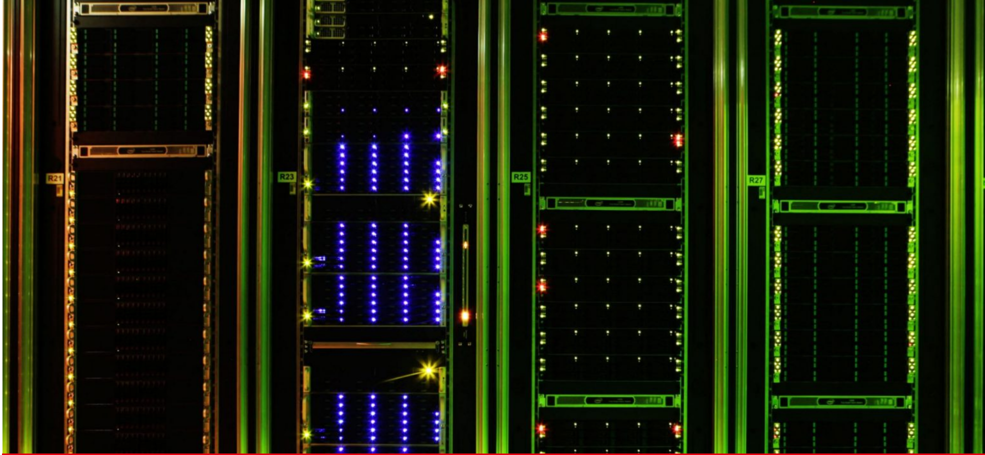
EPFL



MATH-454 Parallel and High Performance Computing
Lecture 1: Execution in an HPC environment

Pablo Antolin

Slides of N. Richart, E. Lanti, V. Keller's lecture notes

February 27 2025

SCITAS

EPFL

Outline

Today exercise session

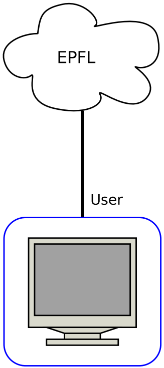
- What is a cluster.
- How to connect to a cluster (any remote machine).
- How to compile codes and use libraries.
- How to submit job on a cluster.
- How to use GIT

Getting your accounts

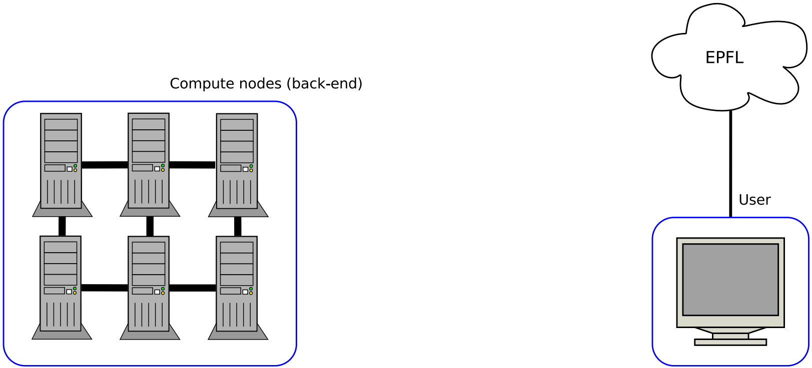
groups.epfl.ch

If you are not in the group hpc-math454 let me know

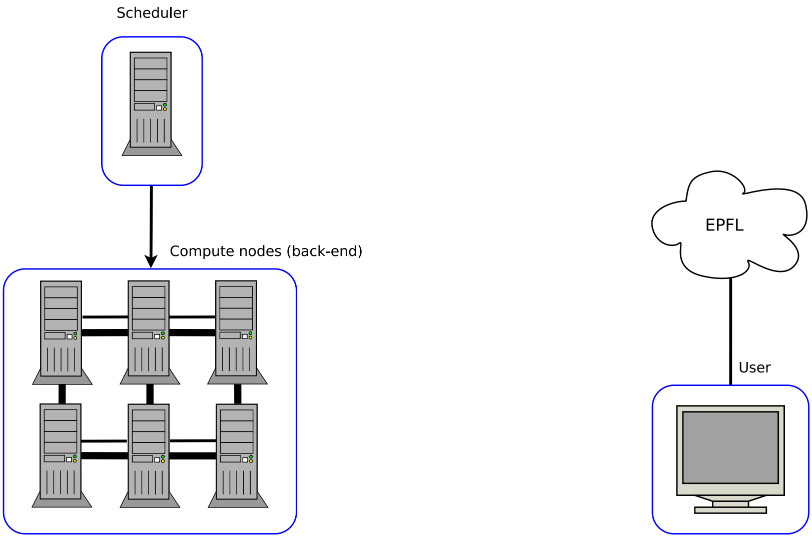
What is a cluster?



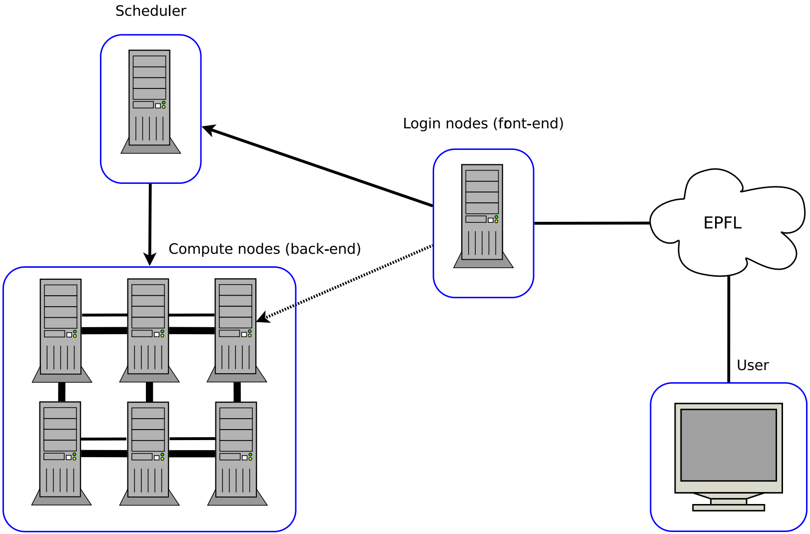
What is a cluster?



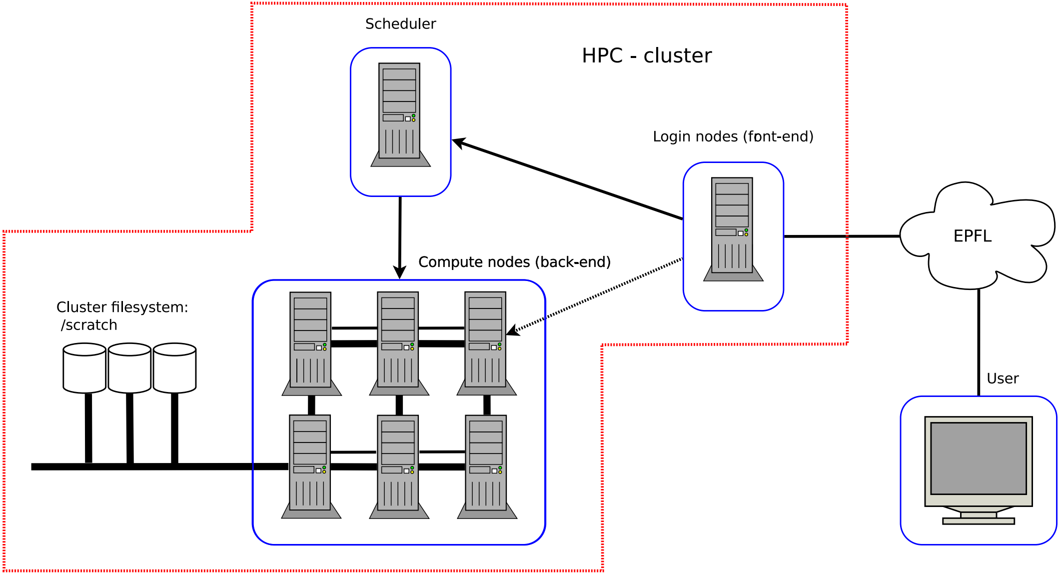
What is a cluster?



