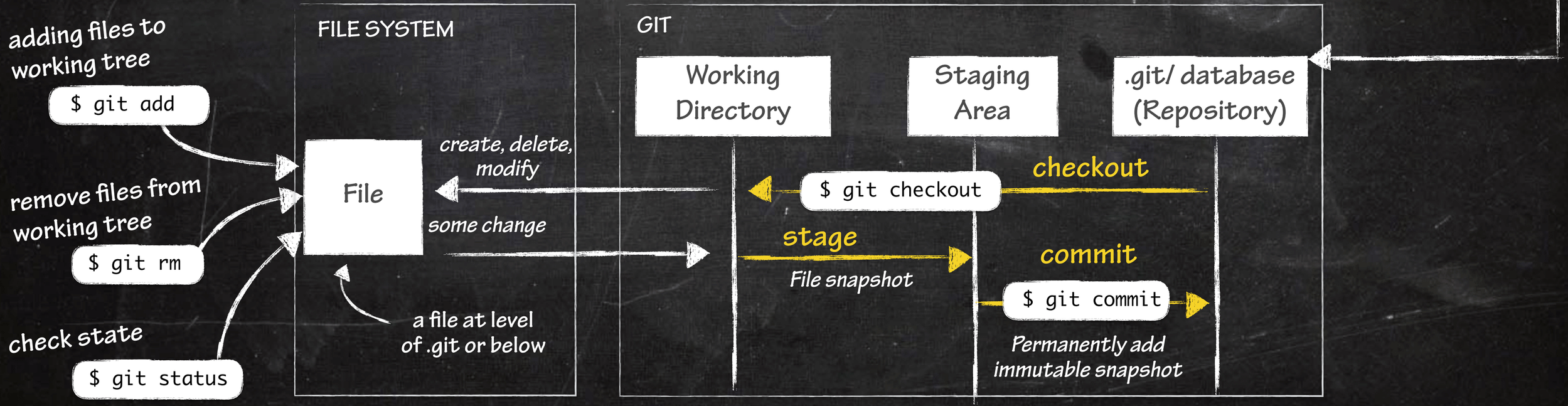
Local Git Repositories (4)

where does .git come from?

.git directory is created by `$ git init`, or
by getting a pre-existing .git from elsewhere
(e.g., from remote repository via `$ git clone`)

Staging Area (aka Index)

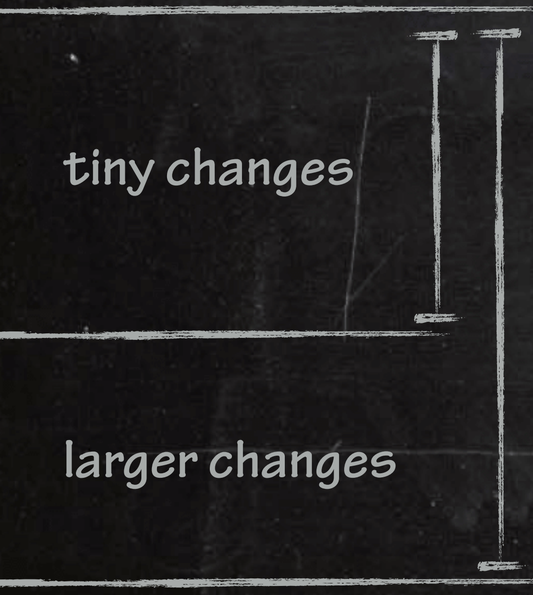
- Modifications on a local repository falls into one of the following stages
  - **Committed**: Changes on the files in the file system are also stored .git/ database
  - **Modified**: Changes on the files in the file system are *not* stored .git/ database
  - **Staged**: Changes on the files in the file system are intended for a later commit



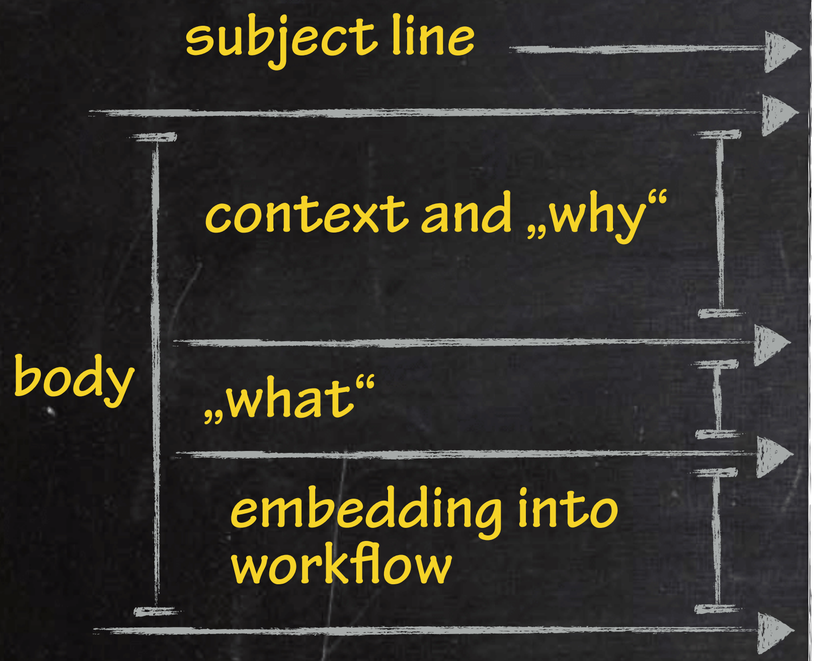
Good Commit Messages

When using `$ git commit` it's reasonable (and often forced) to add a message about the changes the commit introduces:

- The message should be descriptive and helpful
  - the intention of such message is to give follow-up devs *context* information!
  - describe *what* you did and *why* you did it (*not* the *how* in technical terms)
    - do *not* write about something `$ git log` (i.e., file diffs) can tell
  - A good message contains of a subject and body part separated by a blank line
    - subject (mandatory)
      - up to 50 characters
      - starting with a capital letter
      - imperative mood (i.e., „Fix“ rather than „Fixed“)
        - Completing the following sentence: „Apply this commit will…“
    - body (optional)
      - wrapped after max. 72 characters
      - explaining the why and what



Good Commit Messages (2)



Example

commit 6c0e7284d89995877740d8a26c3e99a937312a3c of the Linux Kernel.

