RCS & Git
Part II

Spring 2015

Alfred Bratterud

Agenda:

* Recap
* More advanced git
    * Branches & merge conflicts
    * Fork & Pull
* (Git on the server)
* (Tags & Rollback)
* Exercises

A note on «best practices»:

* Best practice regarding best practices:
    * Get the reasoning behind it. Don't just accept a dogma.
        * If it's really *best* practice, there will be a good lesson in there.
* Git best practice (for this course):
    * Use it
* Github best practice: Use it very carefully. Consider pushing to github as putting it in the newspaper.

It's really important

The Joel Test

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

From http://www.joelonsoftware.com/articles/fog0000000043.html 11-01-2015

Why so important?

* Every change results in a new, unique «state» or «version», with a unique identifier.
  * We know exactly what each deployment contains
  * We know if they need a patch
    * We can easily roll back (or forth) to any other state
* Every change is associated with a timestamp and an author.
  * So we know who to blame - and they can learn
* Every change to a document is tracked - forever (by default).
  * We can trace bugs and progress. ...this worked at location N not N+1

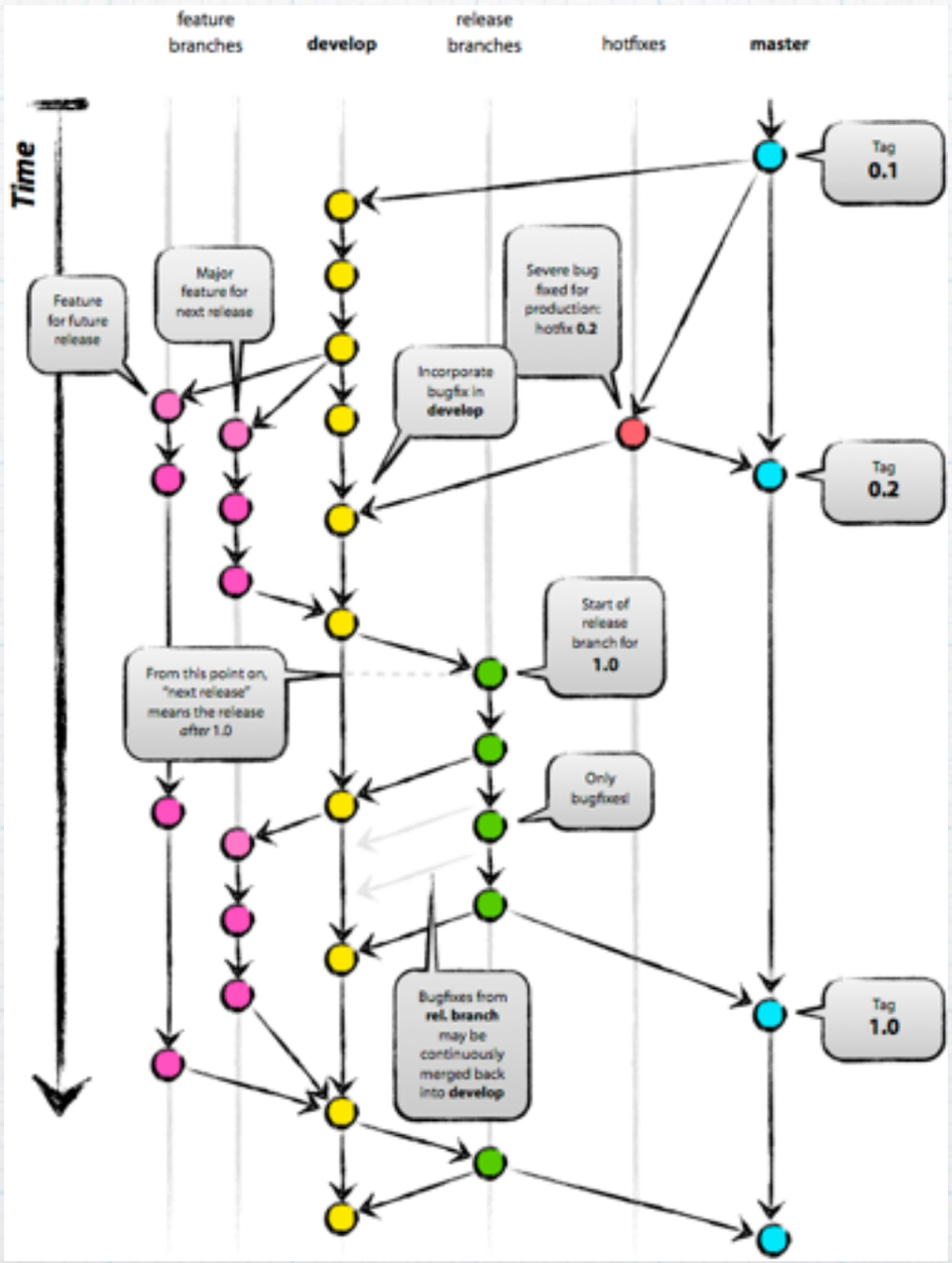
Centralized vs. Distributed RCS

* **Centralized:** Everyone commits to a server - «push»
  * Strict control over what the «Master» copy is
  * Strict control over committers
  * You need a server
  * Every commit requires a server connection
* **Distributed:** No pushing (necessarily) - everybody «pulls»
  * No «service» or server required by default - just a binary. But anyone can set up a «push» server / service if they want
  * No politics around «push rights» - nobody pushes. If you do - it's just to move the data to a server
  * No «Golden master copy» by default.

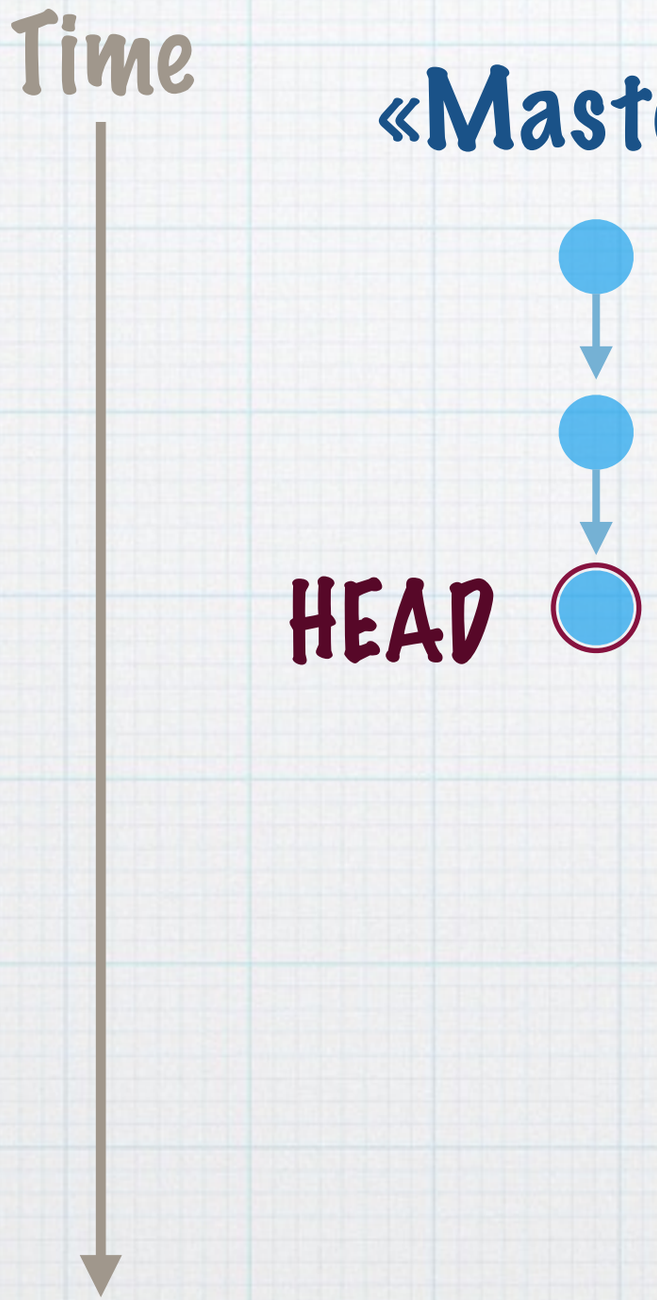
Basic Git commands

* `$ git init`
* `$ git status`
* `$ git add <pattern>`
* `$ git commit -am «Fixed this 1 thing»`
* `$ git log`
* `$ git mv`
  * Just like mv - but you get to keep you git history!
* `$ git rm`
  * Remove from git index - not only from current directory