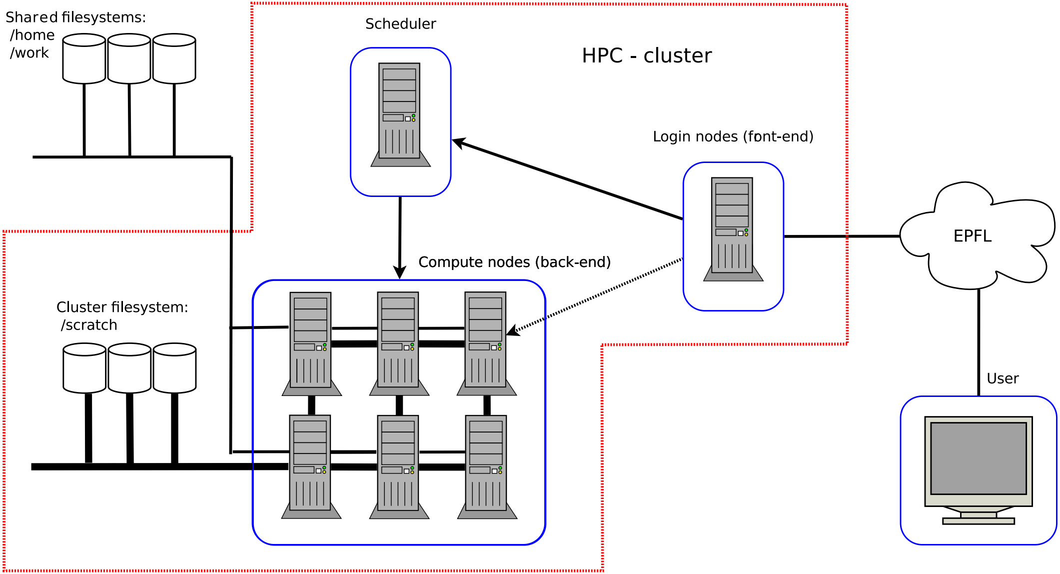
What is a cluster?



What is a cluster?



What is a cluster?



EPFL clusters

Login	Hosts	Cores	RAM	Network
hostname	#	# x GHz	GB	Gbit/s
`helvetios.hpc.epfl.ch`	287	36x2.3	192	100 (IB)
`jed.hpc.epfl.ch`	375	72x2.4	512	Ethernet
	42	72x2.4	1024	Ethernet
	2	72x2.4	2048	Ethernet

EPFL clusters

Login hostname	Hosts #	Cores # x GHz	RAM GB	Network Gbit/s
`izar.hpc.epfl.ch`	35	40x2.1	192	2x100 (IB)
		+ *2x NVidia V100 7TFlops*		
	35	40x2.1	384	100 (IB)
		+ *4x NVidia V100 7.8TFlops*		
	2	40x2.1	768	100 (IB)
		+ *4x NVidia V100 7.8TFlops*		

EPFL clusters

Login hostname	Hosts #	Cores # x GHz	RAM GB	Network Gbit/s
`kuma.hpc.epfl.ch`	84	32x2.7	384	2x200 (IB)
		+ *4x NVidia H100 94 GB*		
	20	32x2.7	384	2x200 (IB)
		+ *8x NVidia L40S 48 GB*		
	2	32x2.7	384	2x200 (IB)

EPFL clusters storage

- The simulation data is written on the storage systems. At SCITAS:
  - /**home**: store source files, input data, small files
  - /**work**: collaboration space for a group
  - /**scratch**: temporary huge result files
- Please, note that only /**home** and /**work** have backups!
- /**scratch** data can be erased at any moment!

Connecting to remote machines

First step

- Connect to a remote cluster to get a shell
- SSH: Secure SHell

How to use

```
$ ssh -l <username>  <hostname>
$ ssh <username>@<hostname>
```

Connecting to remote machines

First step

- Connect to a remote cluster to get a shell
- SSH: Secure SHell

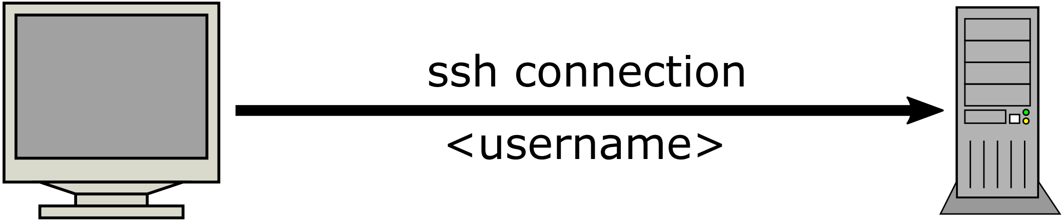
How to use

```
$ ssh -l <username>  <hostname>
$ ssh <username>@<hostname>
```

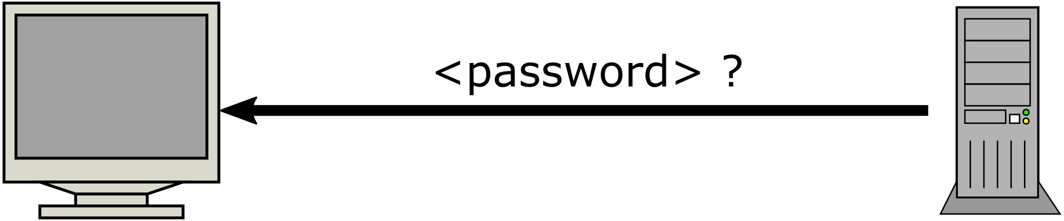
For windows users

Just install **git**, and use **git bash**

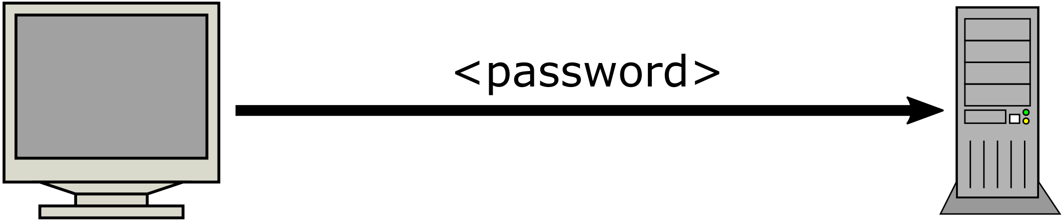
Getting a remote shell



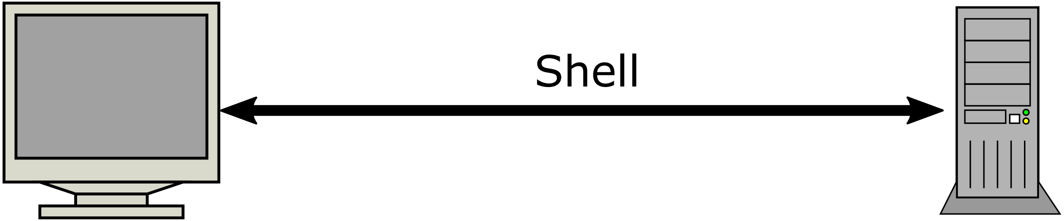
Getting a remote shell



Getting a remote shell



Getting a remote shell



Exercise 1: Simple connection

To connect to the front node of a cluster

```
$ ssh -l jdoe  helvetios.hpc.epfl.ch
$ ssh jdoe@helvetios.hpc.epfl.ch
```

Front nodes

- **helvetios** *[CPU (OpenMP/MPI)]*
- **izar** *[GPU (CUDA)]*

- Connect to helvetios' front node
- Check the different folders /**home** /**scratch**

Transferring files: using scp

scp works fine for Linux/MacOS, it is often pre-installed
On windows use GIT Bash or pscp.exe from Putty

How to use **scp/pscp.exe**

Send data to remote machine:

```
$ scp [-r] <local_path>  <username>@<remote>:<remote_path>
```