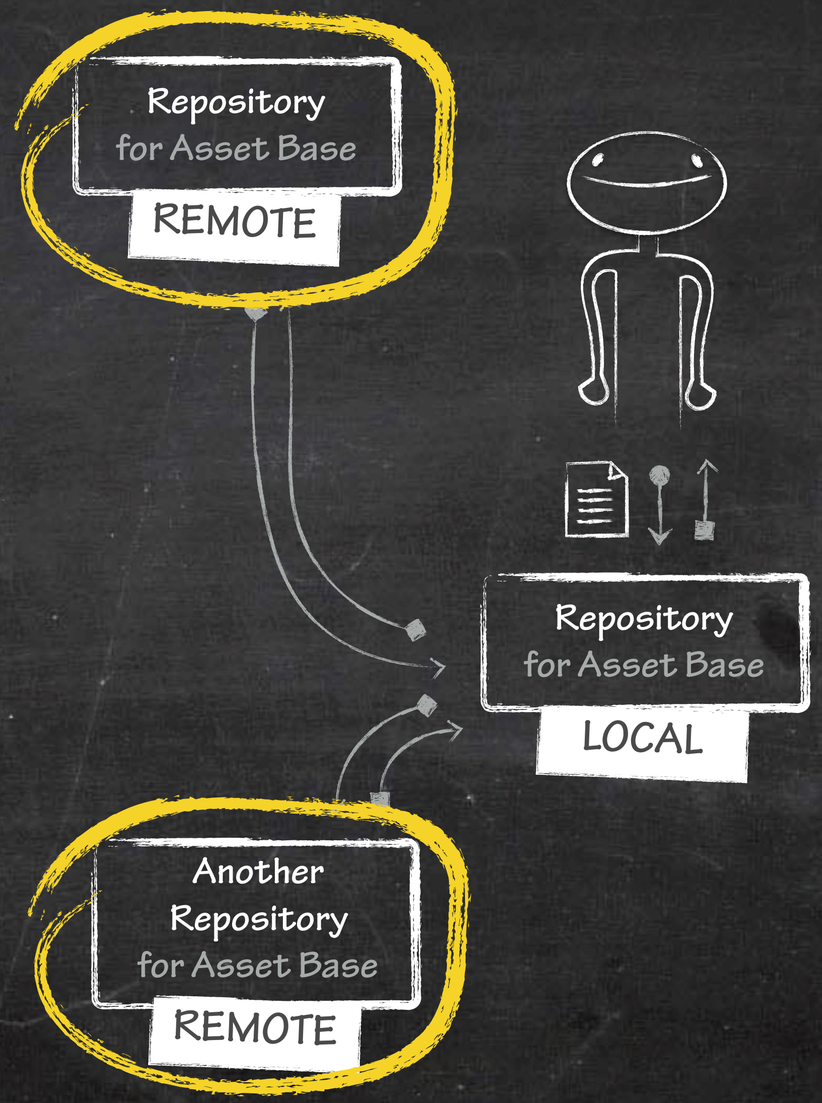
```
ipv4: Fix traffic triggered IPsec connections.

A recent patch removed the dst_free() on the allocated
dst_entry in ipv4_blackhole_route(). The dst_free() marked the
dst_entry as dead and added it to the gc list. I.e. it was setup
for a one time usage. As a result we may now have a blackhole
route cached at a socket on some IPsec scenarios. This makes the
connection unusable.

Fix this by marking the dst_entry directly at allocation time
as 'dead', so it is used only once.

Fixes: b838d5e ("ipv4: mark DST_NOGC and remove the operation of dst_free()")
Reported-by: Tobias Brunner <tobias@strongswan.org>
Signed-off-by: Steffen Klassert <steffen.klassert@secunet.com>
Signed-off-by: David S. Miller <davem@davemloft.net>
```

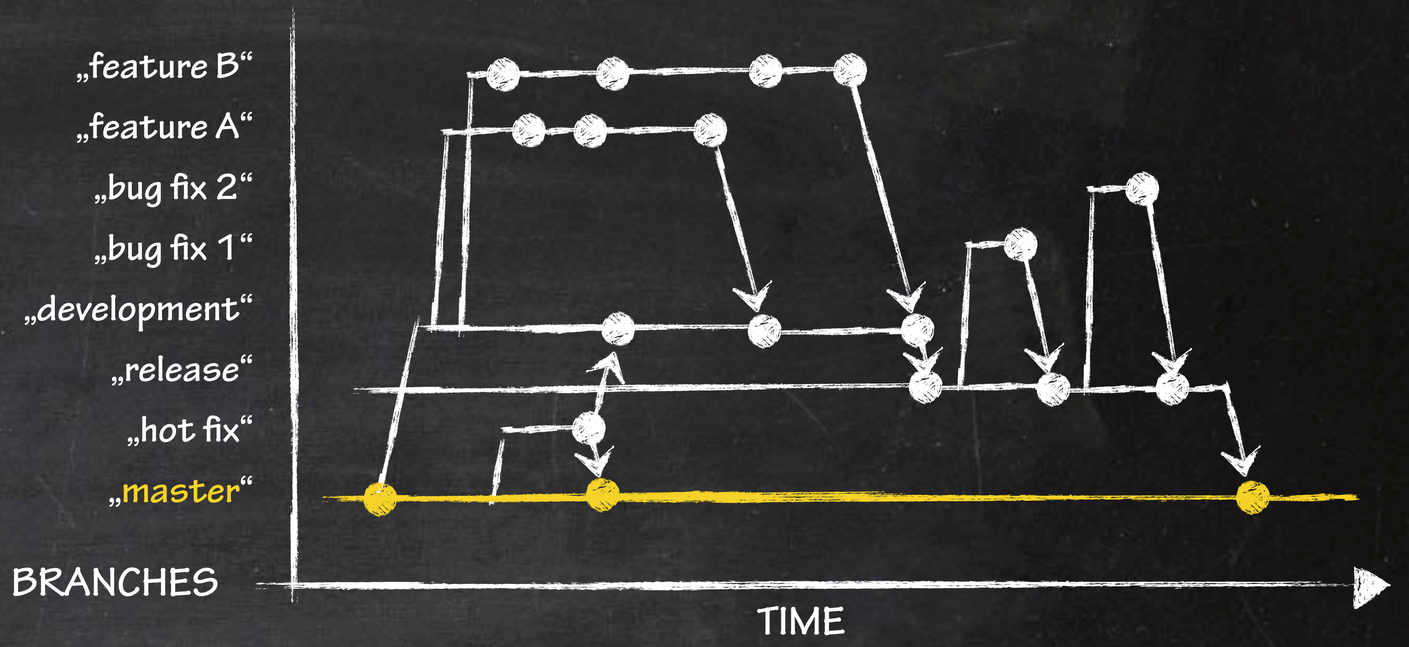
Remote Git Repositories



Remote Git Repositories (2)