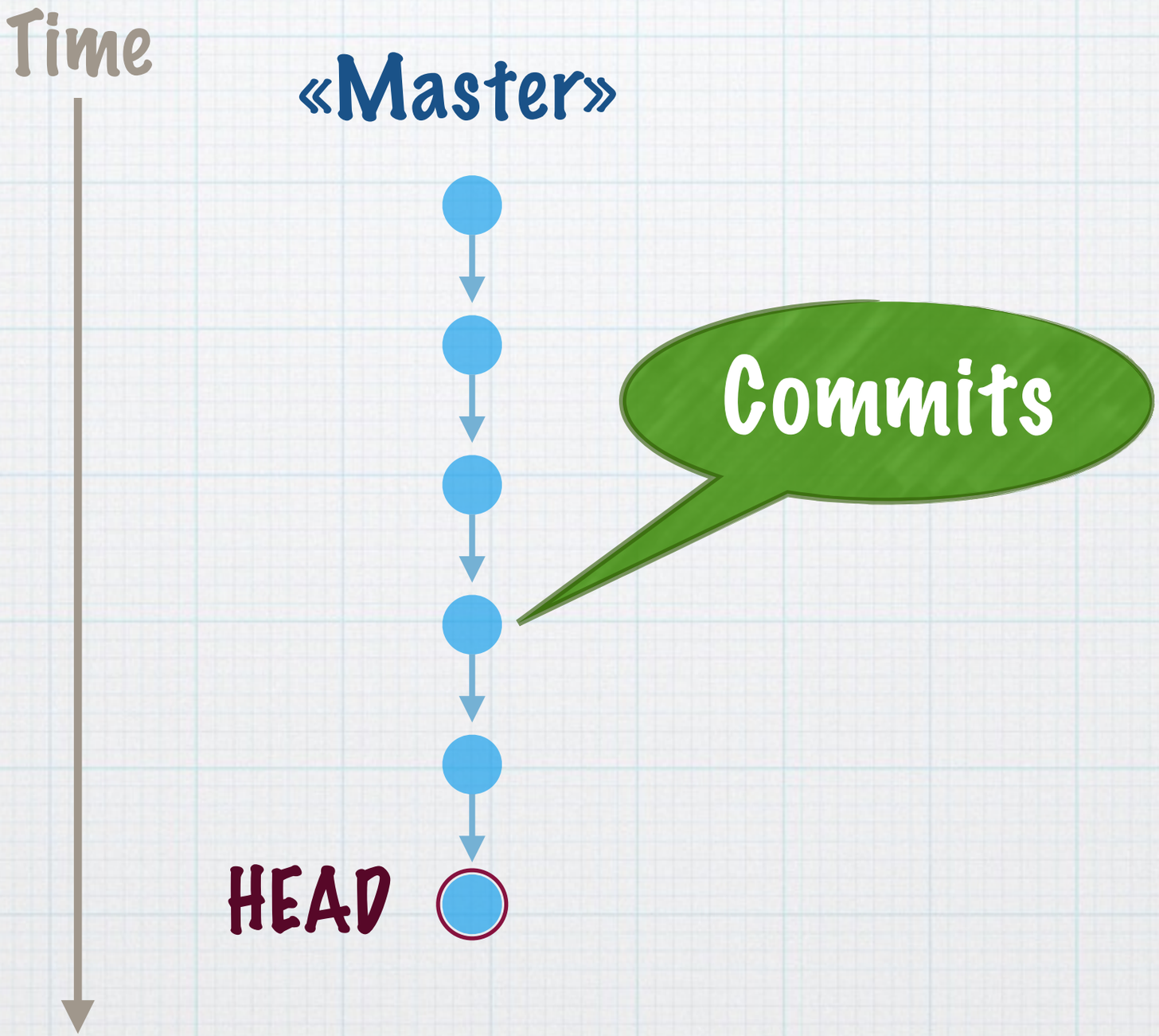
Branching


feature branches
develop
release branches
hotfixes
master
Time
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.1
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

