink, remove, ...)*
- For directories:
  - get the list of files in the directory *(readdir, ...)*
  - move to another directory *(chdir)*
  - create & remove a directory *(mkdir, rmdir)*

Local storage: API *(Application programming interface)* [2/3]

- Python basic examples to
  - open a (text) file and read its lines:

```
#!/usr/bin/env python

import sys

with open(sys.argv[1]) as fp:
  for line in fp:
    smode, uid, ssize = line.split(':')[1:6:2]
[...]
```

  - open a (binary) file for writes and write some data structure at the end of it (*append*):

```
#!/usr/bin/env python
[...]
with open("../outputfile.dat", "wb+") as output:
  output.write(datastructure)
[...]
```

Local storage: API *(Application programming interface)* [3/3]

☞ reads and writes can happen at any offset in a file, a file can be read/written as a whole or in part(s) depending on your program needs

☞ file content can be *executed* (run as a program) either by the system itself (e.g. compiled binary file like `.exe` files on Windows) or through an interpreter (e.g. Python in the previous examples, and scripted languages like PowerShell, bash, Perl, R, ...)

☞ when multiple programs running concurrently want to access the same file and at least one program is writing to that file, a common mechanism is to use a *lock* to synchronize access among the programs to avoid data corruption (in memory and/or on the storage media)