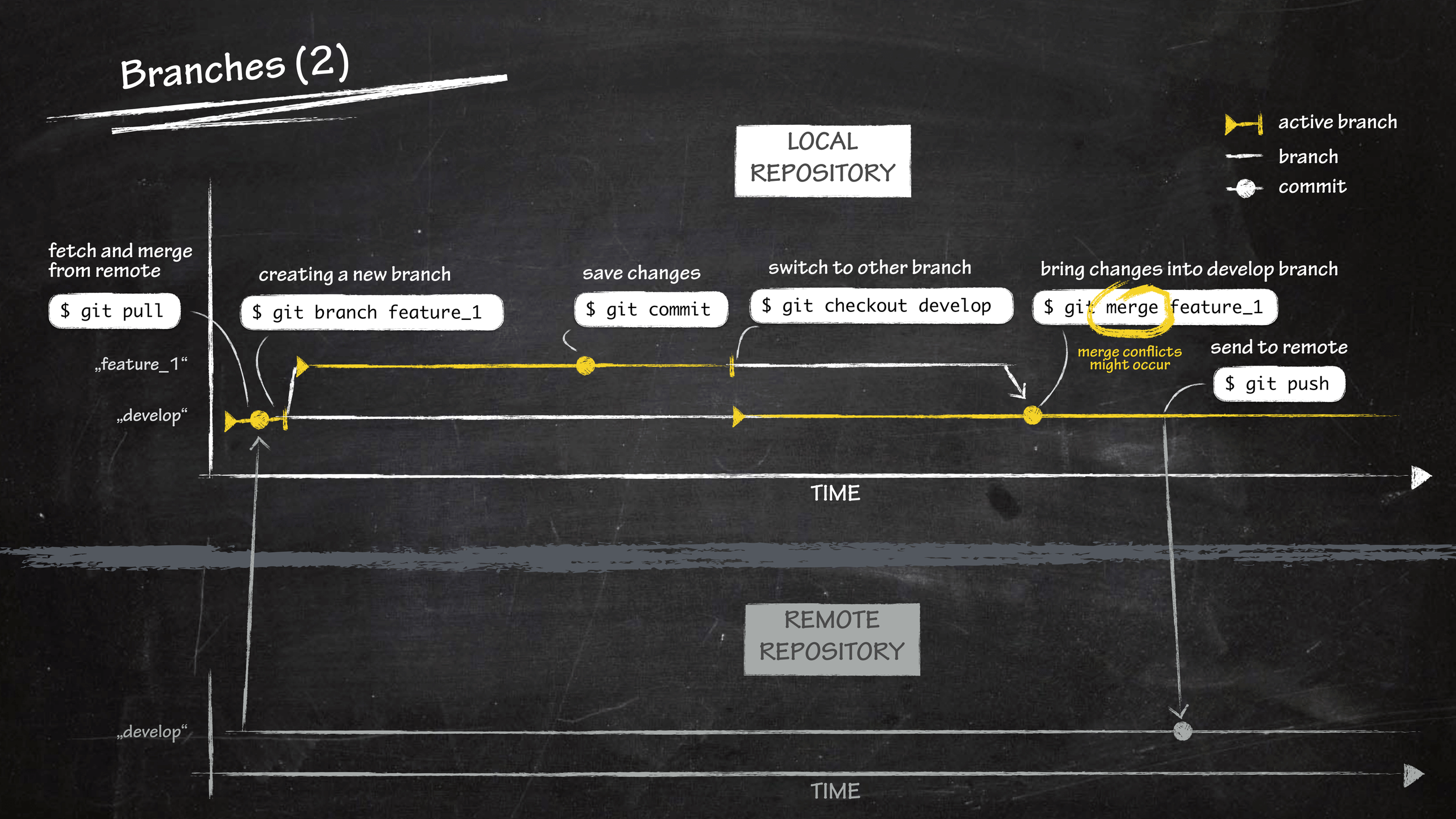
- A remote Git repository is a local Git repository on a remote machine
  - Getting a local copy of a remote repository: `$ git clone`
- Synchronization with remote repositories
  - Transferring (own) changes in a local repository to remote: `$ git push`
  - Bringing local repository up-to-date with changes on remote: `$ git pull`

Branches



- A branch is a series of changes in the working tree having a unique name (and purpose)
- Enables multiple versions of (maybe conflicting) changes concurrently and parallel
- HEAD points to the current (active) branch or commit
- Allows different policy on how bug fixing, feature implementation, hot fixes, releases etc. are synchronized although executed concurrently and parallel
  - Standard branch is called „master“
  - A (remote) repository can contain multiple branches
  - A local repository can cover a subset of a remote repositories branch set
  - Multiple branches can exist but exactly one is active at a time („switched to“)
    - More in detail: HEAD can point only to one commit at a time

Branches (2)


active branch
branch
commit
LOCAL REPOSITORY
fetch and merge from remote
$ git pull
creating a new branch
$ git branch feature_1
save changes
$ git commit
switch to other branch
$ git checkout develop
bring changes into develop branch
$ git merge feature_1
merge conflicts might occur
send to remote
$ git push
„feature_1“
„develop“
TIME
REMOTE REPOSITORY
„develop“
TIME

Diffs

`$ git diff` shows line-by-line changes (similar to a patch) between

- working tree and index or tree,
- index and a tree,
- two trees,
- two (binary or textual) files

before

```
int fibonacci(int n) {
   if (n == 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return (fibonacci(n-1) -
              fibonacci(n-2));
}
```

after

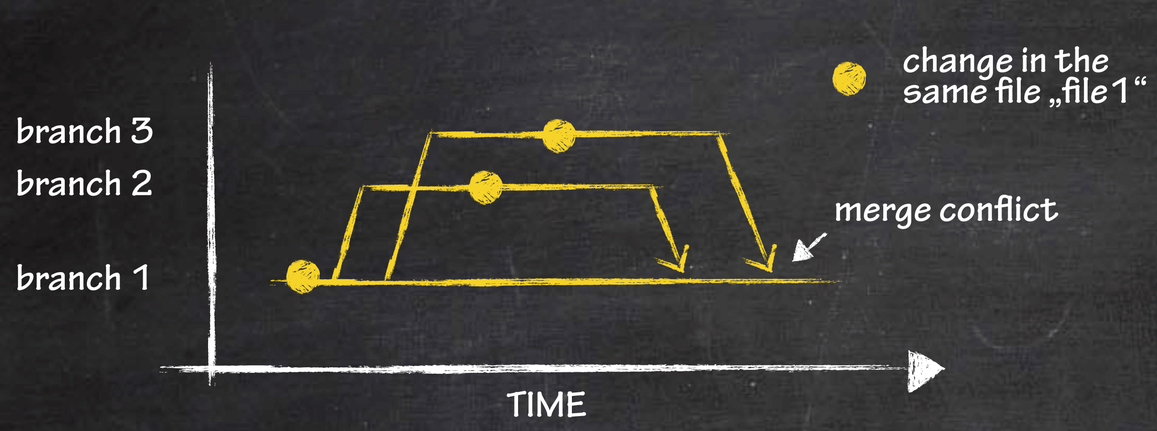
```
unsigned fibonacci(unsigned n) {
   if (n == 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return (fibonacci(n-1) +
              fibonacci(n-2));
}
```