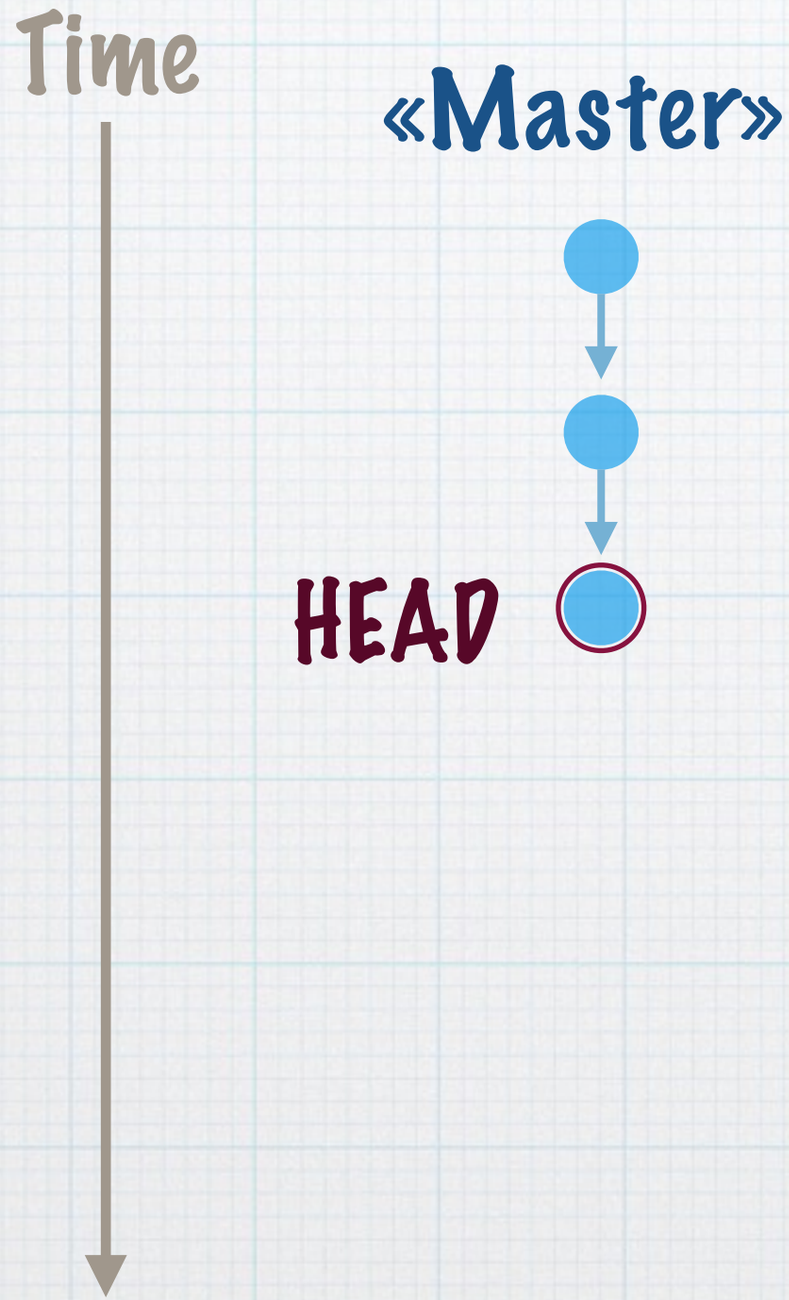
Single branch



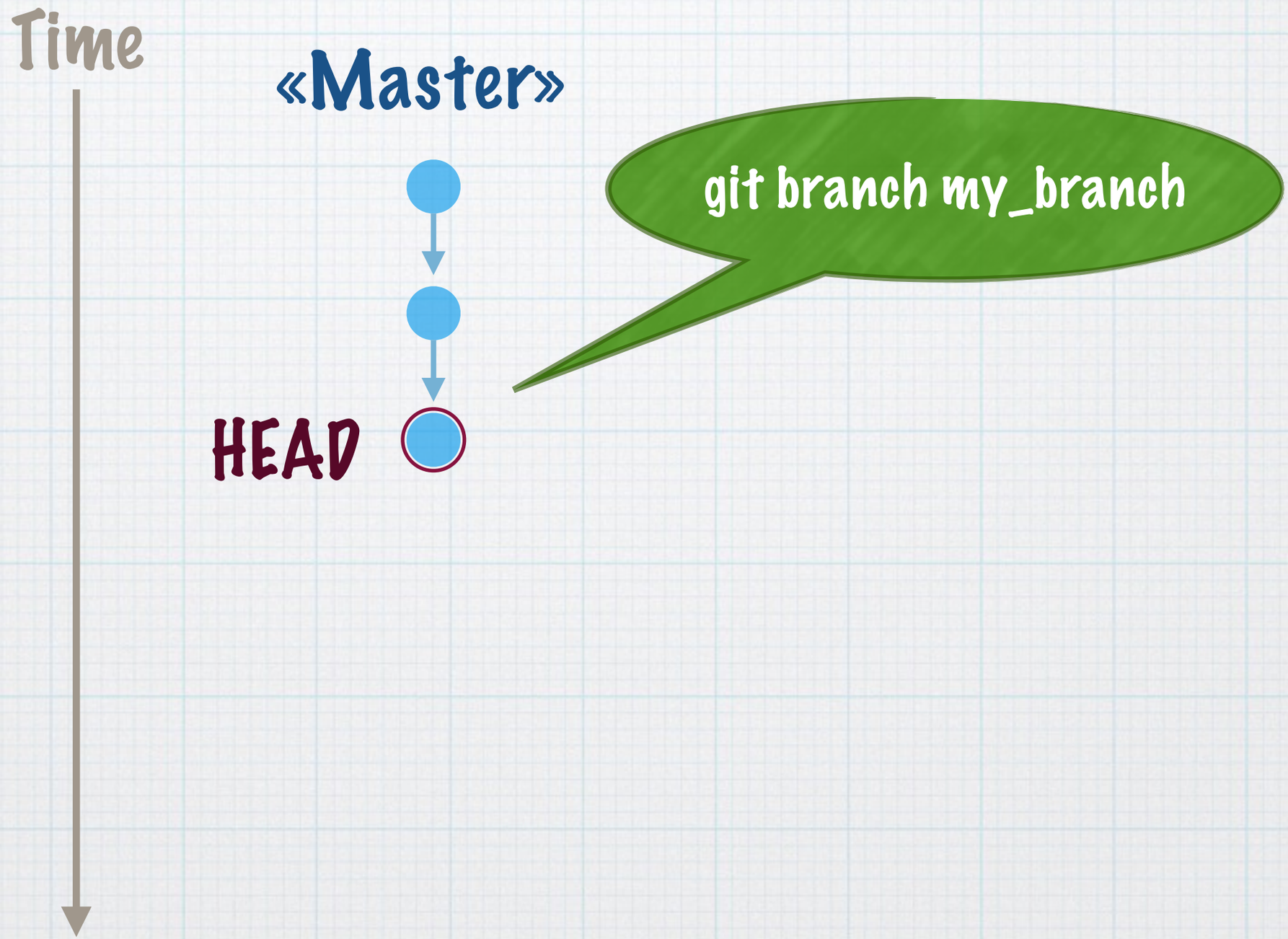
Single branch



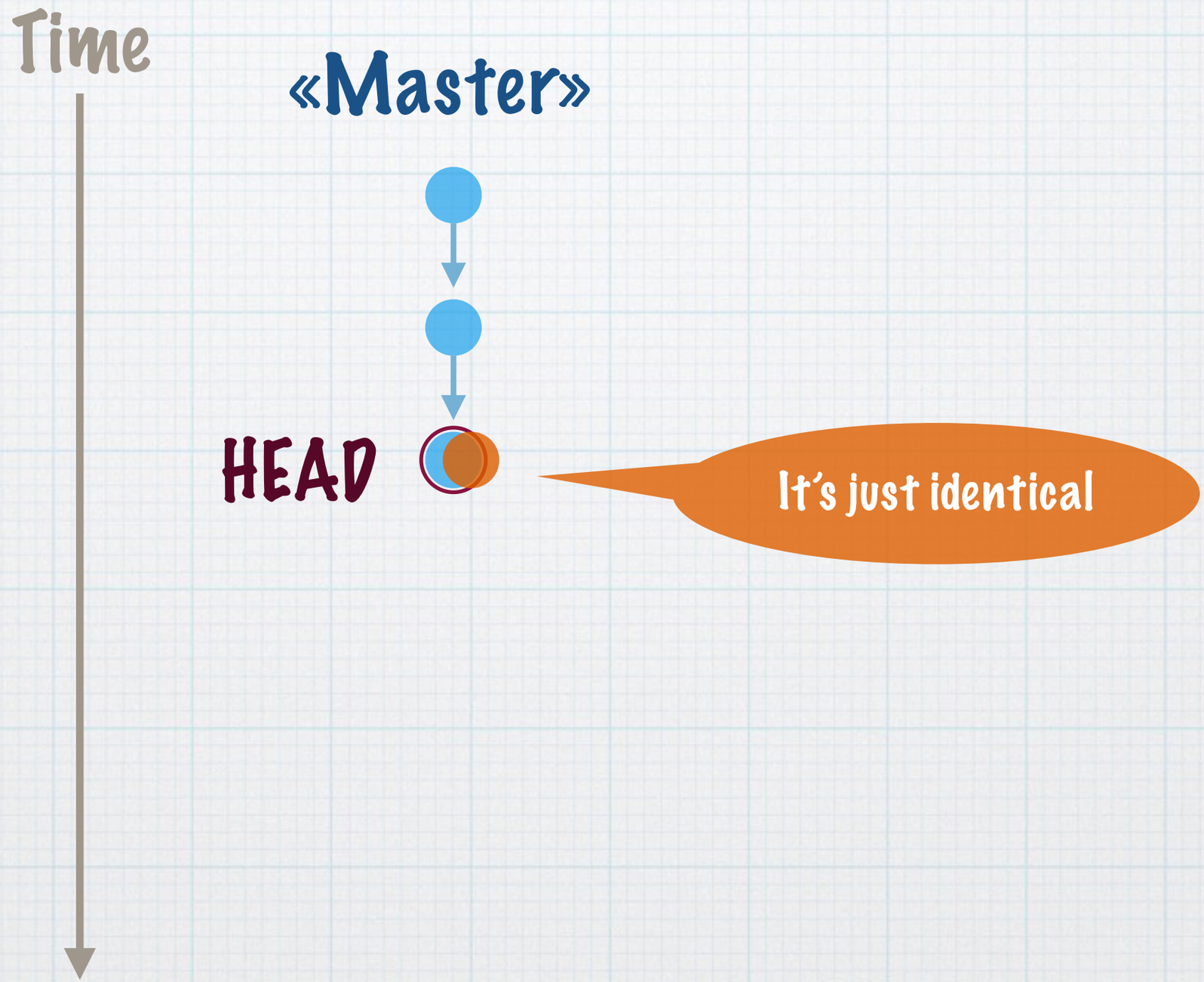
Multiple branches



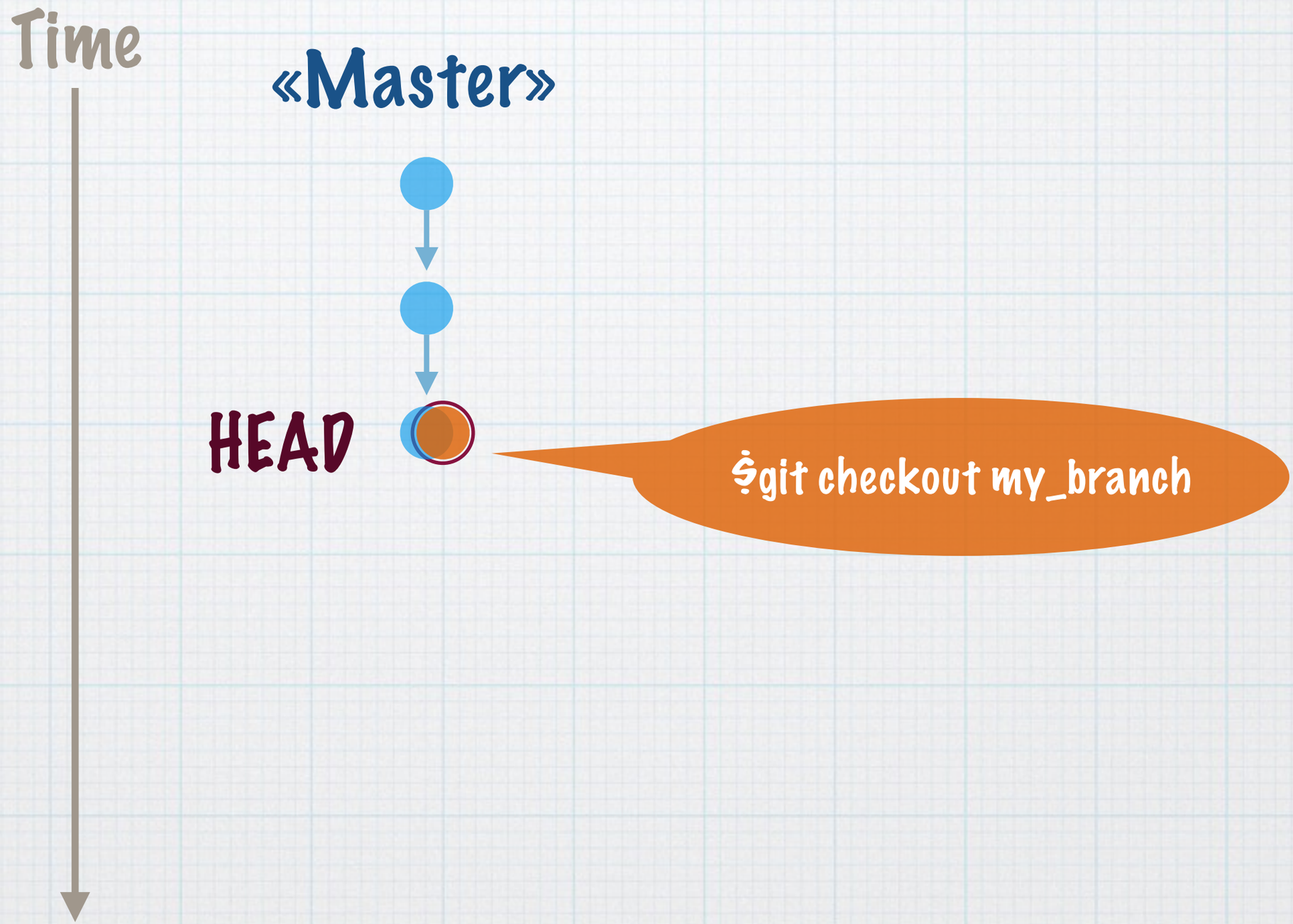
Multiple branches



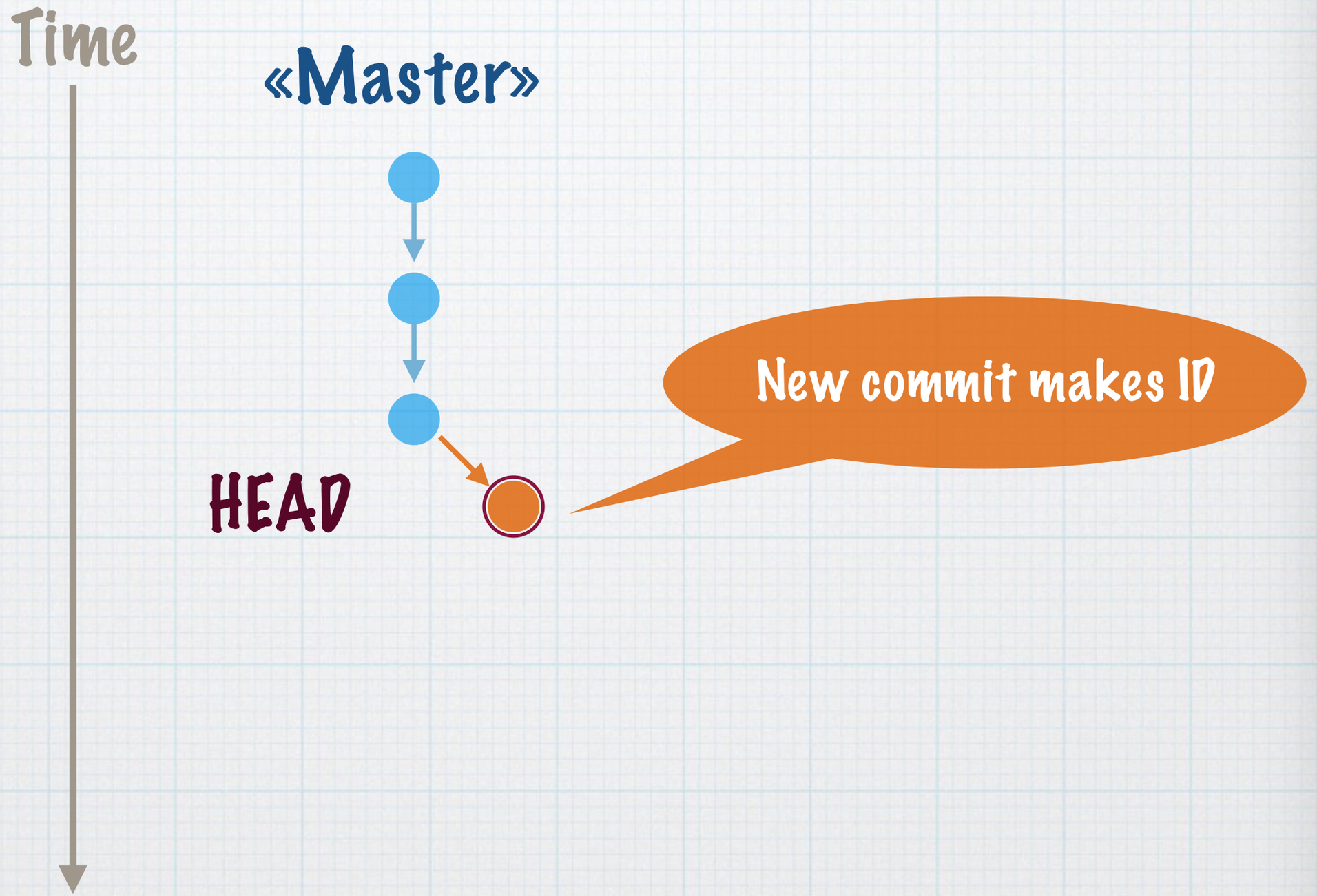
Multiple branches



Multiple branches

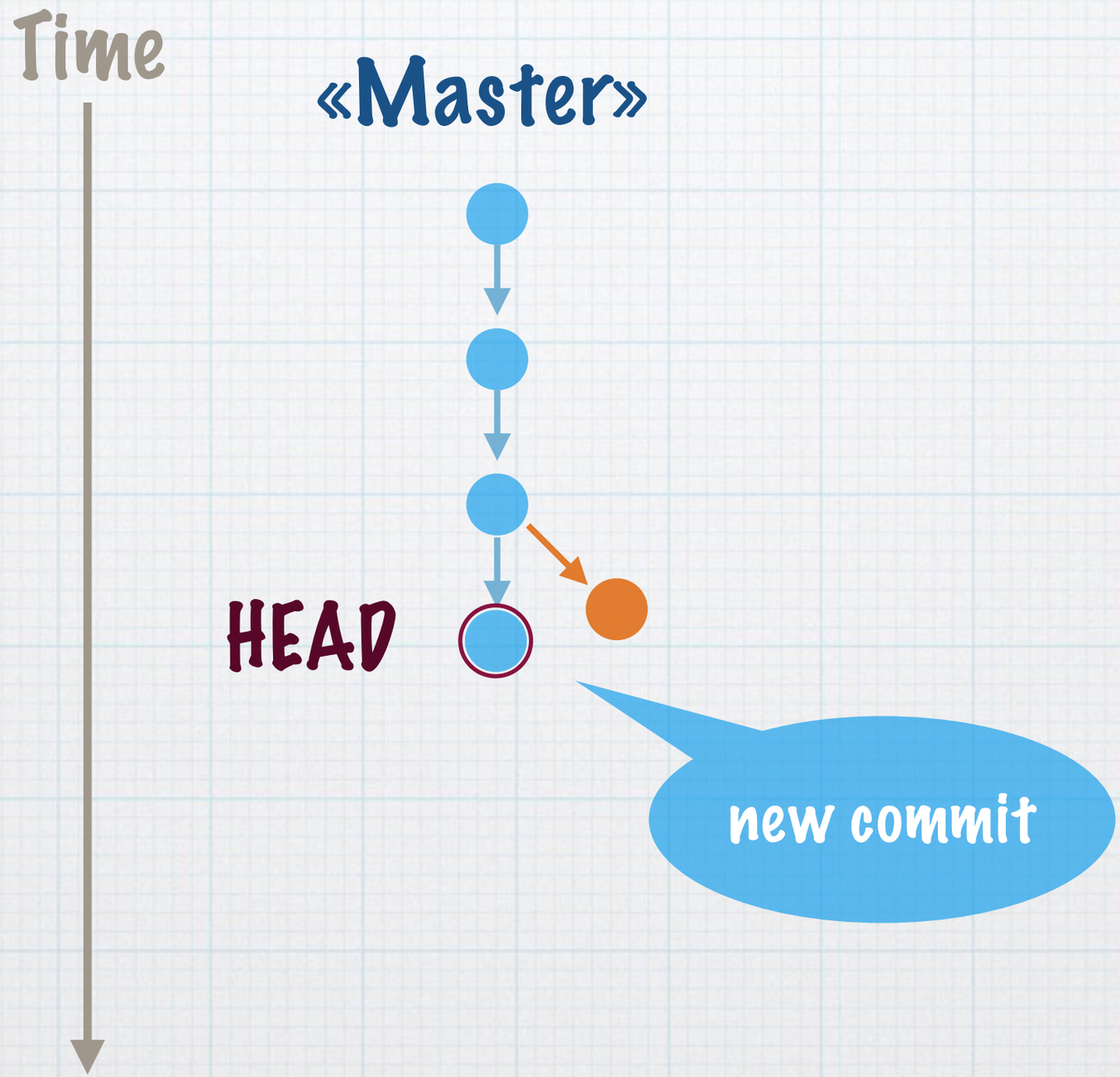


Multiple branches

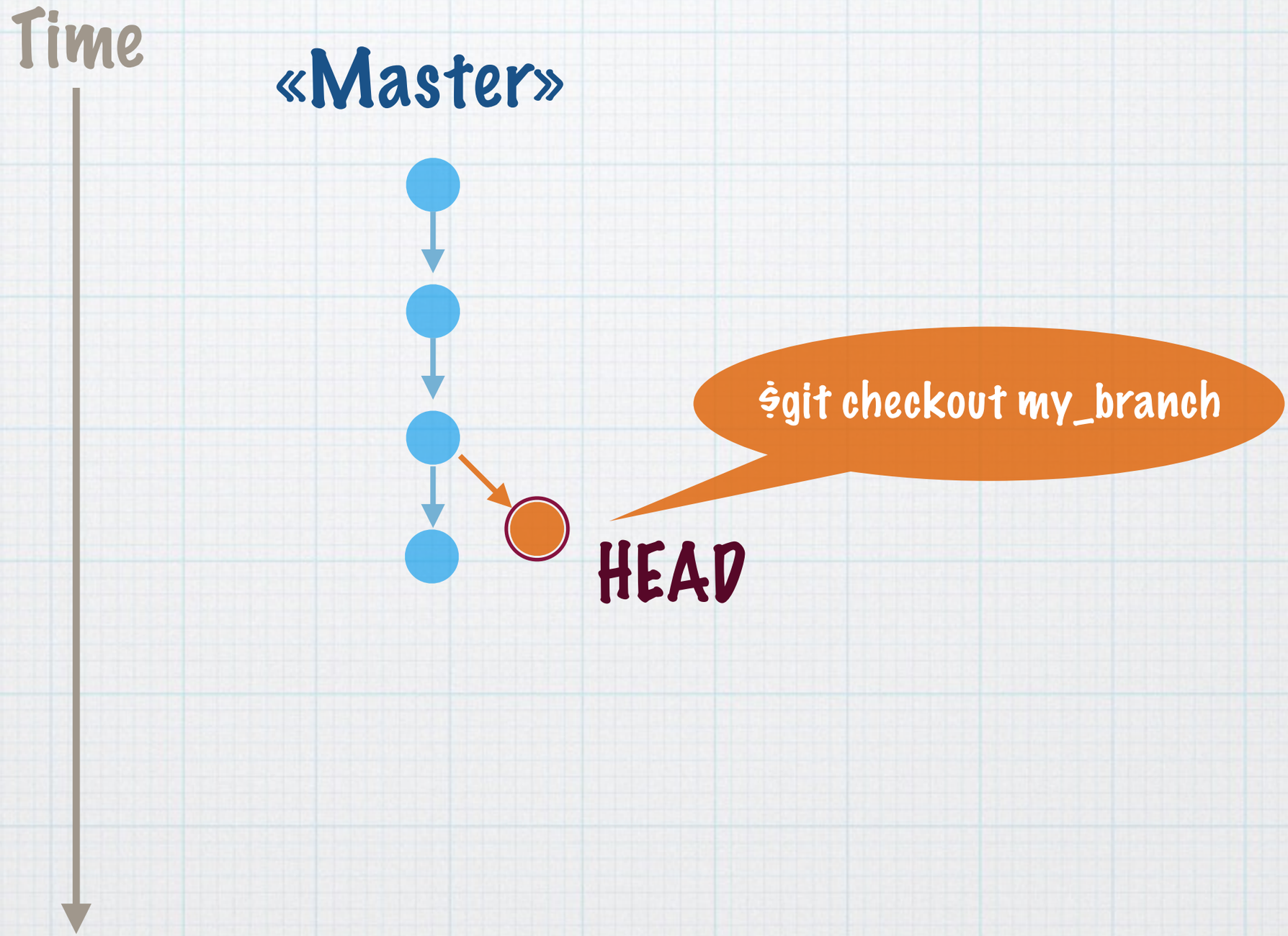


Multiple branches

Multiple branches

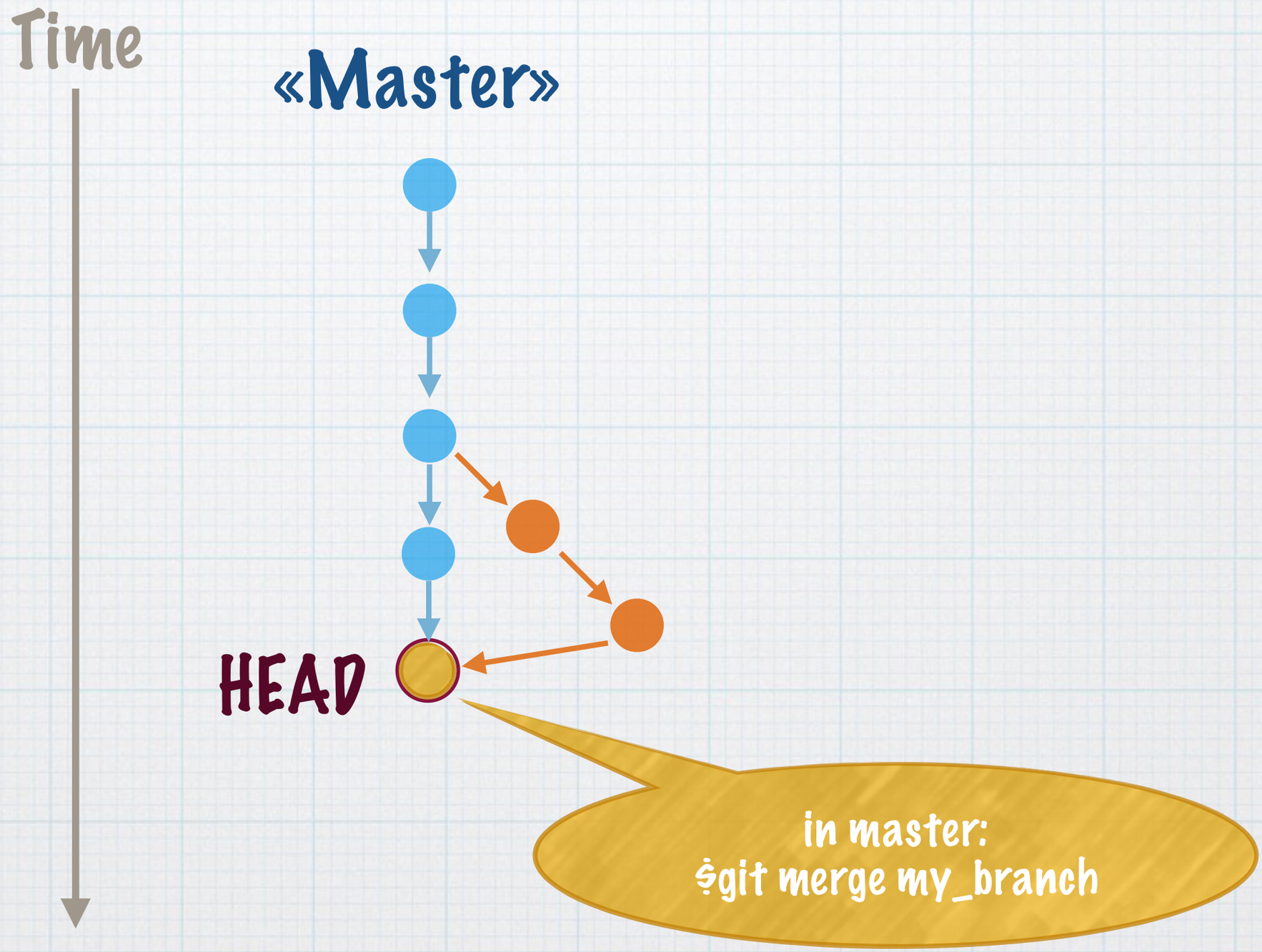


Multiple branches



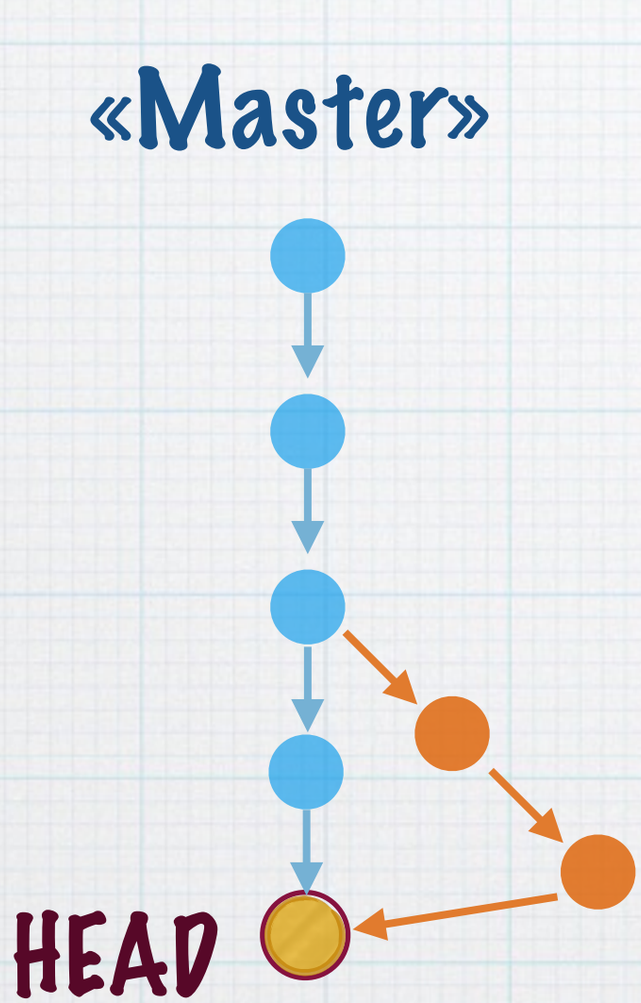
Multiple branches

Multiple branches



Merging branches

Time