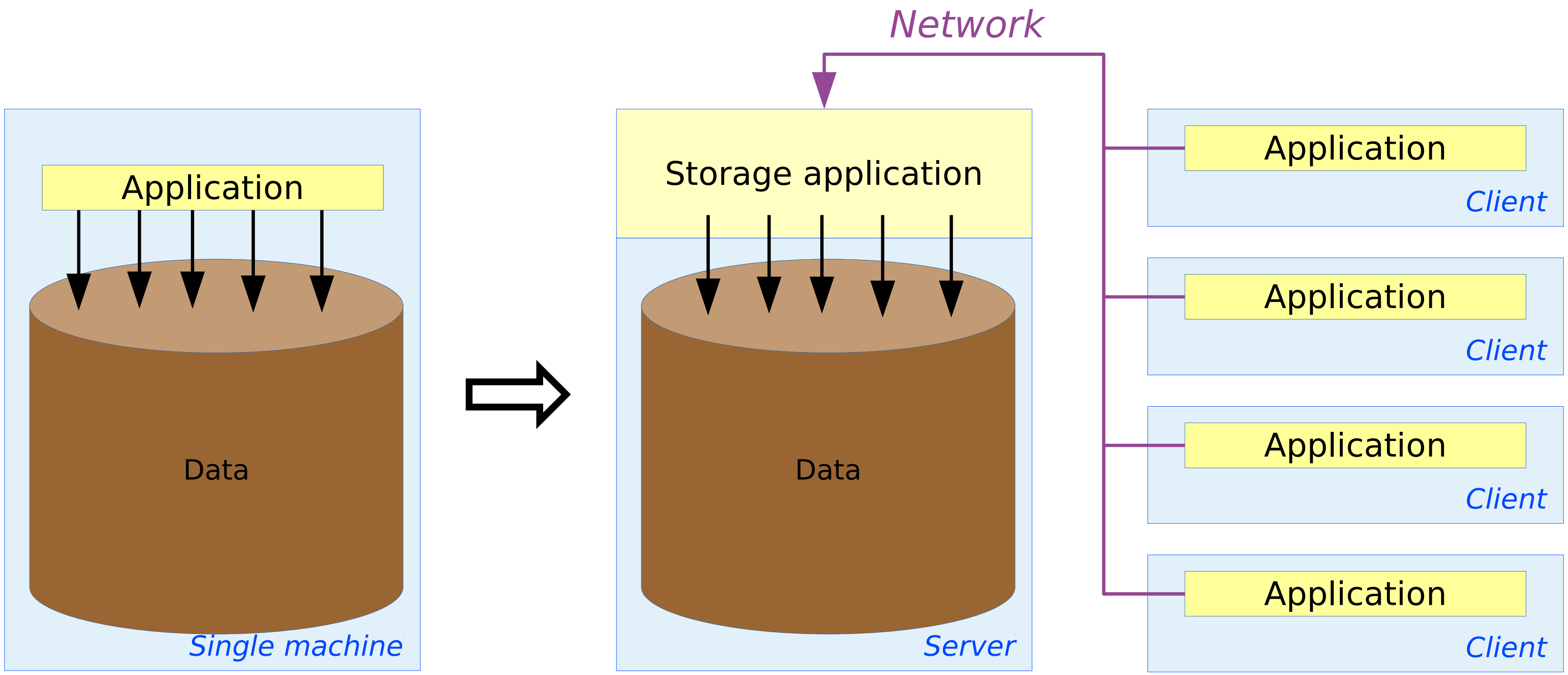
Local storage: cache

- On most operating systems, a non persistent copy of recently accessed data is kept in memory to:
  - avoid reading from the storage media if data is reused
  - aggregate small (or non optimally sized) writes to use the storage media more effectively
  - in some cases, read data in advance (*before* a program requests it)
- There is often limited user control over cache behaviour
- Often depends on available (unused) memory:
  - stale cache content can be discarded to free up memory in order to satisfy programs requests
  - non stale content (writes not commited to storage yet) can be discarded after a *flush* (user/system request or memory pressure)

Local storage: quotas

- Some filesystems allow quotas to be defined
- Quotas can usually be defined for files space and number of files
- Quota enforcement can be:
  - strict, no new data stored after quota is exceeded
  - relaxed/advisory, user may be notified directly
- Rarely used on local storage in a non shared environment

Distributed storage: basic principle



Distributed storage: basic use cases

- Multiple general patterns exist to access data from **another** machine, to allow **local** programs read or write access to this data
- Common data access schemes are:
  - pull/push: *pull* (copy) the input files to the local machine storage before processing, *push* the new or modified (output) files to the remote storage after processing
  - direct access: remote machine provides a mechanism for *local* program to read/write directly to files without the need for a copy
  - mixed: depending on the actual I/O pattern, pulling input files might give better for performance than direct remote access (e.g. file read as a whole anyway, many random reads from a large file, many small random writes to a large file, ...)

Distributed storage: pull/push trivial example

- Batch job script example:

```
#!/bin/sh

# Get input files (pull)
wget -v https://somewhere.fake.fr/data/123456789.tgz -O $TMPDIR/input.tgz
[ -s $TMPDIR/input.tgz ] || { echo "Input data retrieval failed"; exit 1; }

# Extract the content of the input archive
cd $TMPDIR && tar zxf input.tgz && rm -v input.tgz

# Process the data using a Python Virtual Environment
. ~/fake-processing-venv/activate.sh
~/fake-processing/do-something.py --input-dir $TMPDIR --output $TMPDIR/${JOBID}.txt

# Check processing exit value & output file size
[ $? != 0 -o ! -s $TMPDIR/${JOBID}.txt ] && { echo "Processing failed"; exit 1; }

# Push back result somewhere (some configuration required)
scp $TMPDIR/${JOBID}.txt machine.fake.fr:fake-processing/output

# Cleanup
rm -rf $TMPDIR
```

Distributed storage: direct access trivial example

- Batch job script example:

```
#!/bin/sh

# Process the data using a Python Virtual Environment
. ~/fake-processing-venv/activate.sh

INPUTDIR=/data/mygroup/me/fake-processing/inputs/123456789

cd /data/mygroup/me/fake-processing/outputs
~/fake-processing/do-something.py --input-dir $INPUTDIR --output ${JOBID}.txt

# Check processing exit value & output file size
[ $? != 0 -o ! -s ${JOBID}.txt ] && { echo "Processing failed"; exit 1; }
```