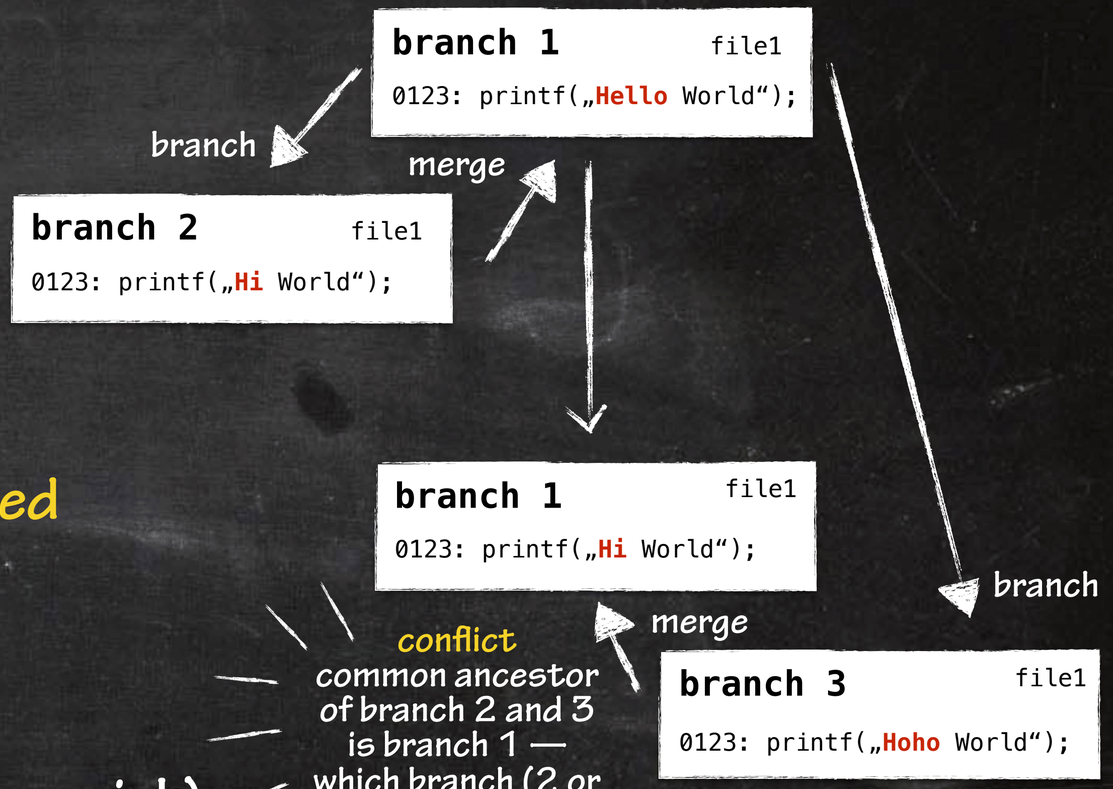
diff (before, after)

```
- int fibonacci(int n) {
+ unsigned fibonacci(unsigned n) {
     if (n == 0)
        return 0;
     else if (n == 1)
        return 1;
     else
-       return (fibonacci(n-1) -
+       return (fibonacci(n-1) +
                fibonacci(n-2));
  }
```

Merge Conflicts

Git merges changes to same file that was edited by more than one dev autonomous.
Sometimes, changes can not be merged automatically due to conflicting changes.





Merge process is stopped and manual resolving is required

- Git shows both changes to the files
  - Option „take mine“ (discard other changes),
  - Option „take theirs“ (discard my changes),
  - Somewhere between (manual editing of file to cherrypick)

When conflicts are resolved, merge process can continue

Fast Forward vs. Non Fast Forward

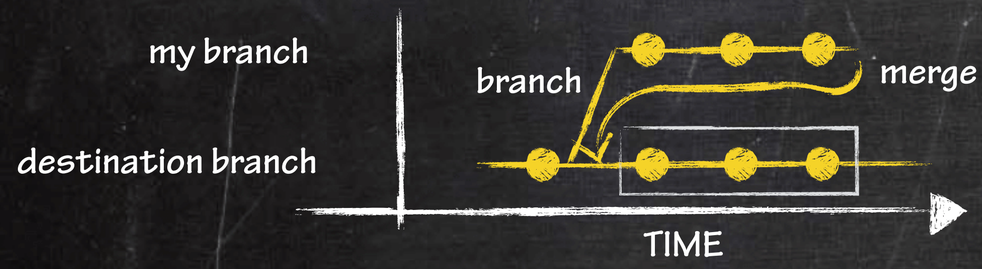
For shortening things, one can create aliases to remote repository URLs. A famous alias is „origin“ that is a particular (default) remote repository.

`$ git push <origin> <local branch>:<remote branch>` updates the branch *<remote branch>* of a remote repository *<origin>* with changes made locally in branch *<local branch>*.

- using `$ git push` w/o arguments moves changes of the current branch to a branch with the same name on a pre-defined remote repository with alias *origin*

To push a branch, merging `$ git merge` branches is a typical operation.

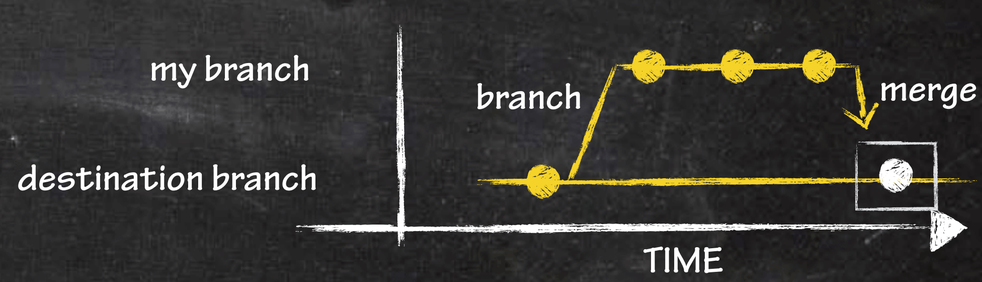
Merging Modes



fast forward (default)

linear history: destination branch history will be linear and no refs to „my branch“ anymore.