* A merge joins two branches, resulting in a new commit
* If two branches modified the same file, there's a conflict
  * It must be resolved
  * Git status will tell you
* Otherwise «fast forward»: append all commits in branch, to master
* **Note:** content that was deleted from branch will now be deleted in master

Handling a merge conflict

* A) Merge with uncommitted changes in working tree: refused.
  * The working tree is «dirty» - it has uncommitted changes
  * Either commit, or `git stash` away everything since last commit
* B) Both branches contain committed changes to the same file: Real conflict.
  * Git status will show you the files with conflicts
  * Edit them, to the «correct» verison and commit
  * This is a manual job... so merging 100 files takes TIME
  * Avoid merge conflicts by agreeing on who works on what

Git branching commands

- Show branches with
  `git branch`
- Create a new branch with
  `git branch my_branch`
  - A new branch is identical to whichever branch you were in when you made it
- Switch to a branch with
  `git checkout my_branch`
  - Do changes, commit as usual
  - Switch back: The whole file tree is automagically reverted - bit-by-bit.
- Merge current branch with another:
  `git merge <branch_to_merge_with>`

...Let's try

Typical «Fork & Pull»

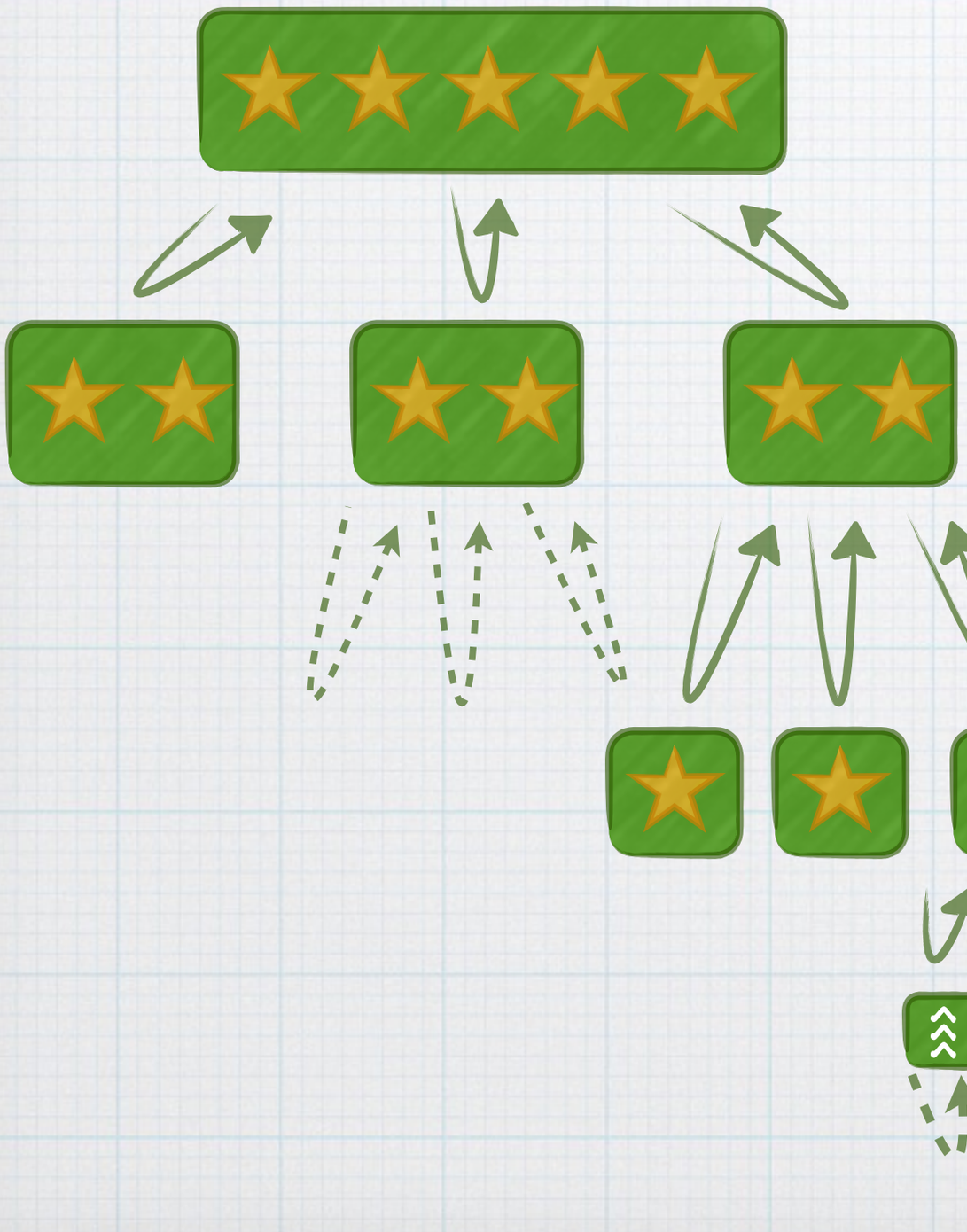
* Using both centralized and distributed properties
    * Only owner gets to push
    * Contributors fork, work on a clone, and send a «pull request»
        1. Fork the repo: it means, make a copy - and there's a button on github
        2. Clone your fork
        3. Do your changes
        4. Commit and push back to fork
        5. Send pull-request (i.e. ask «upstream» to pull from your fork)
    * Reuse the fork! Want to commit more?
        * Pull from original source (upstream) to get updates
        * Repeat from 3.
* Github / gitlab has nice facilities to automate pull-requests