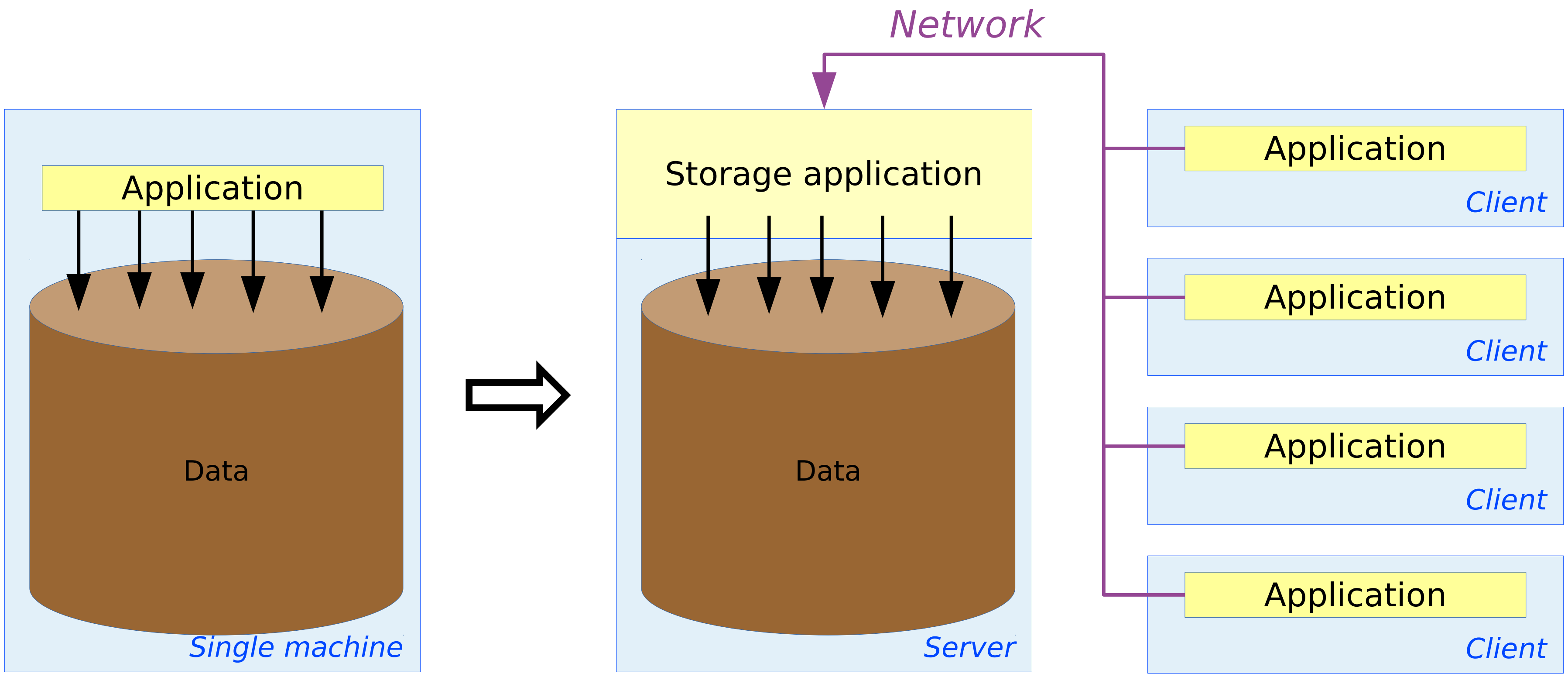
Distributed storage: shared filesystem

- Provide acces to a filesystem beyond the limit of a single machine (like Windows *Shared Folder*)
- Transparent access (same *API*), same program running on:
  - a laptop, accessing data on the laptop
  - on a computing infrastructure, accessing data on the computing infrastructure
- Many solutions exist, for example:
  - export a local filesystem over a network (NFS, SMB, ...)
  - use a parallel filesystem (Lustre, GPFS, BeeGFS, ...)
- Filesystems usually allow: in-place updates, program execution, partial read & writes