PRO easier tooling (log, blame, bisect)



non-fast forward

explicit branches: destination history has commit for merge w.r.t. „my branch“ changes

PRO better tracking

Pull Requests

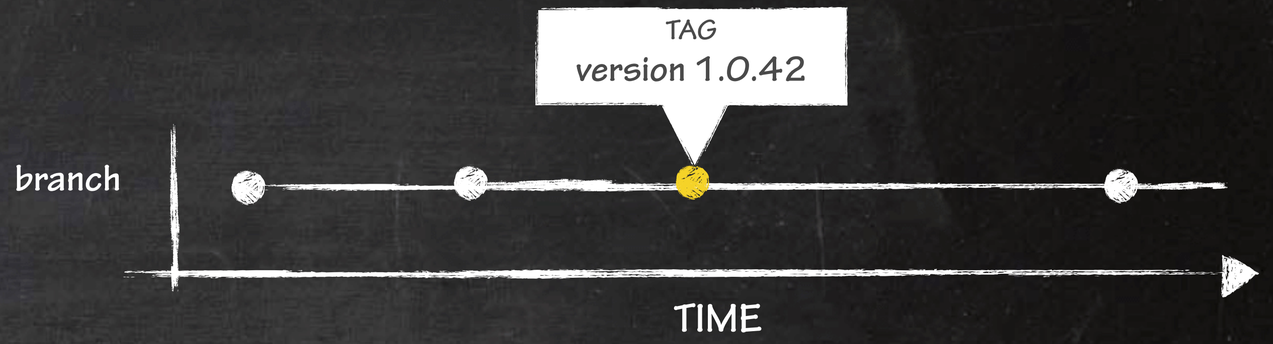
Notification to others that changes are ready & published as remote branch

- Request to remote maintainers to accept and merge („pull“) these changes into main code base (e.g., master)
- Enables discussion, review and follow-up commits (e.g., conflict resolving or synchronization) before changes are merged into main code base
- Anyone with write-privileges to main code base performs the merge
  - often it’s a small team or a single person that accepts or rejects changes
- Pull requests require remote branches (i.e., no centralized VCS workflow)

- PRO Enables managed and more safer merge of changes
- PRO Enables discussion on features that are not yet completed
- CON Often centralizes acceptance/rejection to handful of people

Tagging

Annotating special points in history (e.g., releases) `$ git tag`



Git Ignore

Marking file (patterns) in „.gitignore“ files to never be tracked

- Never commit files that are „intermediate“ or do not belong to the code base into repository
  - Object files
  - Libraries
  - Executables
  - Log files
  - ...

.gitignore (for C)

```
# Prerequisites
*.d

# Object files
*.o
*.ko
*.obj
*.elf

# Linker output
*.ilk
*.map
*.exp

# Precompiled Headers
*.gch
```

...

A list of useful language-specific git ignore files can be found here

https://github.com/github/gitignore

Most Useful Git Commands At a Glance

based on: https://git-scm.com/

`$ git help` Display help information about Git

`$ git help command` Display help about *command*

`$ git init` Create an empty Git repository

`$ git add file-pattern` Add file contents to the index

`$ git rm file-pattern` Remove files from the working tree and from the index

`$ git status file` Show the working tree status

`$ git checkout branch-name` Switch branches or restore working tree files

`$ git checkout commit` Switch to commit by detaching HEAD

`$ git branch branch-name` List, create, or delete branches

`$ git push remote branch` Updates remote repository by sending a branch

`$ git commit` Record changes to the repository

`$ git show object` Show various types of objects

`$ git diff` Show changes between commits, commit and working tree, etc

`$ git log` Show commit logs

`$ git blame` Show what revision and author last modified each line of a file

`$ git merge` Join two or more development histories together

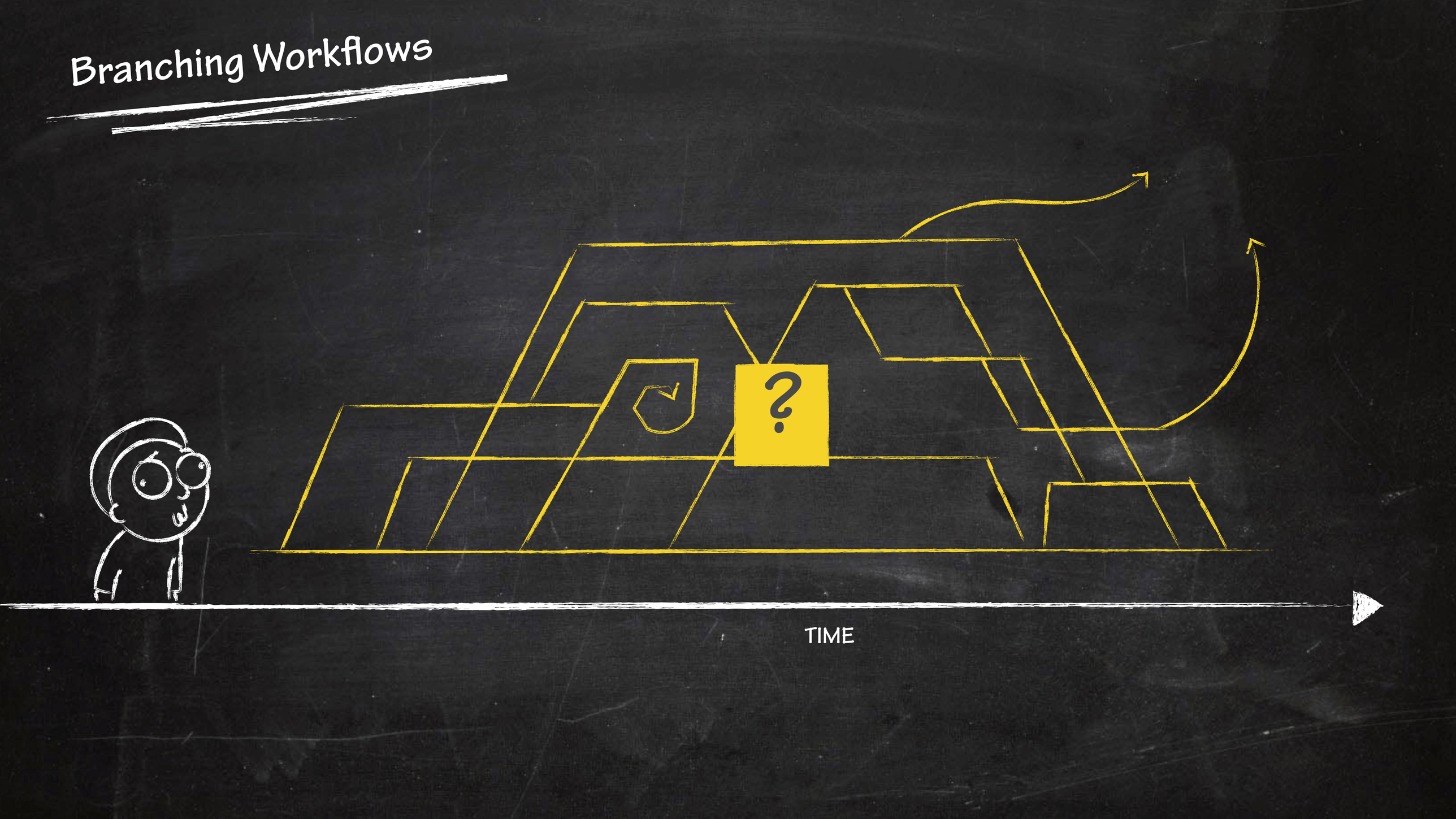
`$ git clone URL` Clone a repository into a new directory

`$ git fetch remote` Download objects and refs (e.g., branches) from another repository

`$ git pull` short for `$ git fetch` followed by `$ git merge`

`$ git rebase` Reapply commits on top of another base tip

Branching Workflows



Branching Workflows (2)