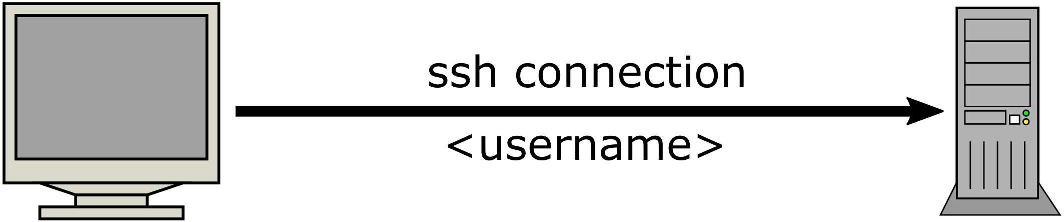
Retrieve data from remote machine:

```
$ scp [-r] <username>@<remote>:<remote_path> <local_path>
```

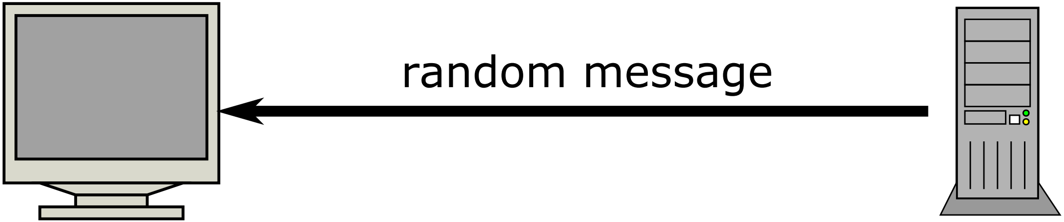
Note: *It is always easier to "send to" and "receive from" the clusters since they have a fix ip/name*

- Copy a file from your machine to the cluster.
- Modify the file (e.g., using **vim** or **nano**) and retrieve it from the cluster onto your machine

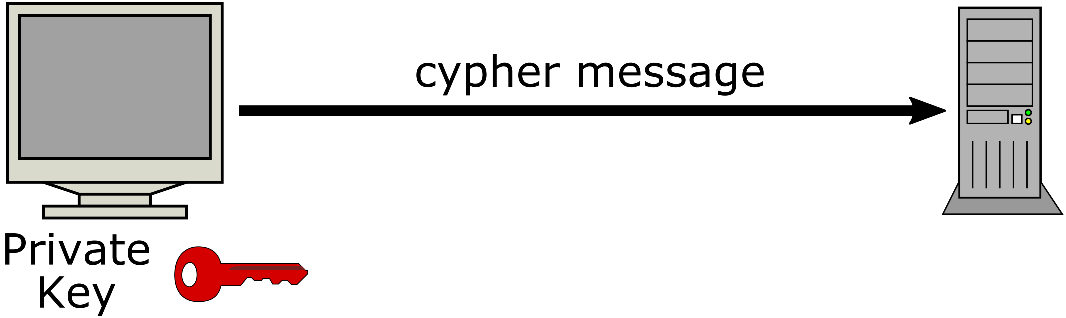
Getting a remote shell using keys



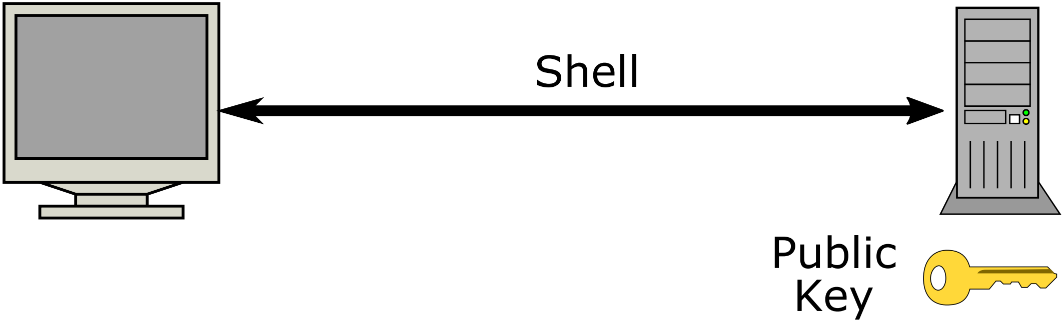
Getting a remote shell using keys



Getting a remote shell using keys



Getting a remote shell using keys



Exercise 3: Connection using keys

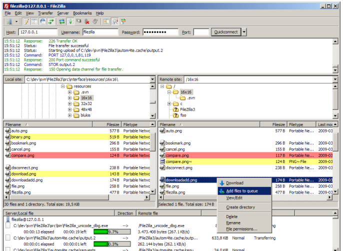
- Generate a pair of public/private key using **ssh-keygen**.
- Copy the generated public key
- Try to connect (it should not ask your password)
- Same ssh key should be already working for **helvetios**, **jed**, and **izar**

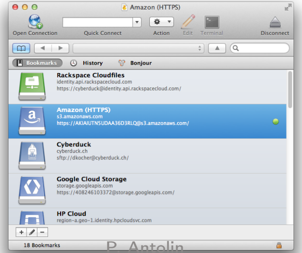
Example

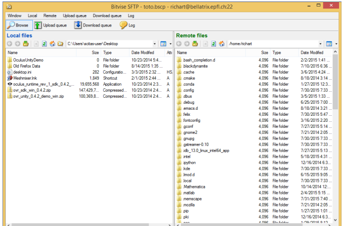
```
$ ssh-keygen -b 4096 -t rsa
  [ Follow instructions ]
$ ssh-copy-id jdoe@helvetios.hpc.epfl.ch
```

Transferring data using a GUI

sftp tones of GUIs, for example: Filezilla `https://filezilla-project.org` ("all" OSes)
Cyberduck `https://cyberduck.io/` (Windows and Mac)
Tunnelier `https://bitvise.com/tunnelier` (Windows)
TUIs also exists like **lftp**







Connecting to a Remote Linux Machine with VS Code

It is common practice to use VS Code to edit code on remote machines just as if it was your local machine.

Steps:

1. Install the "Remote - SSH" Extension
   - Open VS Code
   - Go to Extensions (`Ctrl+Shift+X` on Windows/Linux, `Cmd+Shift+X` on Mac)
   - Search for "Remote - SSH" and install it
2. Open the Command Palette (`Ctrl+Shift+P` on Windows/Linux, `Cmd+Shift+P` on Mac)
3. Select: `Remote-SSH: Connect to Host...`
4. Enter your remote machine details:
   - Example: `ssh username@helvetios.hpc.epfl.ch`
5. Authenticate
   - Enter your password or use an SSH key as shown before
6. Start coding!

Using modules

What are modules

- A way to dynamically modify the environment
- The contain configurations to use an application/a library

How to use them

- **module avail** list all possible modules
- **module load <module>** load a module
- **module unload <module>** unload a module
- **module purge** unload all the modules
- **module list** list all loaded modules
- **module spider <module>** info about a module

Exercise 4: Module command

- List all available modules
- Try **g++ --version**
- Load the module `gcc`
- Try again **g++ --version**
- Try **icpc --version**
- Load the `intel` module
- Try again **icpc --version**
- List the currently loaded modules

Exercise 5: Compilation

GCC:

```
$ g++ -o <executable> <sources>
```

Intel:

```
$ icpc -o <executable> <sources>
```

Compilation all-in-one

If the compiler is given source code only it will do all the stages at once

- Load a compiler module
- Compile the code `hello.cc`

Separated compilation

- Generate object files: compile

```
$ g++ -c file1.cc
$ g++ -c file2.cc
```

Transform code file **file?.cc** in object file **file?.o**

- Generate the executable: linking

```
$ g++ -o hello file1.o file2.o
```

Transforms object file **file?.o** in executable **hello**

- Translation is made by a compiler in 4 steps

  Preprocessing Format source code to make it ready for compilation (remove comments, execute preprocessing directives such as `#include`, etc.)