Distributed storage: *shared* filesystem

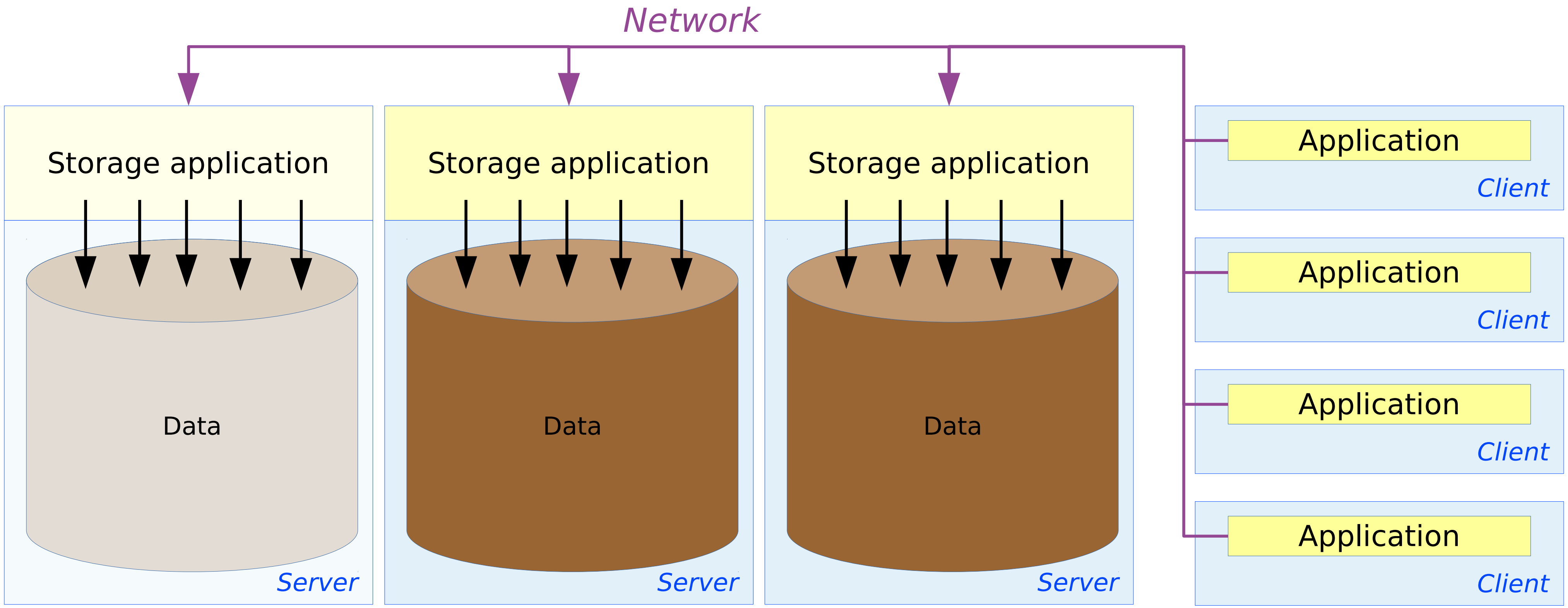
- **Shared** namespace among clients
- **Shared** user identification (permissions, accounting)
- Single or multiple servers for actual data storage
- Single or multiple servers for data access
- Consistent (shared) view (POSIX-like semantics) or not:
  - data caching on clients
  - data/file locking on clients
- Quotas

Distributed storage: distributed filesystem



1 ⇨ 1 server:N clients

Distributed storage: parallel filesystem/storage



M:N ⇨ M servers:N clients

Distributed storage: object storage [1/3]

- Shared namespaces impose synchronisation which limits *scalability*
- Object storage provides access to **independent** objects: no shared namespace (no directory)
- Objects can be viewed as a generalization of files
- Duality of *object storage API* and actual *object storage system* (OSD, like Ceph, Panasas, ...)
- Most common current meaning is <u>object storage API</u>:
  - HTTP(S) transport, REST API, PUT/GET/DELETE of (mostly) whole objects
  - *de facto* standard is Amazon S3, supported by most object storage systems (often in addition to their own)
  - *de jure* standard is CDMI, but is uncommon

Distributed storage: object storage [2/3]

- *User oriented* object storage system (Dropbox, iCloud, ...) provide a user specific namespace, often simulating familiar visible namespace features (directories)
- Basic data organization is the *bucket* (container), generally:
  - one user per container
  - no nested containers
  - multiple containers per user common
- With Amazon S3 (*Simple Storage Service*):
  - Object references look like:
    `http://s3.amazonaws.com/bucket/key`
  - `bucket` is the bucket identifier
  - `key` is the object identifier

