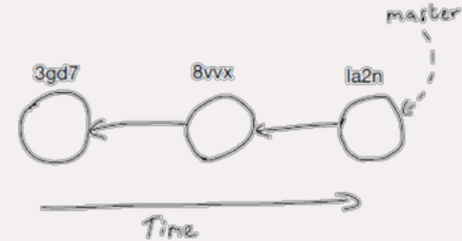
EINDHOVEN UNIVERSITY OF TECHNOLOGY

GIT

emc 2019

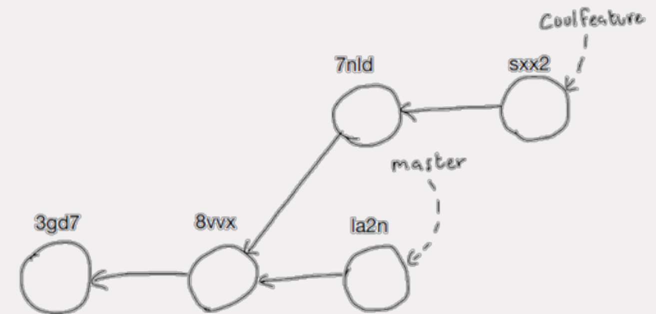
GIT version control

Using **git** is easier if you understand the underlying **representation**



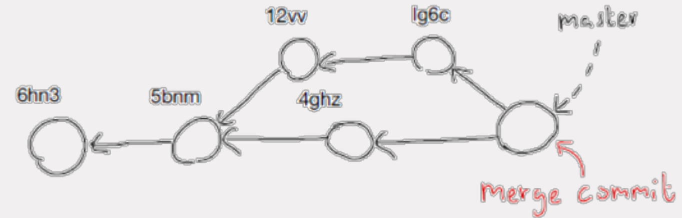
- Git is a graph of **commits**
- A **commit** represents the **state** of your repository
- Each **commit** points to its *parent(s)* ..

  .. and contains the *changes* (lines) w.r.t. that parent



- A **branch** has a symbolic pointer to a commit
- Checking out a **branch** checks out the **commit** pointed to
- Making a **commit** within a **branch** updates the pointer

- When you **merge** two branches, you either:
  - Fast forward
  - **Make a merge commit**



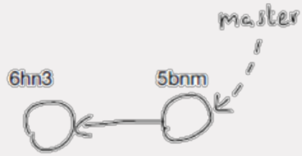
- A **merge commit** has two parents
- Sometimes **conflicts** must be solved first

- For the remote repository, the idea is the same
- **master** on remote is simply a different branch from yours

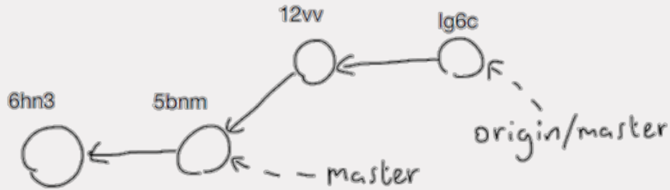
- git pull = git fetch (add branch **origin/master** to local repository)

  git merge **origin/master**

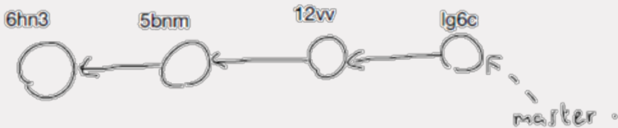
git pull origin master



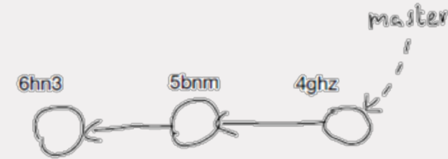

```
git fetch origin master
```



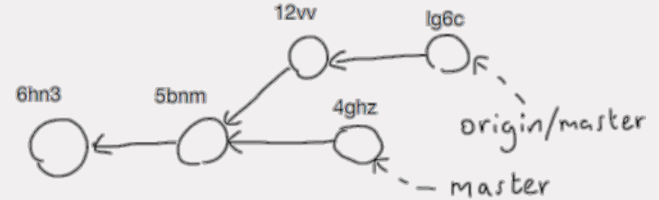
`git merge origin/master` → Fast forward