  Compiling Translate the source code (C, C++, Fortran, etc) into assembly, a very basic CPU-dependent language

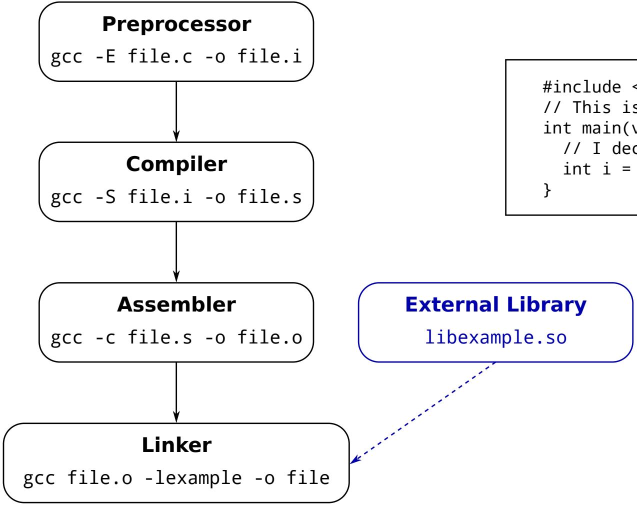
  Assembly Translate the assembly into machine code and store it in object files

  Linking Link all the object files into one executable

- In practice, the first three steps are combined together and simply called "compiling"

Compilation stages

The four compilation steps (visually)




```
#include <stdio.h>
// This is my main function
int main(void) {
  // I declare i to be equal to 2
  int i = 2;
}
```

Preprocessor
`gcc -E file.c -o file.i`

↓

Compiler
`gcc -S file.i -o file.s`

↓

Assembler
`gcc -c file.s -o file.o`

↓

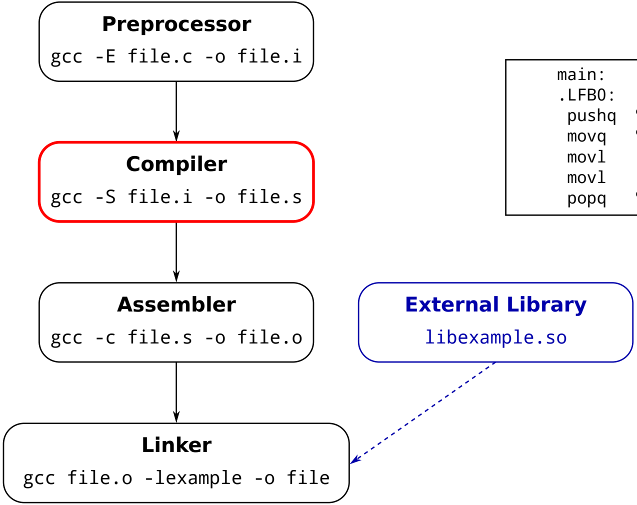
Linker
`gcc file.o -lexample -o file`

External Library
`libexample.so`

```
[...]
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);

int main(void) {
  int i = 2;
}
```

Compilation stages

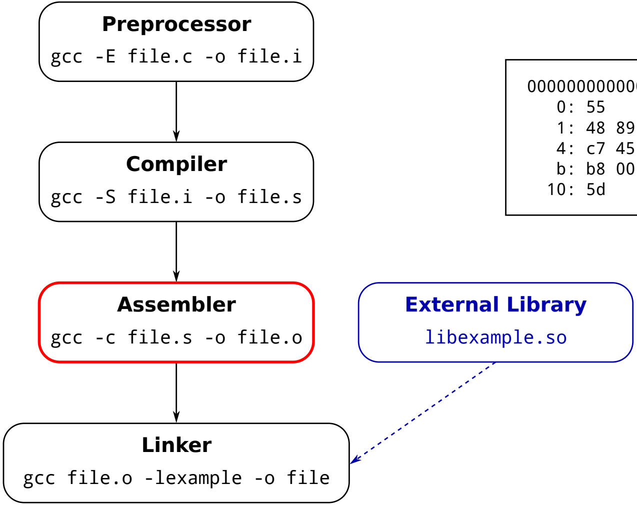
The four compilation steps (visually)




```
main:
.LFB0:
 pushq  %rbp
 movq   %rsp, %rbp
 movl   $2, -4(%rbp)
 movl   $0, %eax
 popq   %rbp
```

Compilation stages

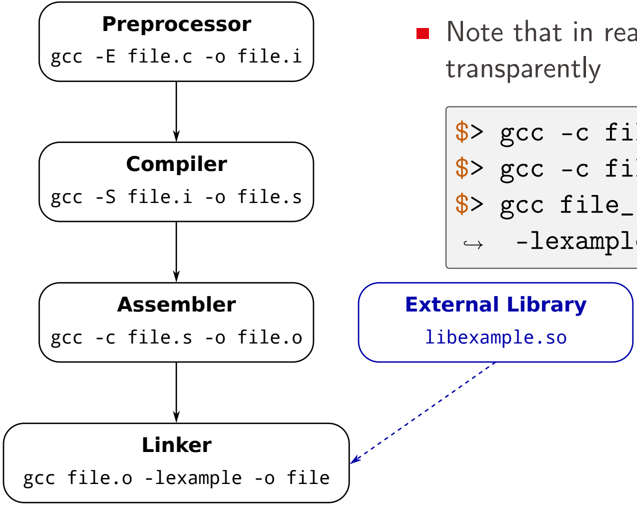
The four compilation steps (visually)




```
0000000000000000 <main>:
   0: 55
   1: 48 89 e5
   4: c7 45 fc 02 00 00 00
   b: b8 00 00 00 00
  10: 5d
```

Compilation stages

The four compilation steps (visually)



- Note that in reality, everything is done transparently

```
$> gcc -c file_1.c
$> gcc -c file_2.c
$> gcc file_1.o file_2.o
↪  -lexample -o exec
```

Exercise 6: Separated compilation

- Clone the repository
  https://gitlab.epfl.ch/math454-phpc/exercises-2025
  (you have to use your gaspar credentials)

```
$ git clone
↪  https://gitlab.epfl.ch/math454-phpc/exercises-2025.git
```

  You can clone it either in your local machine or in `helvetios`.
  - If you clone it in your local machine, you have to copy the folders to `helvetios` using `scp`.
- From `helvetios`, enter the folder **hello-separated-files**
- Generate `hello.o` and `greetings.o`
- Generate `hello` from `hello.o greetings.o`

What is makefile

- It is a build automation system
- It is a set of rules on how to produce an executable:
  - what and how to compile?
  - what and how to link?
- Rule for compiling

```
.cc.o:
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $<
```

- Rule for linking

```
$(EXE): $(OBJS)
	$(LD) -o $@ $(LDFLAGS) $(OBJS)
```

- Usage: **make**

Exercise 7: Makefiles

- Enter the folder **hello-makefile**
- Read the **Makefile**
- Add the -**Wall** -**Werror** option to the compilation options
- Compile the code
- Modify the file **greetings.cc** as:

greetings.cc

```cpp
#include <iostream>
#include "greetings.hh"

void greetings() {
    int dummy;
    std::cout << "Hello world !" << std::endl;
}
```

- Compile the code

Generate a library

- Two types of libraries **static** or **shared**
  - static: archive of object file
  - shared: library loaded dynamically at execution
- Generate a shared library
  Object files have to be compiled with -**fPIC** option

```
$ g++ -o libgreetings.so -shared -fPIC  greetings.o
```