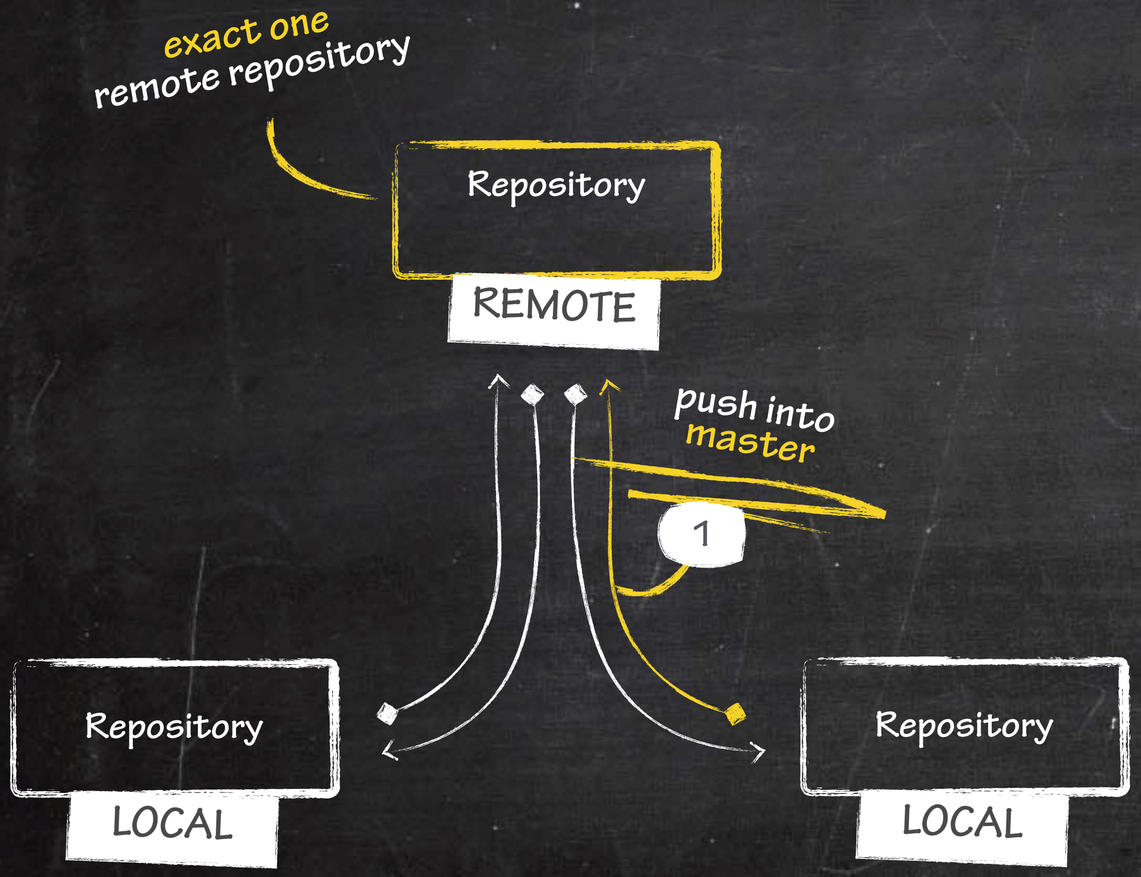
Git enables lightweight creation of and switching to branches both locally and remotely

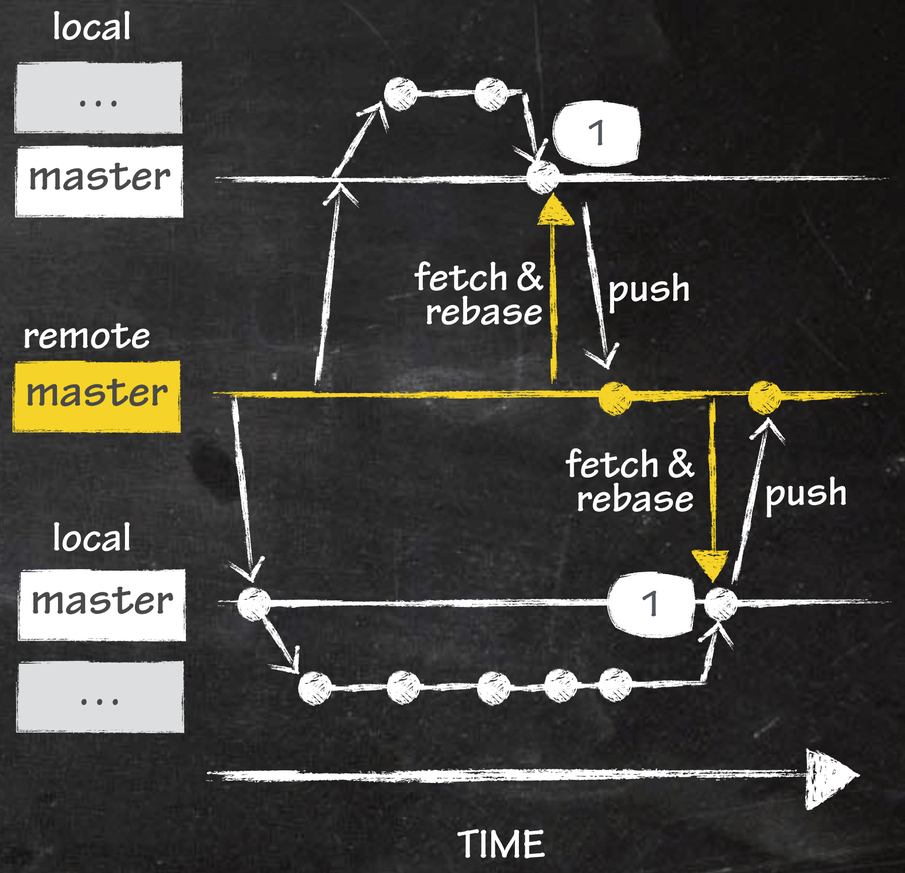
- All branches are treated as equal from Git's point of view
  - How to utilize the branching feature is not in responsibility of the VCS but on the user level (by design)
  - How the branching feature is used is called „branching workflow"
    - depends on
      - purpose,
      - development cycle,
      - company constraints,
      - …
  - Huge design space of workflows; common workflows:

Centralized Workflow	Feature Branch Workflow	GitFlow Workflow	Forking Workflow

Branching Workflows (3)

Centralized Workflow





Branching Workflows (4)