Distributed storage: object storage [3/3]

- Permissions can be set per object or container, no need for shared user identification
- Often no support for in-place update: no modification to stored objects, a new *version* of the object is created instead
- No support for direct program execution (without filesystem view)
- Partial reads frequently supported
- Partial or continuous writes support uncommon
- User defined *tags* (attributes) can be created & used to select groups of objects (*collections*)

Distributed storage: object storage API example

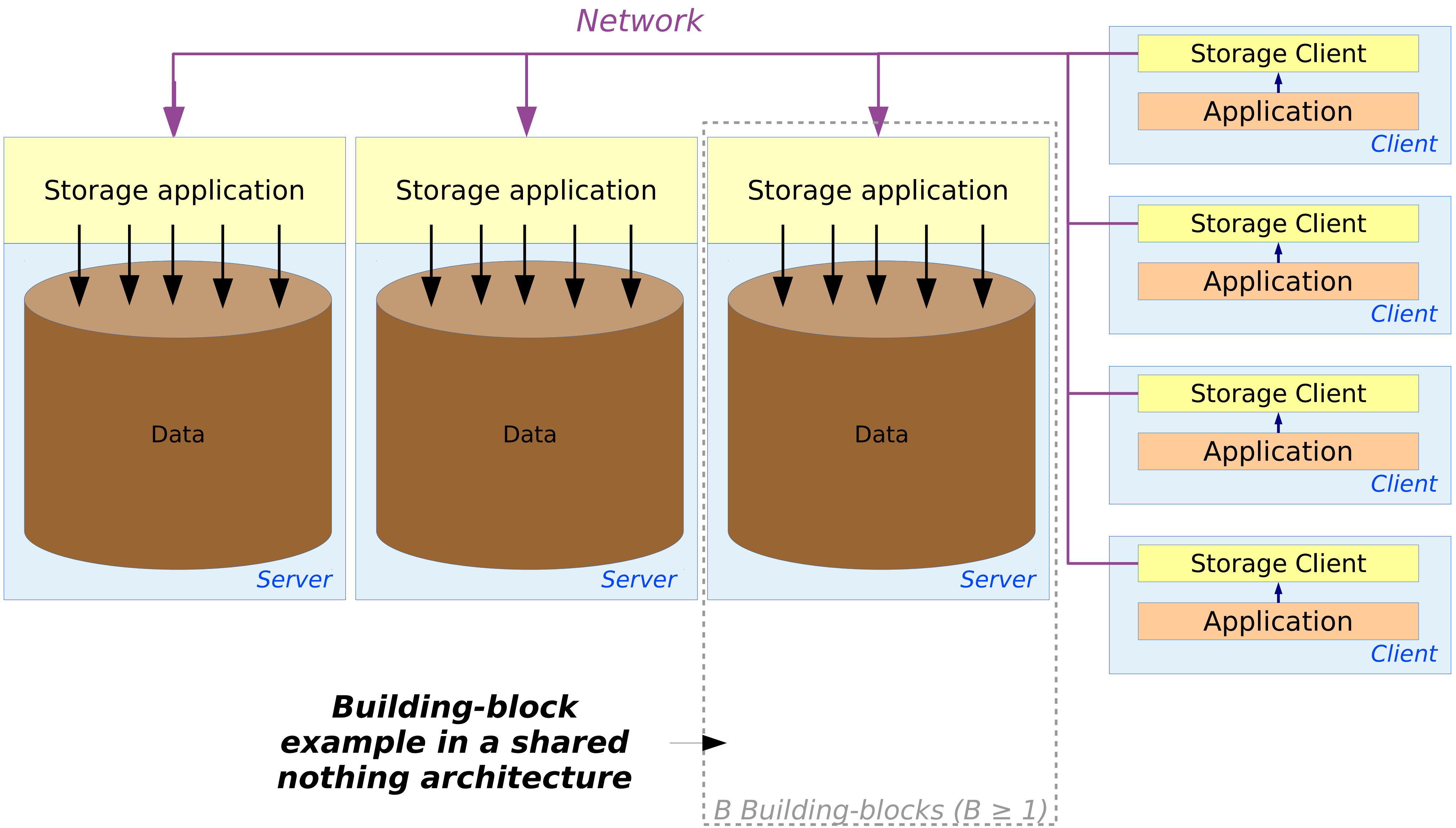
- Python example using the Boto module:

```
#!/usr/bin/env python

import boto.s3.connection

connection = boto.s3.connection.S3Connection(
    aws_access_key_id='EC2_ACCESS_KEY',
    aws_secret_access_key='EC2_SECRET_KEY',
    port=8080,
    host='s3.amazonaws.com',
    is_secure=True,
    validate_certs=True,
    calling_format=boto.s3.connection.OrdinaryCallingFormat()
)

buckets = connection.get_all_buckets()
for b in buckets:
    print b.name

bucket = conn.get_bucket('mybucket', validate=True)
for key in bucket.list():
    print "{name}\t{size}\t{modified}".format(
          name=key.name,
          size=key.size,
          modified=key.last_modified)
```