- Link

```
$ g++ -L<lib path> -lgreetings -o <exec> <obj_1> ... <obj_n>
```

- Execute

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<lib path> <exec>
```

Real life project

For project a bit more consequent with multiple dependencies to external libraries, usage of a build automation system

- CMake
- Autotools: configure
- SCons

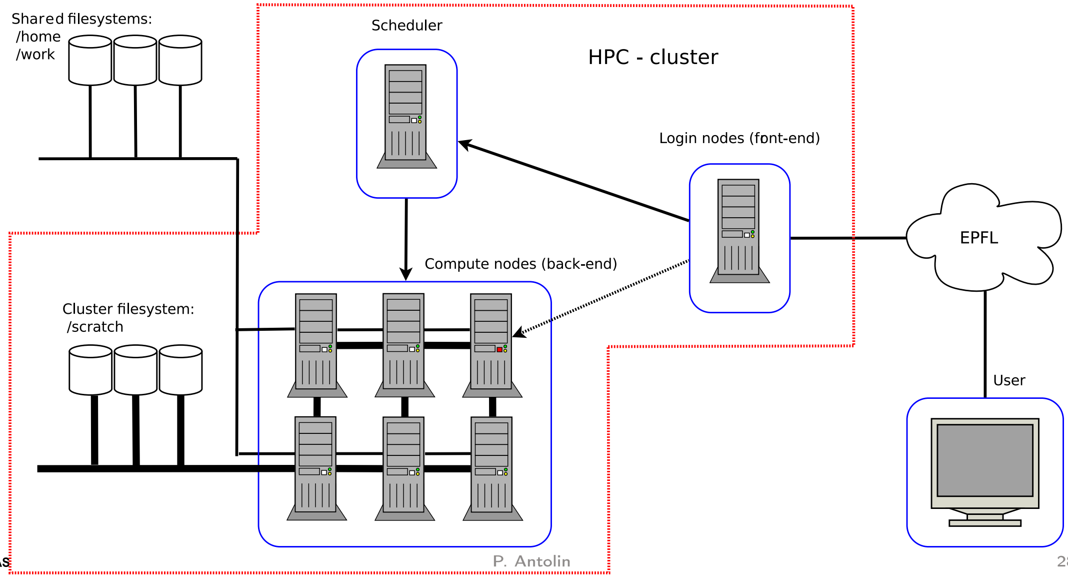
SLURM

SLURM

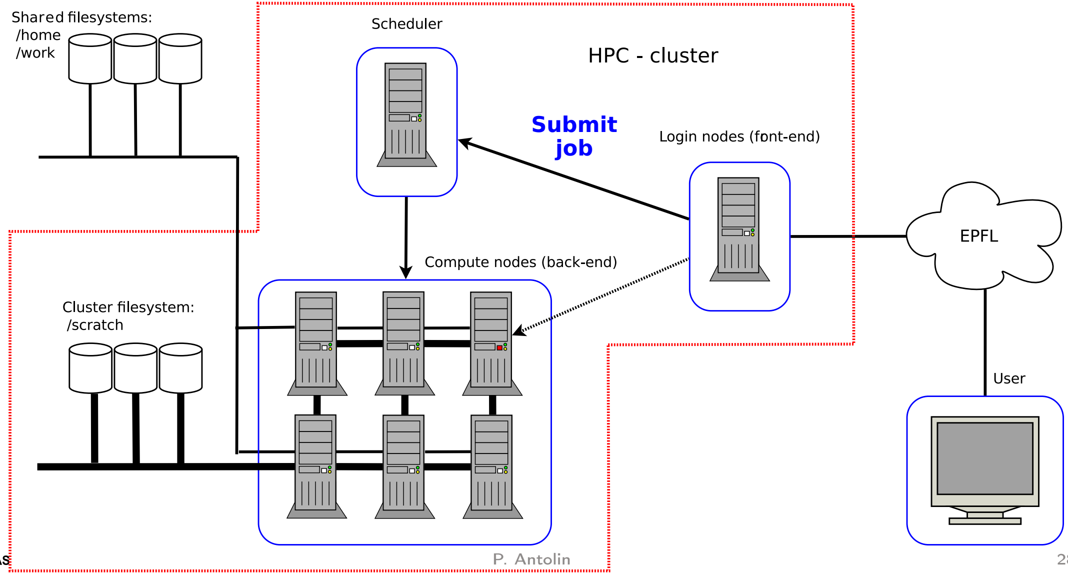
slurm

workload manager

Scheduler

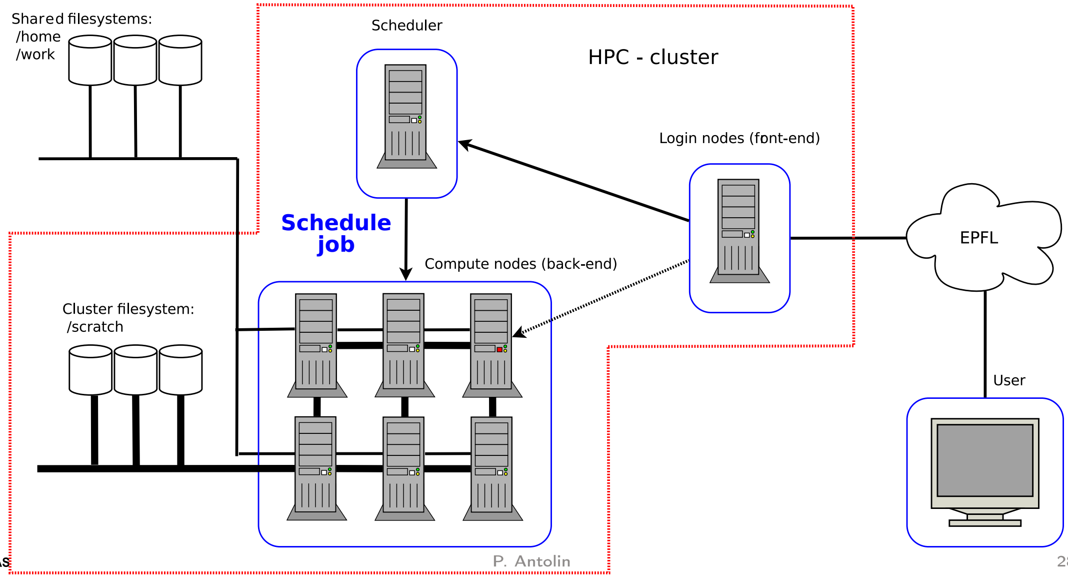

Shared filesystems:
/home
/work
Scheduler
HPC - cluster
Login nodes (font-end)
EPFL
Compute nodes (back-end)
Cluster filesystem:
/scratch
User

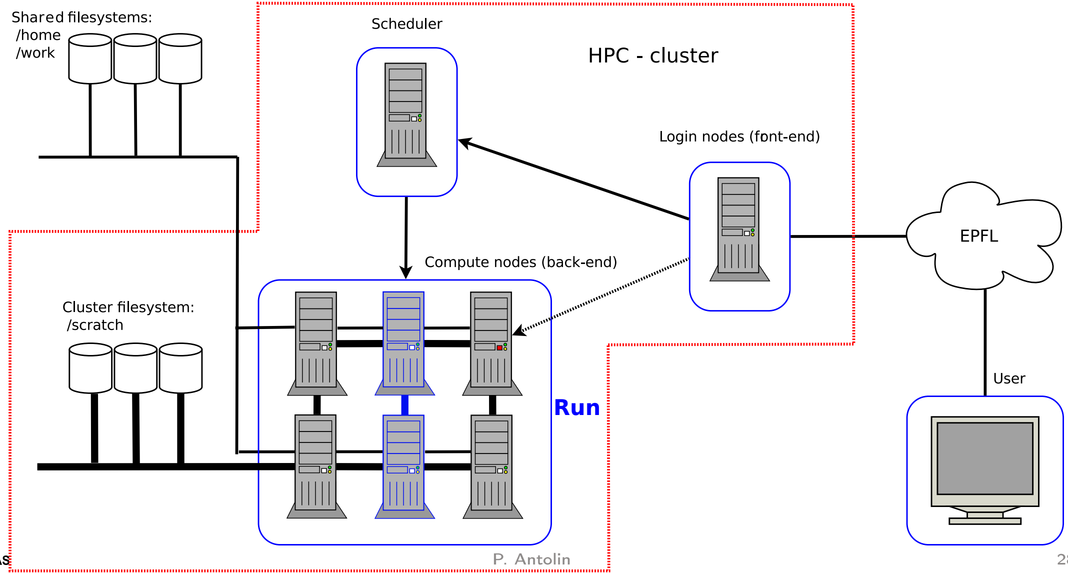
Scheduler



Shared filesystems:
/home
/work
Scheduler
HPC - cluster
Submit job
Login nodes (font-end)
EPFL
Compute nodes (back-end)
Cluster filesystem:
/scratch
User

Scheduler


Shared filesystems:
/home
/work
Scheduler
HPC - cluster
Login nodes (font-end)
Schedule job
Compute nodes (back-end)
EPFL
Cluster filesystem:
/scratch
User


Shared filesystems:
/home
/work
Scheduler
HPC - cluster
Login nodes (font-end)
EPFL
Compute nodes (back-end)
Cluster filesystem:
/scratch
Run
User

SLURM

What is SLURM

- Simple Linux Utility for Resource Management
- Job scheduler
- Soda in Futurama

Basic commands

- **sbatch** submit a job to the queue
- **salloc** allocates resources
- **squeue** visualize the state of the queue

SLURM: common options

SLURM options

- **-q** / **--qos=<QOS>** sets the quality of service (affects the scheduling priority). During the course set it to **--qos=math-454**
- **-t** / **--time=<HH:MM:SS>** set a limit on the total run time of the job
- **-N** / **--nodes=<N>** request that a minimum of **N** nodes be allocated to the job
- **-n** / **--tasks=<n>** advises SLURM that this job will launch a maximum of **n**, in the MPI sense
- **-c** / **--cpus-per-task=<ncpus>** advises SLURM that job will require **ncpus** per task
- **--ntasks-per-node=<ntasks>** number of tasks per node
- **--mem=<size[units]>** defines the quantity of memory per node requested

Need more help? Have a look at `https://scitas-doc.epfl.ch` and `https://slurm.schedmd.com/sbatch.html`

SLURM: common options

Or you can put everything in a file, e.g.

mysimulation.job