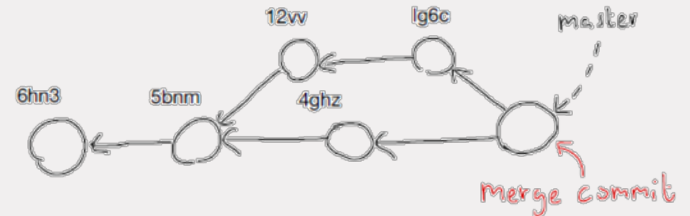
git pull origin master




```
git fetch origin master
```



`git merge origin master` → solve conflicts



TU/e

When merging

- Take a look at how the graph looks (`git log -all -oneline -graph -decorate`)
- Inspect differences after fetch (`git diff origin master`)
- `git merge` (and `git push` afterwards)

TU/e

Merge conflicts

- Git leaves you with a working directory of uncommitted changes (see `git status`)

- Conflicts are marked with conflicts markers:

```
<<<<<<< HEAD
//This line was added in the branch I'm currently in
float test = 1.0;
=======
//This line was added in the branch I'm merging
float test = 5.0;
>>>>>>> cb1abc6bd98cfc84317f8aa95a7662815417802d
```

- Solve the conflicts, delete markers and commit the files

TU/e

Gitlab workflow

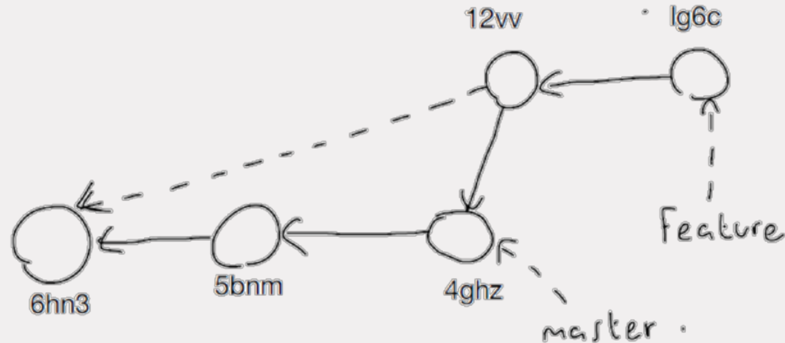
- **Collaboration** requires a **workflow strategy**
- Keep things simple and tidy: “***Don’t push directly to master, only merge into master***”
- Use a **feature/, fix/ or refactor/ branch** for all your coding
- Make a **merge request** on **Gitlab** when it works and assign someone
- When **approved**, **merge** it into **master**

Branches and merge requests

People can push commits to your branch before merging into master

Sometimes changes can be fast-forwarded, sometimes they can't.

You can always **rebase** *your branch*, but **never** rebase **shared branches !**



TU/e

Gitlab and Scrum

- Gitlab has **issues** and an **issue board**
- **Labels / milestones**
- Important commits → `git tag` (e.g. `escape_room_release`)
- (Advanced features like continuous integration and automated testing)

TU/e

Some tips

`git log --oneline --graph --decorate --all`

Use `git show` to see the changes made in a commit

Use `git grep` to search in your repository

Use `git blame` to see who committed each line in a file

HEAD points to the currently checked out commit, HEAD^ to parent commit

`git diff [ ] [HEAD HEAD^^] [master origin/master]` to see differences

`git reset (--hard)` – reset branch pointer to old commit (--hard changes the files in working dir)

`git revert` – make a new commit that undoes changes

TU/e

How to handle changes on the robot ideally?

- Quickly edit files on robot using **vim / nano** in terminal
- You want to push / pull but git won't let you?
- Commit changes to a branch (`git checkout -b fix/test1`)
- Push branch to gitlab and cleanup / merge after testing on your laptop

TU/e

When things get confusing during testing (git wont let you pull/push, not easy to fix now)

- Fix the state of your code on a laptop
- Check that it compiles with `cmake .. && make` on your laptop
- Push to gitlab master (or branch..) from your laptop
- (Optionally) push the changes on your robot to a branch to keep them somewhere
- Reset the master on your robot to a point before things got messy (`git reset --hard HEAD^^^^..`)
- (You loose uncommitted changes!)
- On your robot: `git pull origin master`