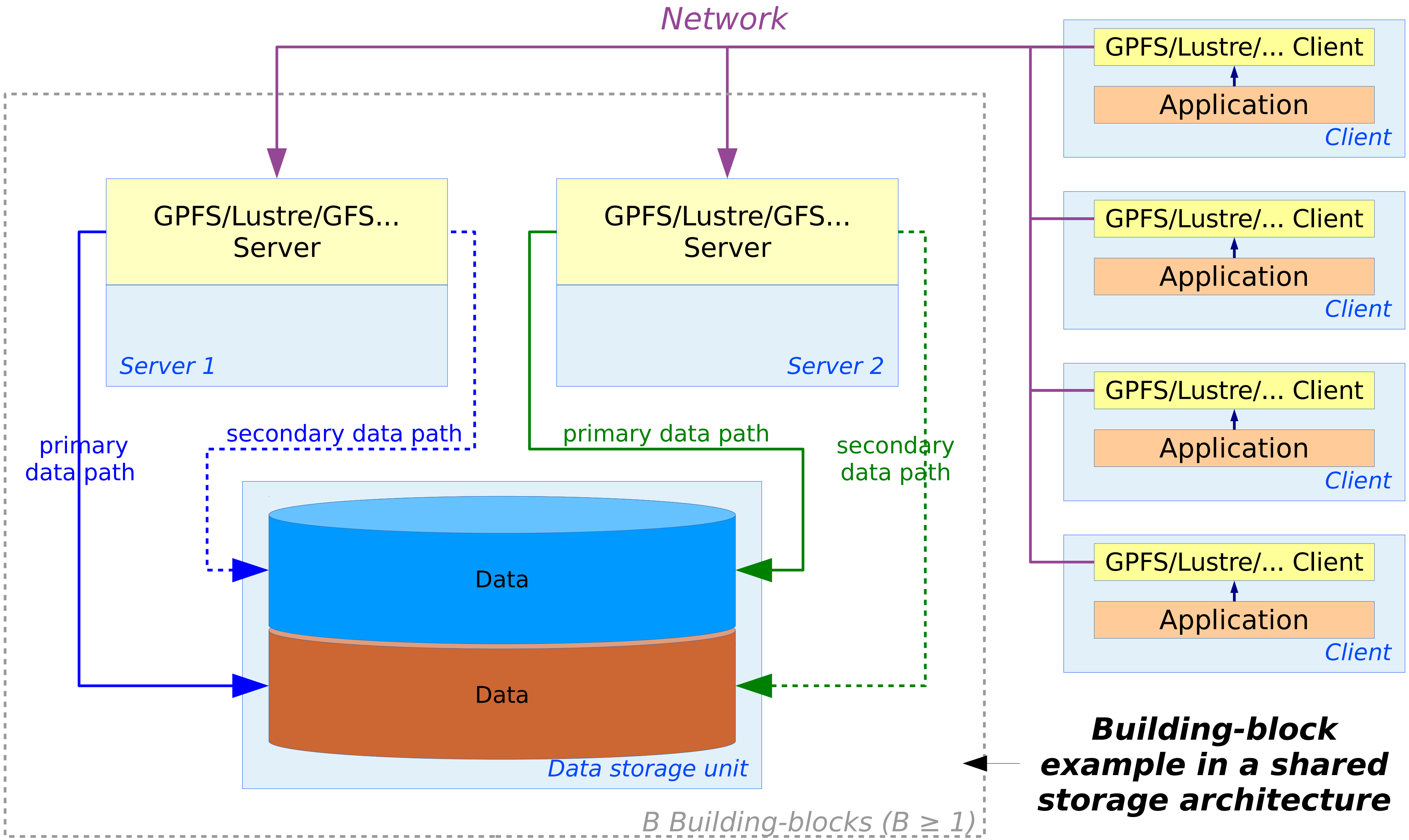
Distributed storage: architectures

- Two main *classes* of distributed storage architectures:
  - shared nothing: independant servers with no shared storage resources ⇨ basic model for *modern* storage systems, *scale-out* architectures
  - shared storage hardware resources: storage specific network and devices, often only a limited amount of resources actually shared
- Difference is mostly balance between:
  - cost, shared nothing architectures often rely on cheaper servers
  - available user space, shared nothing architectures often rely on (simple) data replication
- *Scale-out* and *building-block* approaches can both be used with shared nothing & shared storage architectures

Distributed storage: shared nothing architecture



Distributed storage: *shared* storage architecture



Building-block example in a shared storage architecture

Data locality

- The *closer* data is to the program the faster it can be accessed
- Big data infrastructure mantra: bring the code to the data and not the data to the code
- Memory hierarchy:
  - for speed: RAM > NVM (Flash etc.) > Disk *(& Network)* > Tape
  - for capacity: Tape > Disk > NVM > RAM *(Network ∞ ?)*
- *In memory* processing

Basic data performance best practices

- Write only what matters, while storage may be "cheap", data management is not
- Avoid intermediary files (or write them locally)
- Read/write largest possible relevant *chunks*, some storage systems provide a *preferred* I/O size information
- Group writes (at the thread, process or file level)
- Use efficient high-level data libraries (ROOT, HDF5, even NumPy, ...) before doing your own
- Avoid *synchronous* I/O unless you're sharing a file between machines or if it's the final output

Basic non-performance best practices

- Data management plans (DMP) are important, data management is tedious work