```bash
#!/bin/bash -l
#SBATCH --qos=math-454
#SBATCH --time=01:10:00
#SBATCH --nodes=2
#SBATCH --ntasks=72

./myprogram
```

and submit the job

```bash
$ sbatch mysimulation.job
```

Exercise 8: SLURM: first commands

To list all your running jobs

```
$ squeue -u <username>
```

To cancel a simulation

```
$ scancel <jobid>
```

The <**jobid**> can be found using squeue

To allocate a node

```
$ salloc --qos=math-454 --account=math-454
```

Exercise 9: SLURM: first commands

- Check the queue state
- Allocate one node with QOS `math-454`
- Try **srun hostname**, we will see this command more in detail in later exercise sessions
- Exit the allocation to not block resources: **exit** or Ctrl-d (**IMPORTANT !!**)

Exercise 10: SLURM: command sbatch

- Write a script that runs the hello world code

myscript.job

```
#!/bin/bash -l
./hello
```

- Try your script.
  note: *in general you should not try your codes on the front node*

```
$ sh myscript.job
```

Exercise 11: SLURM: command sbatch

- Submit your script with **sbatch** and adding the QOS

```
$ sbatch --qos=math-454 myscript.job
```

- Try

```
$ squeue -u <username>
```

- A file named `slurm-<jobid>.out` should have been created, check its content

Exercise 12: SLURM: command sbatch

- Add the QOS as an option directly in the script

myscript.job

```
#!/bin/bash -l
#SBATCH --qos=math-454
./hello
```

- Submit it again

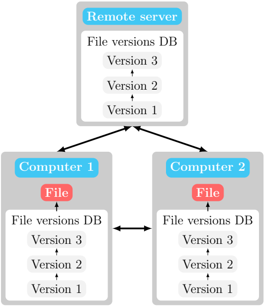
```
$ sbatch myscript.job
```

GIT

"Versioning": with Git

Git: *the stupid content tracker*

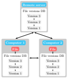
- Distributed revision control
- Originally developed by Linus Torvald
- Named after the *egotistical bastard* Linus



"Versioning": with Git

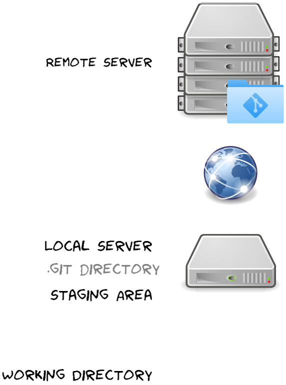
Git: *the stupid content tracker*

- Distributed revision control
- Originally developed by Linus Torvald
- Named after the *egotistical bastard* Linus



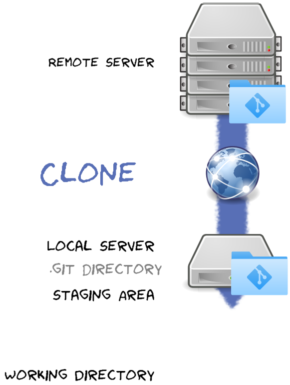
Git means "unpleasant person", "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." – Linus

git clone



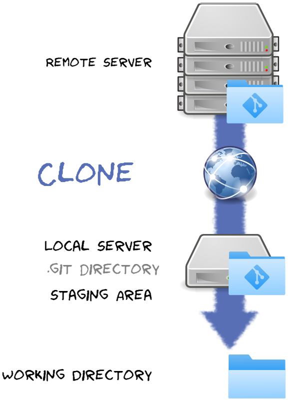

```
$ git clone <uri repo.git>
```

git clone




```
$ git clone <uri repo.git>
Cloning into '<repo>'...
remote: Counting objects: 6940, done.
remote: Total 6940 (delta 0), reused ...
Receiving objects: 100% (6940/6940), ...
Resolving deltas: 100% (3286/3286), done.
```

git clone




```
$ git clone <uri repo.git>
Cloning into '<repo>'...
remote: Counting objects: 6940, done.
remote: Total 6940 (delta 0), reused ...
Receiving objects: 100% (6940/6940), ...
Resolving deltas: 100% (3286/3286), done.
```

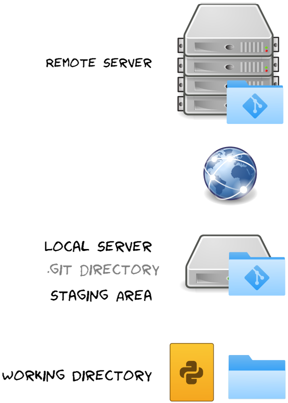
git status is your friend

```
$ git status
```

git status is your friend

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

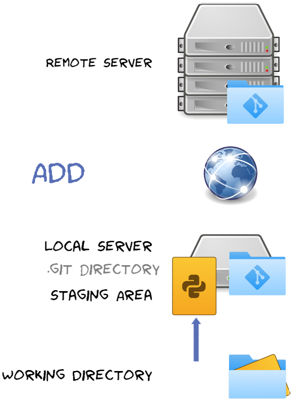
Lets add a file: staging/commit



Lets add a file: staging/commit

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
(use "git add <file>..." to include in what will be committed)

        my_code.py

nothing added to commit but untracked files present
```

Lets add a file: staging/commit




```
$ git add <filename>
```

Lets add a file: staging/commit

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
(use "git reset HEAD <file>..." to unstage)

        new file:   my_code.py
```

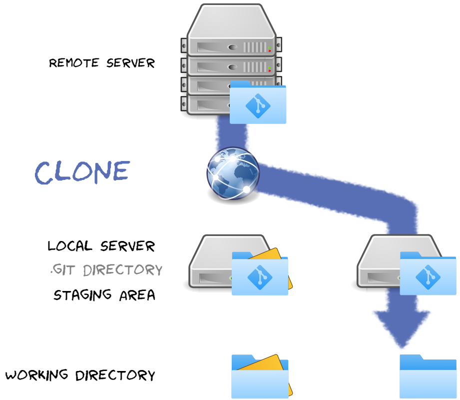
Lets add a file: staging/commit


```
$ git commit -m <message>
```

Lets add a file: staging/commit

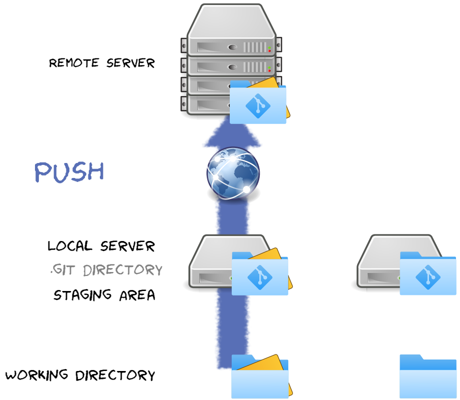
```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Synchronizing with the remote server



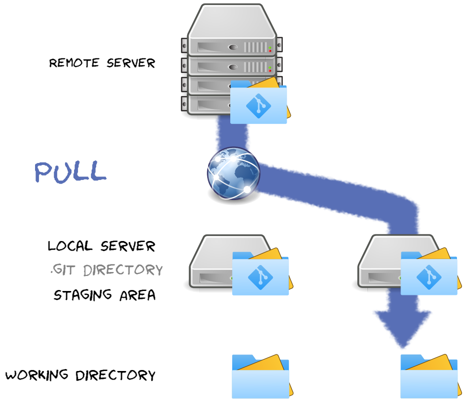

```
$ git clone <uri>
```

Synchronizing with the remote server




```
$ git push
```

Synchronizing with the remote server




```
$ git pull
```

Exercise 13: First step with Git

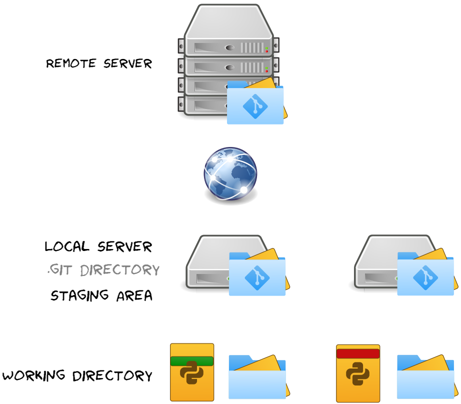
- If you do not have git installed, get it from `https://git-scm.com/downloads` or from your package manager
- Go on `https://gitlab.epfl.ch/` and login with your EPFL account.
- Once connected go on your user settings page (circle icon top left corner)
- In the **Preferences** > **SSH Keys** > **Add new key** menu set your public ssh key. This key will be used to connect to the git server through **ssh**.