...Let's try

Case: Fork&Pull in our project

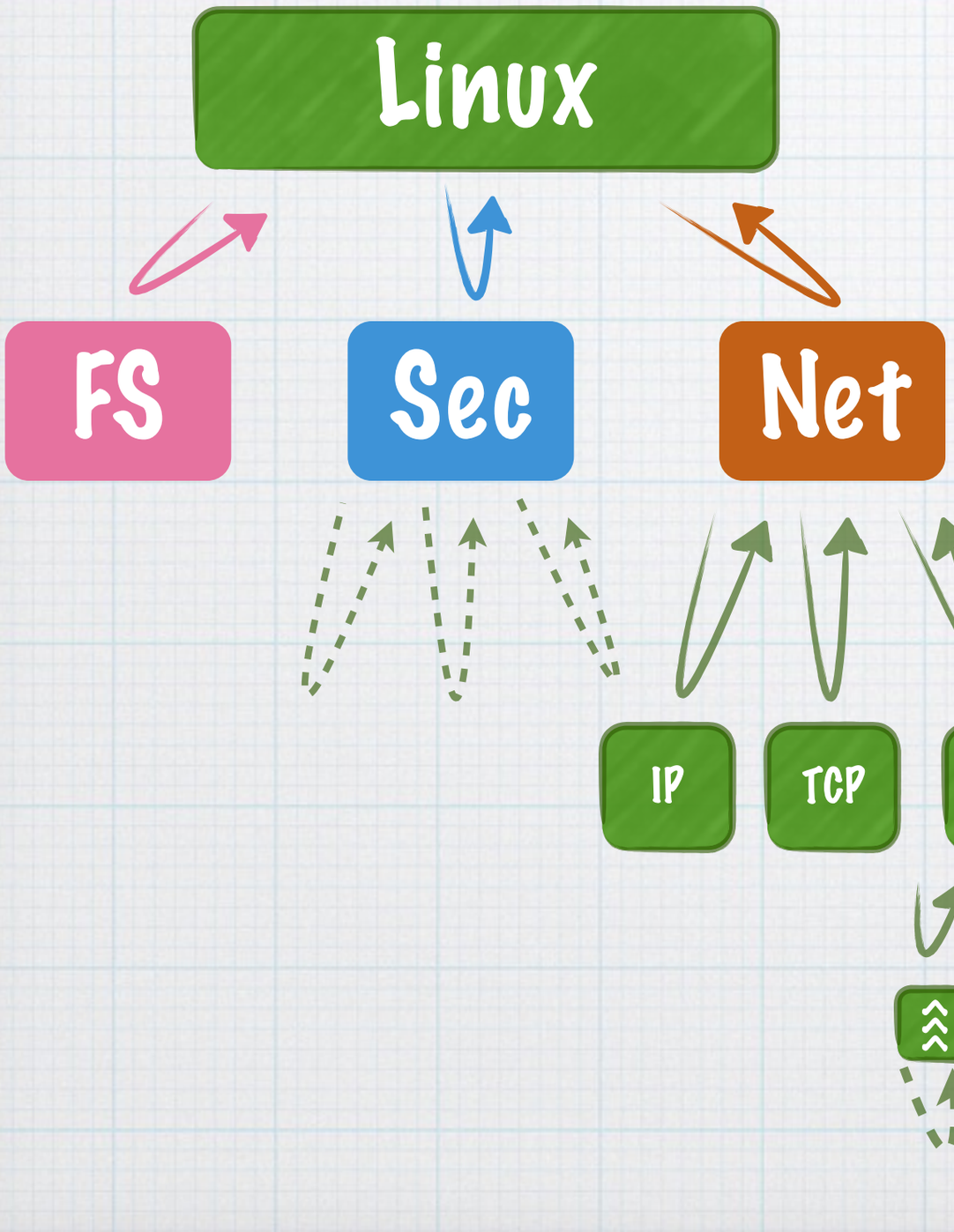
* Our research group uses github for a joint cloud computing OS project
* The ops' team notices big weaknesses in the build system
    * No need to agree beforehand - they just fork, and go to work
    * Whenever upstream changes, they refresh their fork
    * When happy, they submit a pull-request
    * SW-developers can now try it out
        * Discussion held directly on github
        * Changes? Further commits hooks into the pull-request
* New guy on the project? Sure - read-access like everybody else
* Distributed model makes it easy to have different fields collaborate

And it scales!



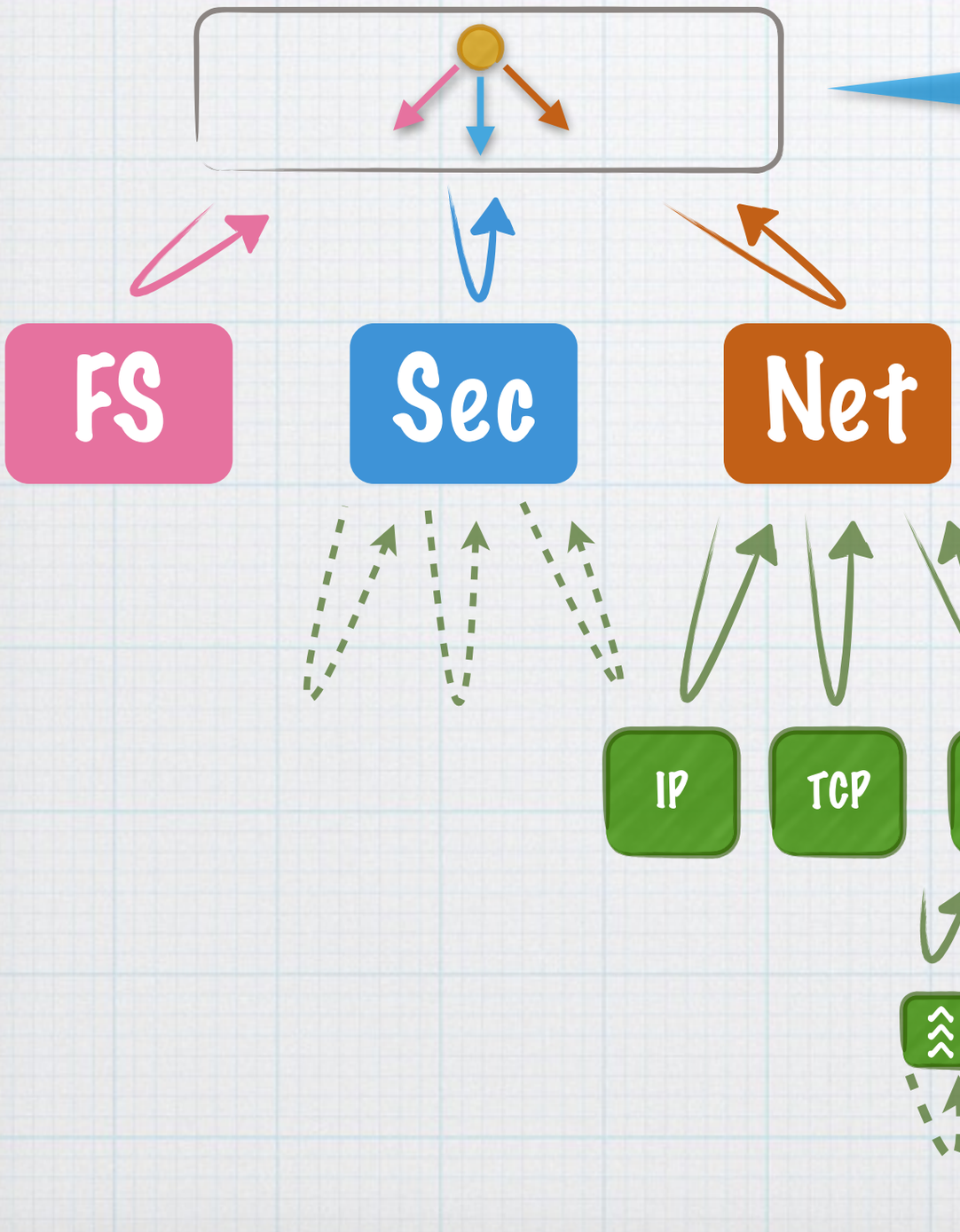
- Linus Torvalds only «pulls from his leutenants»
- They can in turn do the same - recursively!
- Huge, distributed organization
  - No «push politics»
  - Everybody Forks&Pulls
- Pro git: «Dictator Leutenant workflow»
- Genaralization of «Integration manager workflow»
- Github/Gitlab optional at every level

Topic branches



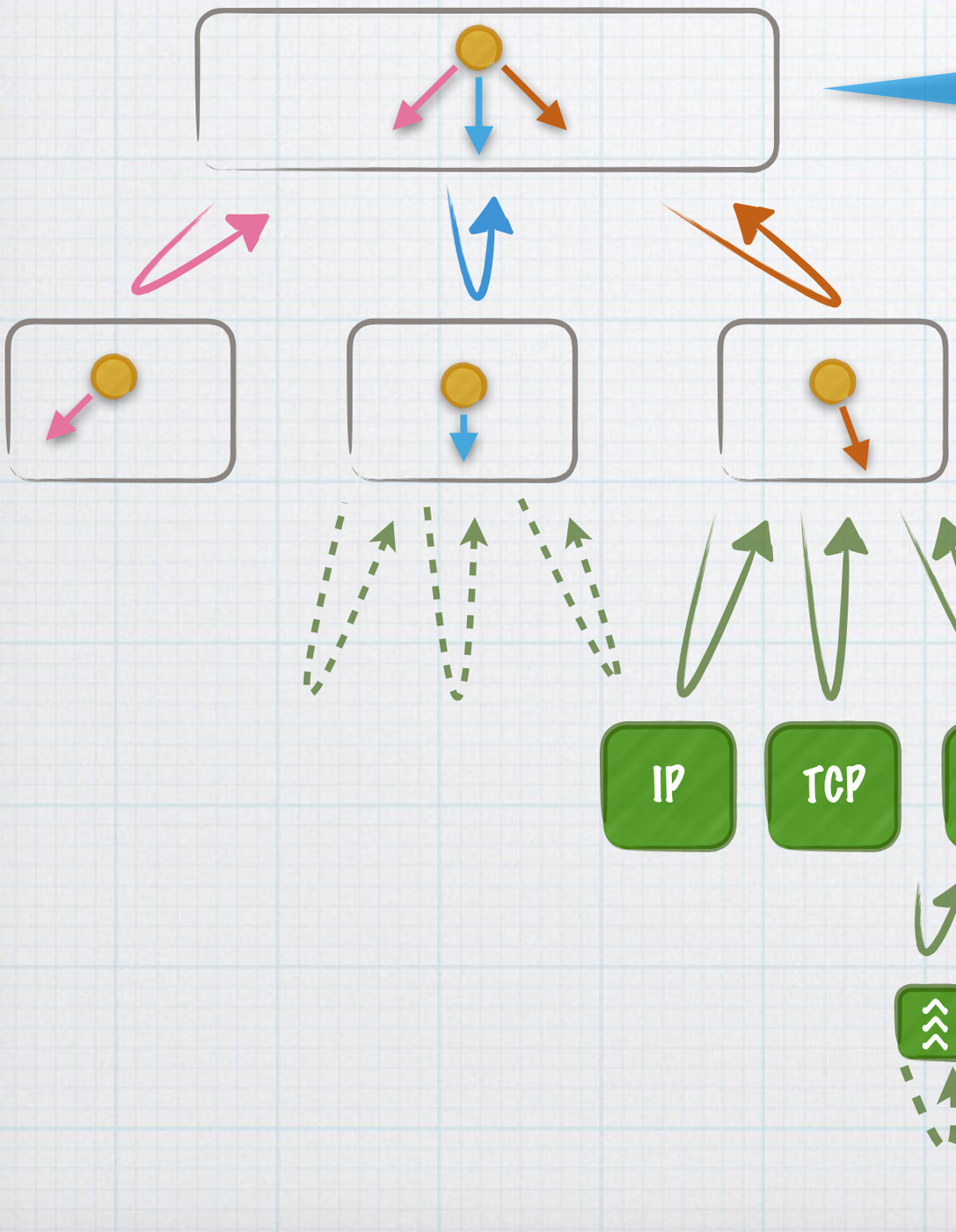
* Each level corresponds to a «topic», each with a leutenant
* Remember: remote branches are branches!
  * But you'd want corresponding local branches as well - at least some places
  * And Linus would merge each branch with «master» when he's happy with it