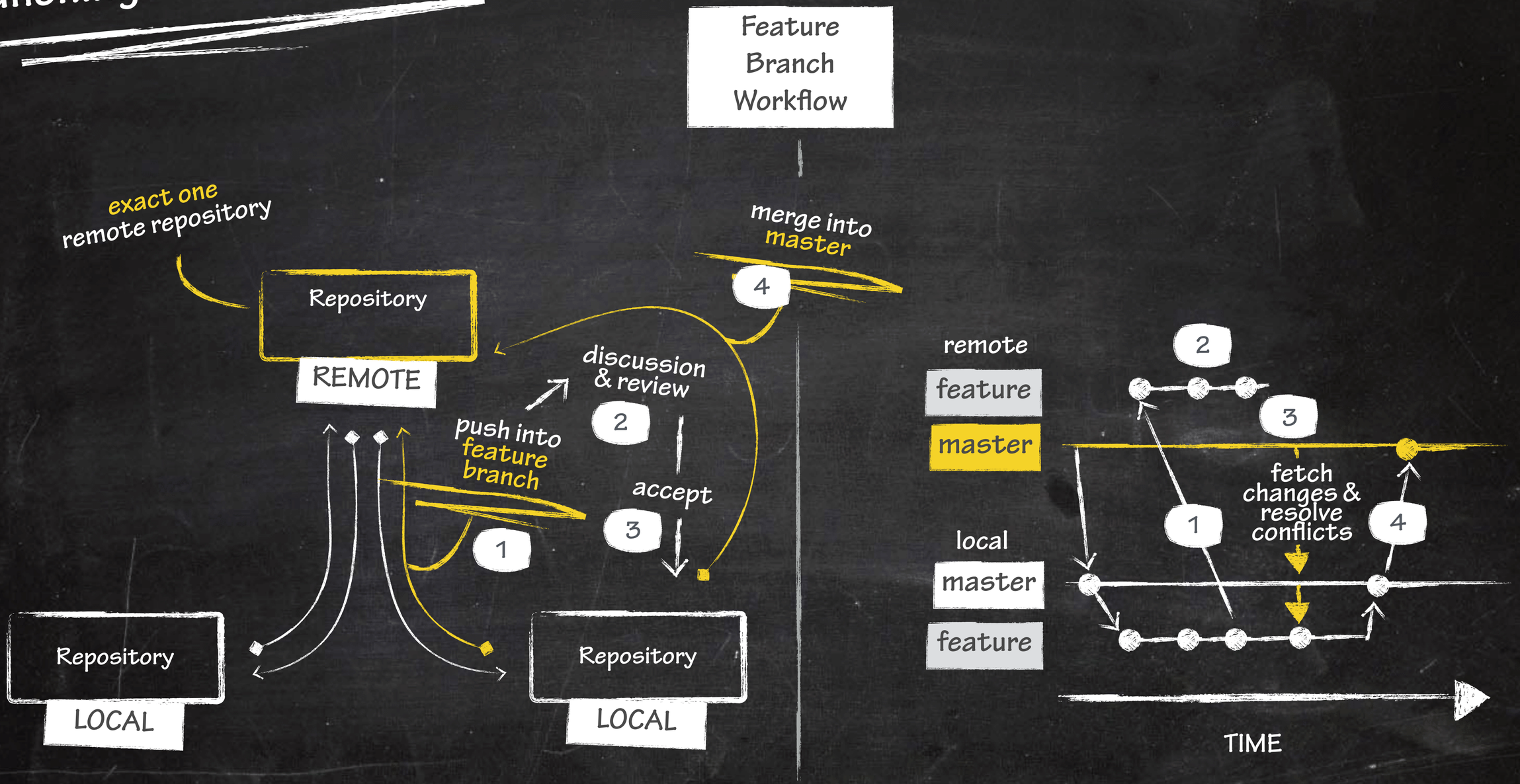
Centralized Workflow

remote
master

rebasing enables pushing local changes always at „the end of the time line“ of the remote master.

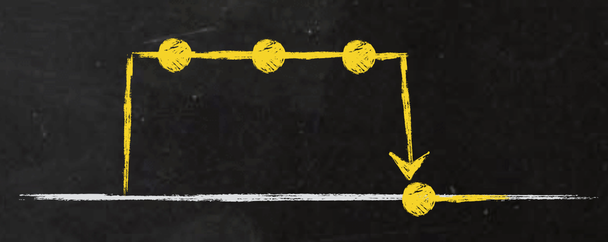
- classical workflow similar to a centralized VCS workflow, e.g., Subversion (SVN)
  - PRO Easy to understand & to use
  - PRO Single latest product version in remote
  - CON Merge conflicts before pushing into remote master
  - CON No distribution and remote master „single point of failure“
  - CON Potential buggy remote master due to broken code
- single centralized remote repository containing „the only one truth“
  - default branch is master in which all changes are committed
  - developers clone centralized repository, make changes locally
    - fetch changes from remote master before pushing
    - Resolve merge conflicts locally
    - rebase own changes on top of latest changes in master
    - push their changes into master of remote repository

Branching Workflows (5)



Branching Workflows (6)

Feature Branch Workflow



- Core idea: each feature development in a dedicated branch rather than in master
  - PRO Parallel development of features
  - PRO Less likely having buggy remote master
  - PRO Pull requests to discuss features
  - CON Longer reviewing process and risk of working on out-dated code
  - CON Long-running feature branches, less feedback & less communication
  - CON Having a feature branch with refactoring may break other feature branches
- single centralized remote repository containing „the only one truth“
  - default branch is master in which all feature branches are merged
  - developers clone centralized repository, make changes locally
    - push their changes as a dedicated feature branch to remote
    - code review & discussion (e.g., by opening a pull request), resolve conflicts
    - when accepted, fetch changes from remote master before pushing
    - push changes into master of remote repository

Branching Workflows (7)

GitFlow Workflow

Extension of feature brach workflow with roles and interactions of branches.

Core idea: Remote repository contains purpose-driven branches with infinite, and finite lifetimes & roles to branches and their interaction.

Permanent Branches - infinite lifetime

- Master is latest stable release shipped (planned or as reaction to hotfix request)
- Develop is current development state with new features for next release

Non-Permanent Branches - finite lifetime

- Feature branches to add new, or modify existing functionality or behavior
    - never interact with master; are merged into develop branch
- Release reflects feature-complete develop state that is currently getting bug-fixed
    - branches from develop branch & indicates release cycle (i.e., feature-stop)
    - When stable, release branch is tagged and merged into master and develop
- Issue branches
    - „hotfix“ resp. „bugfix" used to repair something broken in master resp. develop

Based on: http://www.app-entwicklung.info/2014/12/das-branching-modell-git-flow/#

Branching Workflows (8)

GitFlow Workflow

permanent branch

intermediate branch

feature_a

feature_b

feature_c

bugfix_a

bugfix_b

bugfix_c

...