Exercise 14: First step with Git

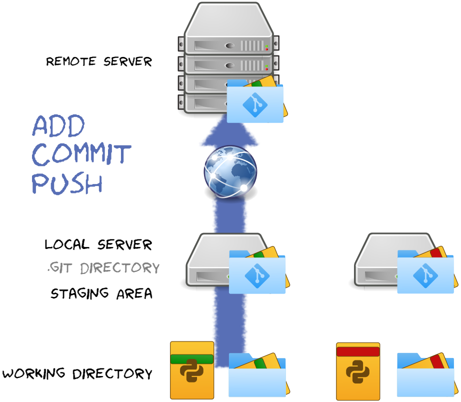
git clone <repo url> [local name]	Clone a remote repository
git add <files...>	Stage modified/new files
git commit -m "comment"	Commit staged files
git pull	Pull and merge remote modifications
git push	Push the local modifications to the remote server
git status	Check the local state

- Now you should be able to clone a repository
  Either create a repository or clone **git@gitlab.epfl.ch:math454-phpc/test-repo.git**
- Create a file, use a filename that will not clash with the others
- Check the state of your working copy

Collaborative work with potential problems

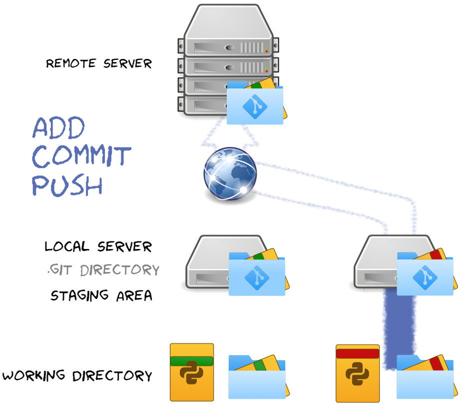


Collaborative work with potential problems




```
$ git add <filename>
$ git commit -m <message>
$ git push
```

Collaborative work with potential problems

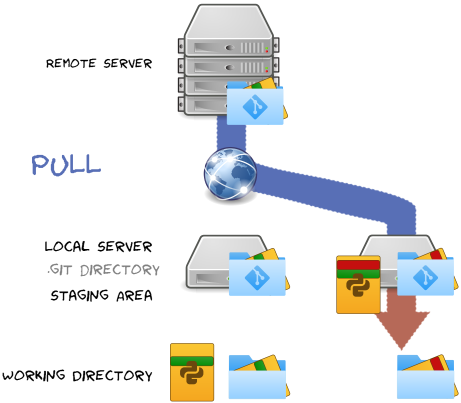



```
$ git add <filename>
$ git commit -m <message>
$ git push
```

Collaborative work with potential problems

```
$ git push
To <repo>
! [rejected]        master -> master (fetch first)
error: failed to push some refs to '<repo>'
hint: ...
```

Collaborative work with potential problems




```
$ git pull
```

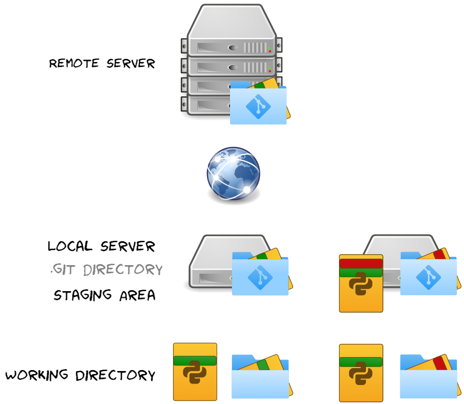
Collaborative work with potential problems

```
$ git pull
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From <repo>
fe22d81..0bcfb99  master     -> origin/master
Auto-merging my_code.py
CONFLICT (content): Merge conflict in my_code.py
Automatic merge failed; fix conflicts and then commit the result.
```

Collaborative work with potential problems

```
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)

You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)
```

Collaborative work with potential problems

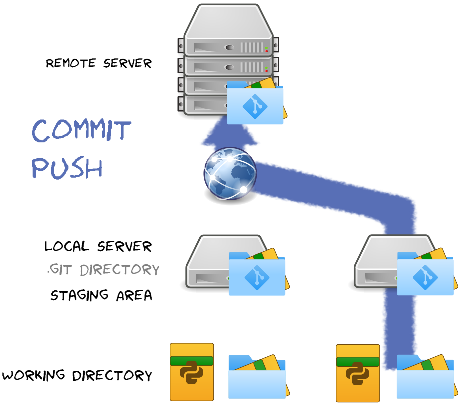


Correct the conflict:

my_file.py

```
<<<<<<<<<<
One version
==========
Other version
>>>>>>>>>>
```

Collaborative work with potential problems



my_file.py

```
One version
```

```
$ git commit -a
$ git push
```

Exercise 15: Generate and solve conflicts

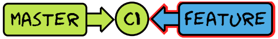
- Modify the file created in the previous exercise in both clones
- Commit this both modifications
- Pull and push in one of the clone
- Pull in the second clone, You should get a conflict

```
<<<<<<<<<<
One version
==========
Other version
>>>>>>>>>>
```

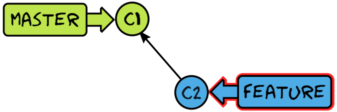
- Check the local status
- Correct the conflict and commit using **git commit -a**
- Push the modifications

Introduction to branches


```
$ git clone <uri repo.git>
```

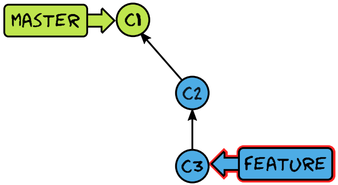



```
$ git checkout -b feature
```

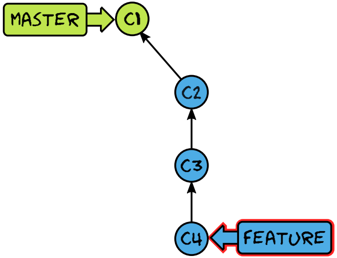



```
$ git commit -m <message>
```

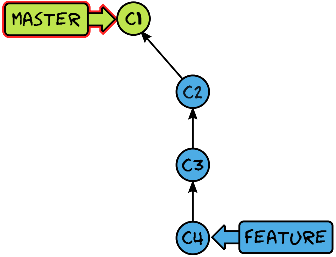
Introduction to branches




```
$ git commit -m <message>
```