Topic branches



Do we need local branches here?

Topic branches

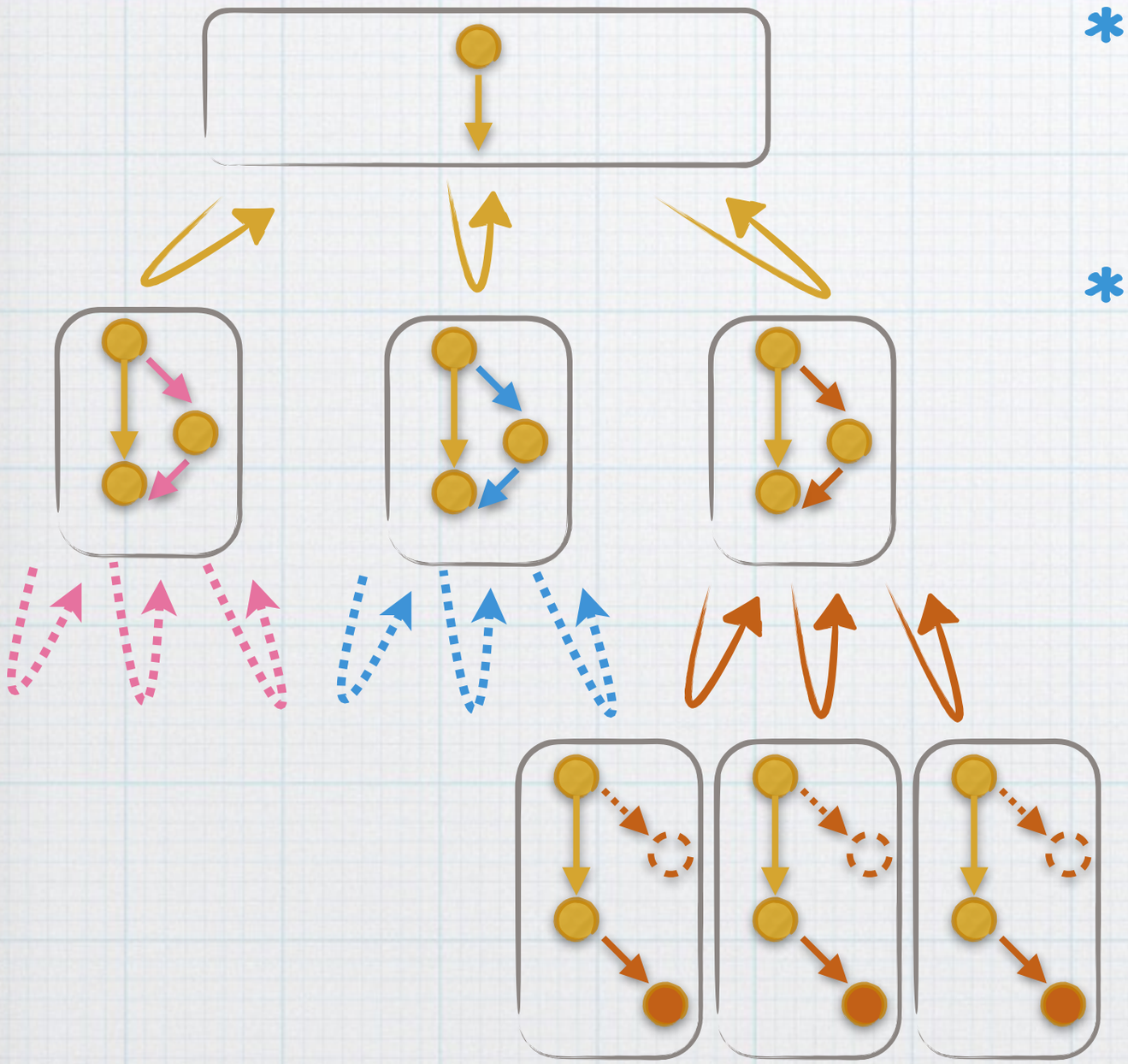


Do we need local branches here?

When you pull, you merge, so you don't have to first merge to a local branch, then to master - remote copies are branches!

NOTE:
Pulling only syncs one branch

«Dictator Leutenant»



* Dictator only has «master branch» locally. That's the authorative version.
* He merges directly from leutenants master branches
* ...Once the leutenants have merged in the «topic branches» from each developer
* The developer needs to continously «rebase» directly against «master» - i.e. root node.
* So: Leutenants are topic-wise «integration managers»

Pro git

A few more details on the workflows in the book

http://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows

Key concepts:

* Source control is not backup
  * Each change is saved, tagged and ID'd «forever»
  * Git supports active use of branch merge / swap
* Distributed v.s. Centralized
  * Corresponds to Pull vs. Push
  * Git gives you both
* The «git index» / source tree is separate from the file tree
  * Git index (commit tree) consists of historical versions of the «working tree»
  * If something is in your «working tree» (file tree) does not mean it's in the «git index» / source tree.
* Branching gives you exponential increase in possibilities
  * And - every remote copy is it's own branch - so you have to deal with it somehow
  * Facilitates Fork&Pull workflows, such as «Dictator Leutenants»

Bits 'n pieces

Get a knife with that fork!

The Joel Test

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

From http://www.joelonsoftware.com/articles/fog0000000043.html 11-01-2015

Get a knife with that fork!

The Joel Test

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

Included with github & gitlab - USET IT!

From http://www.joelonsoftware.com/articles/fog0000000043.html 11-01-2015

Get a knife with that fork!

The Joel Test

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

Included with github & gitlab - USET IT!

From http://www.joelonsoftware.com/articles/fog0000000043.html 11-01-2015

Remote Git commands

* `$ git clone <link>`
  Get a full copy of a remote repo, over ssh or https.
  You only want the full copy once.
* `$ git remote add <remote_name> <link>`
  * Add link as a remote branch, with name such as «upstream», «origin», «github» etc.
* `$ git pull <remote_name> <local_branch>` Means:
  * `$ git fetch <remote_name> <local_branch>`
    Download remote content to a «kind of hidden branch»
  * `$ git merge <remote_name>/<local_branch>`
    Merge remote content with current branch.
* `$ git push <remote_name> <local_branch>`
  * Just like forcing a «git pull» on the remote repository.

Exercises

ALL things GIT:
http://git-scm.com/doc

Exercises - Fork&Pull

* Everybody Fork&Pull Kyrres Repo
    * Creae an issue, requesting your name in the name list (namelist.txt)
    * Assign the issue to you
    * Fork & Pull
        * Add your name to the list, commit - #marking the issue number in the comment.
        * then push to your fork, and make pull request
* Implement the «dictator leutenant workflow» using virtual machines (using git rebase)

Exercises - Branching

* In a repository with a committed «README», create a branch, «new_feature»
  * Add one file in each branch, commit them, then merge
  * Delete README in the new branch, then merge. Is it gone from Master?
  * Add the README file again, in Master, and commit
  * Check out the new branch. Is it there now?
  * Add another file to new branch «NEW_BRANCH.txt», with some content. Commit.
  * Do «git rebase master». Do you have README now? Read up on git rebase in the man page.
  * Make changes to README in both branches, and commit.
    * Try to merge, from master, into the new branch
    * Resolve the merge conflict
    * Now, merge the new branch into master. Does this fast-forward? Should it?