- Avoid putting all your files in the same directory
- Limit the number of files: more files ➪ more work to manage data
- Avoid (lots of) extremely small files (< 512 B or 1 KiB), use a database when it makes sense
- Data structure in memory and on persistent storage do not need to be identical (*serialization/marshalling*)
- On Unix/Linux, be careful when you *seek* inside a file opened for:
  - reading: going past the end of a file will yield an error
  - writing: going past the end of a file will **not** (append & sparse files)

Best practices for filenames (& permissions)

- Use meaningful (and concise) directory names & filenames
- Even with Unicode, try to limit files/directories names to simple printable characters (`-+:%@_.`) and *alphanumericals* (`0-9`, `A-Z`, `a-z`):
  - avoid characters which can be interpreted by the shell or programming language, or difficult to use in command arguments
  - **avoid** creating files named: `*, $, \, \n, \ESC^C^Z^D`
- When creating filenames with a program, make sure you create a printable name (C++, ROOT, ...): format integers as text
- When a filename contains a date, use ISO-8601 date format (e.g. something between `2020-03-10` and `2020-03-10T09:58:19Z`)
- Do **not** use `chmod 777`, `chmod -R 777` (or `chmod -R a+rwx`) or similar

Application design

- If possible, do not focus on a preferred storage solution:
  - parallel filesystems are nice, but does your use case actually need one (global consistency, in-place updates, single namespace, etc.) ?
  - object storage is nice, but do you actually need billions of objects or pseudo-infinite scalability ?
- **Ideally** use an I/O abstraction layer which will allow a switch to another storage system without changing the application itself
- For user defined metadata: use tags/extended attributes if available, otherwise use a database instead of ancillary files (even SQLite)