<<prepare release>>

<<prepare release>>

hotfix_a

version 1.0.0

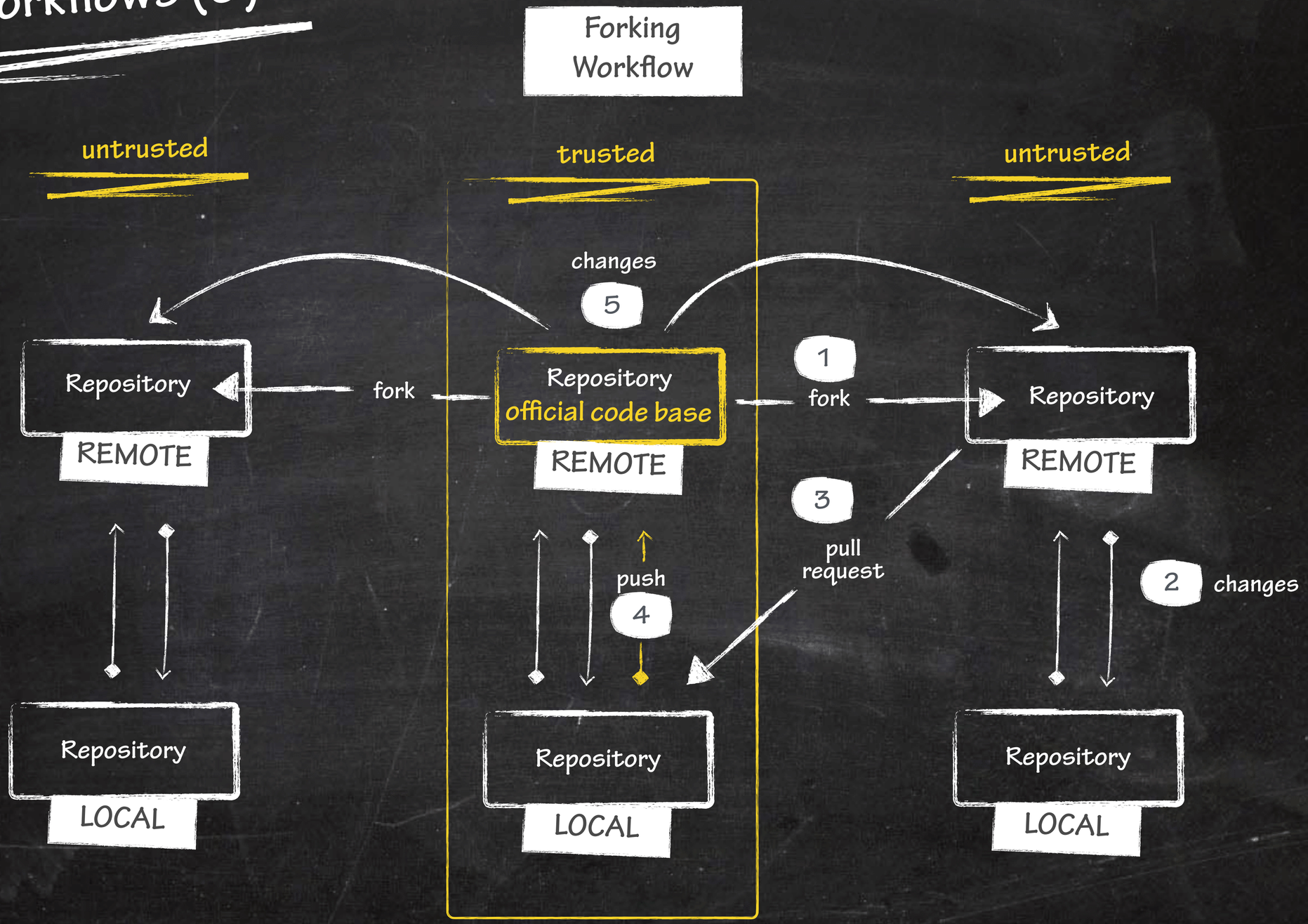
version 1.1.0

version 1.1.1

version 1.2.0

TIME

Branching Workflows (9)



Branching Workflows (10)

Forking Workflow

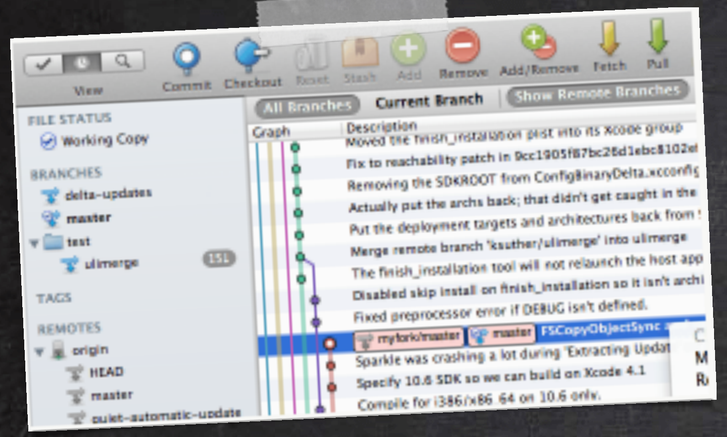
Core idea: Each developer has own remote repository & trusted project maintainers of official code base act as „gate keepers“ for untrusted contributors

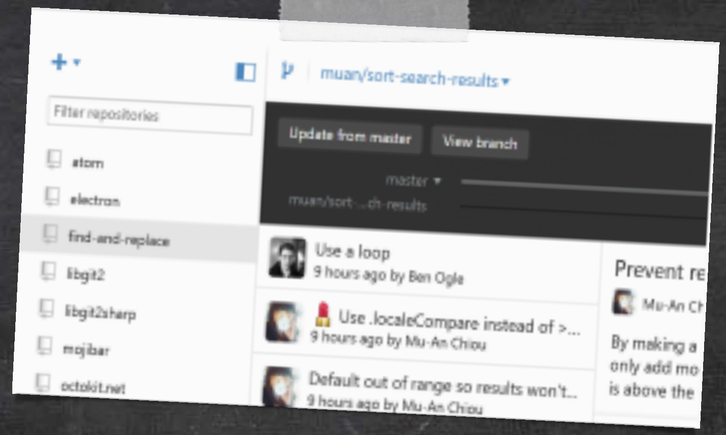
- PRO Flexible workflow to securely collaborate with changing untrusted dev teams
- PRO Maintainer must not provide write access to project repository
- PRO Good fit for many open source projects since official code base is trusted
- CON For companies, employees are generally trusted. Hence, often no need for forking

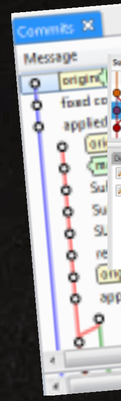
- single remote read-only repository („remote original“) is „the only one *trusted* truth“
  - developers copy remote original to their own servers („remote fork“)
  - developers clone from their remote fork, make changes locally in local fork
    - push their changes to their remote fork
    - open a pull request in remote original repository
  - maintainer of remote original pulls changes from remote fork into local original
    - when accepting, merges changes into local original and push into remote original
    - notify others that remote original has changed
  - Each contributing repository manages its own workflow policy (e.g., GitFlow)

Tooling

Not a fan of the command line?
Maybe GUI clients will help you…







see more: https://git-scm.com/downloads/guis/
or google for it: Git GUI clients