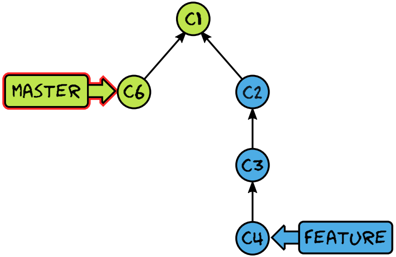



```
$ git commit -m <message>
```



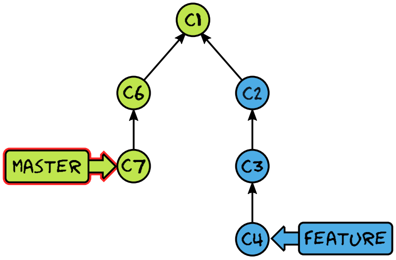

```
$ git checkout master
```

Introduction to branches

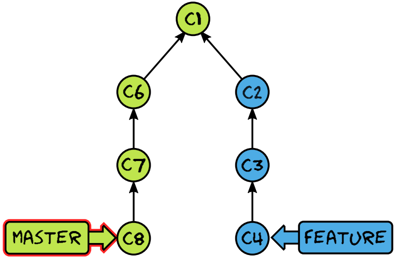



```
$ git commit -m <message>
```

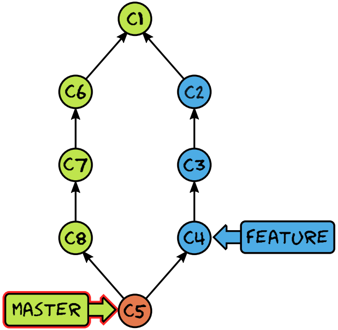
Introduction to branches




```
$ git commit -m <message>
```

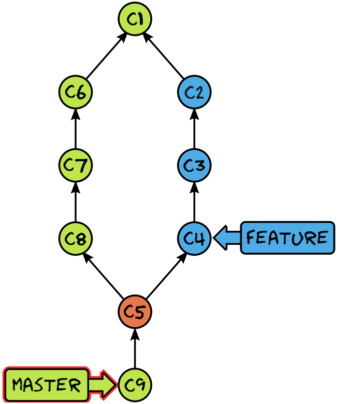



```
$ git commit -m <message>
```



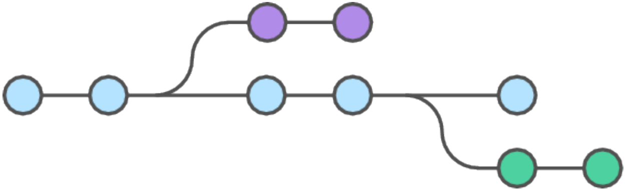

```
$ git merge feature
```

Introduction to branches

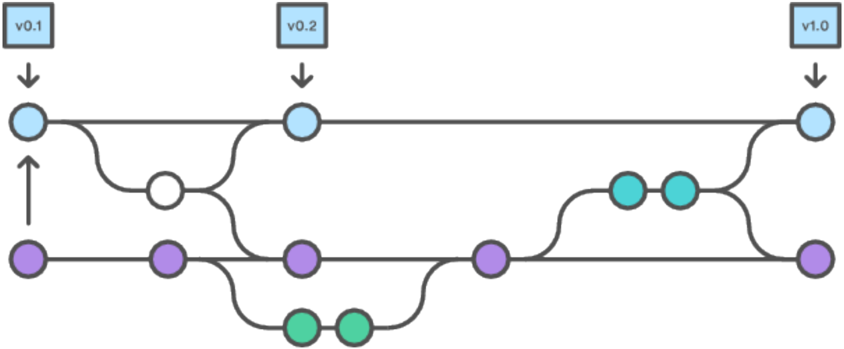



```
$ git commit -m <message>
```

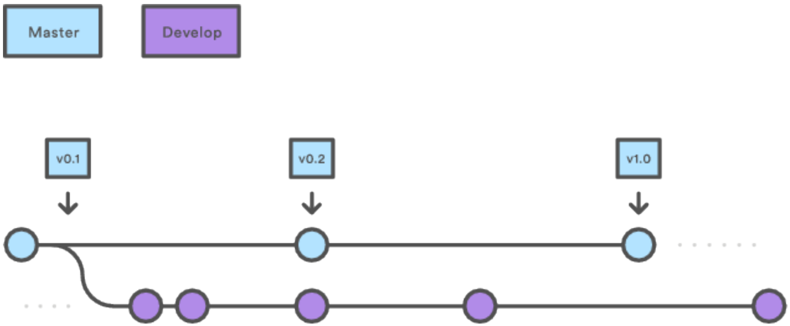
Feature branch



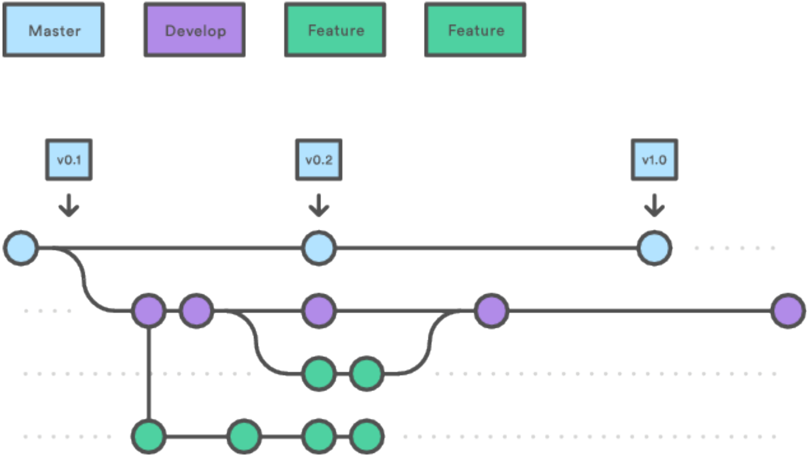
Gitflow



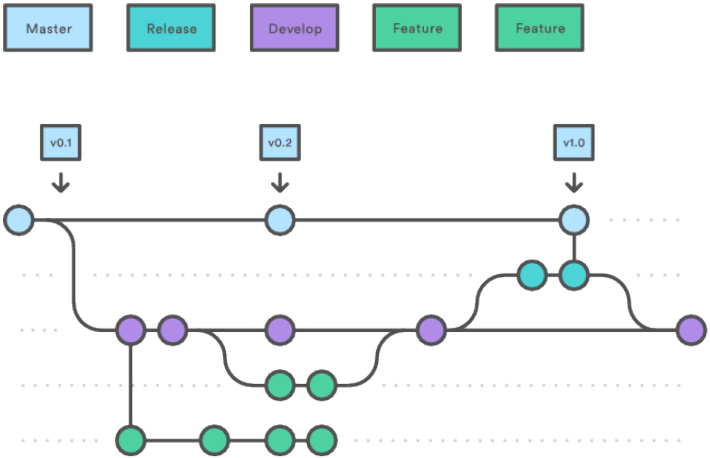
Workflow: gitflow



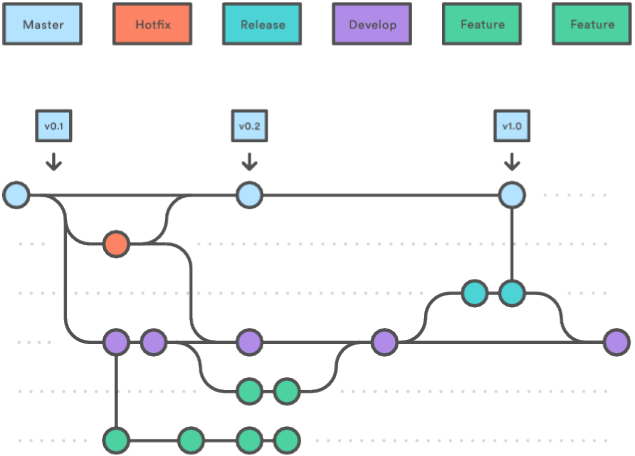
Workflow: gitflow



Workflow: gitflow



Workflow: gitflow



Exercise 16: Branches/merges

git branch <name>	Create a new branch from the current HEAD
git checkout <name>	Switch to the specified branch
git merge <name>	Merge the branch specified in the current one
git branch -d	Delete a branch
git branch -a	List all branches
git log	List the different commits of the current branch
git log –graph –all	Show also the branches

Exercise 17: Branches/merges

- Create a branch with the name of your choice
- Modify a file and commit the changes
- Checkout the **master** branch
- Modify a file and commit the changes
- Merge the branch previously created in the **master** branch
- List all branches
- Print the logs of the different modifications

Sources and extra infos

Sources

- Wikipedia
- http://git-scm.com
- Manpages: rsync, git
- https://www.atlassian.com/git/
- http://nvie.com/posts/a-successful-git-branching-model/

Learn more

- Git with a game: http://learngitbranching.js.org/