Using GIT

Pr. Olivier Gruber

olivier.gruber@imag.fr

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Overview

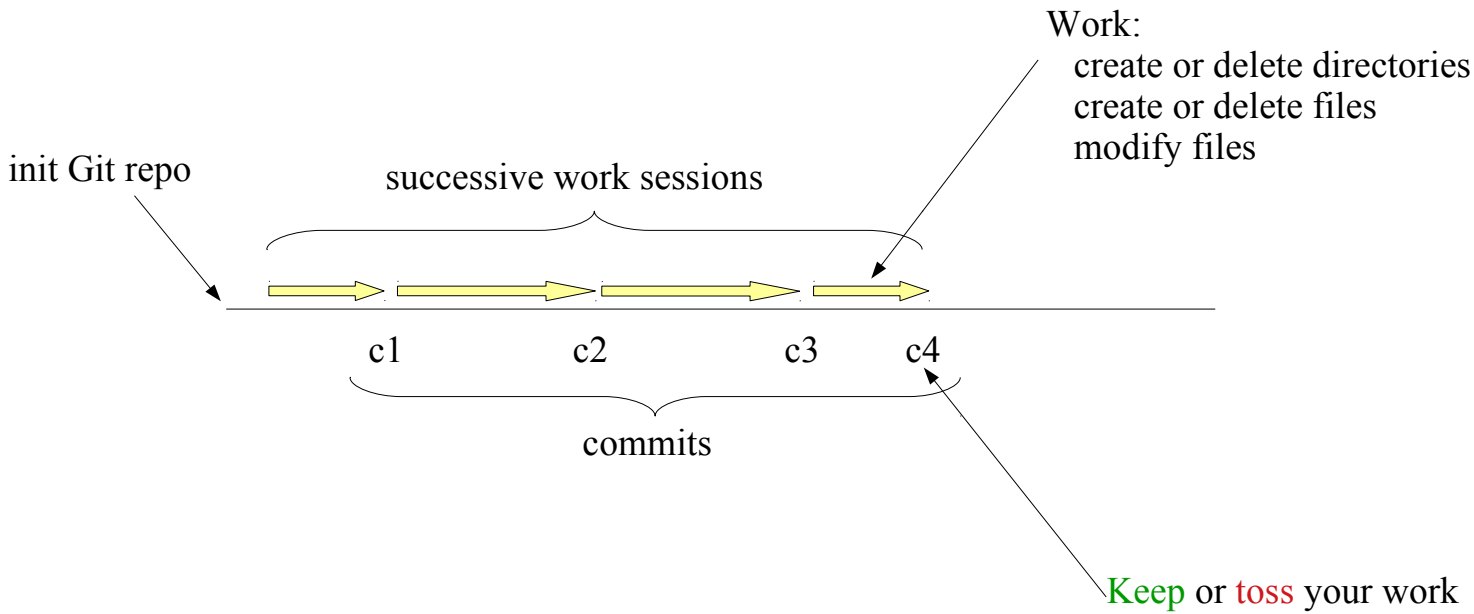
- What is GIT
  - Keeping histories of the evolution of files and directories
  - The ability to merge histories, working alone or in a team
- Personal usage
  - Protect and evolve your code
  - Look at the past
- Team work
  - Dividing work and be able to merge later
  - Flexible cooperation

Real-life Scenarii

- Personal usage examples
  - Last minute, last change…
  - It started as a simple change, really…
  - Hum, what if...
- Team work examples
  - Which USB key as the last version already?
  - Trust me, I tested my changes...
  - What? Fix you a bug now! Can't, in the middle of something…
  - Hey guys, sorry, but my machine died on me...

GIT concepts

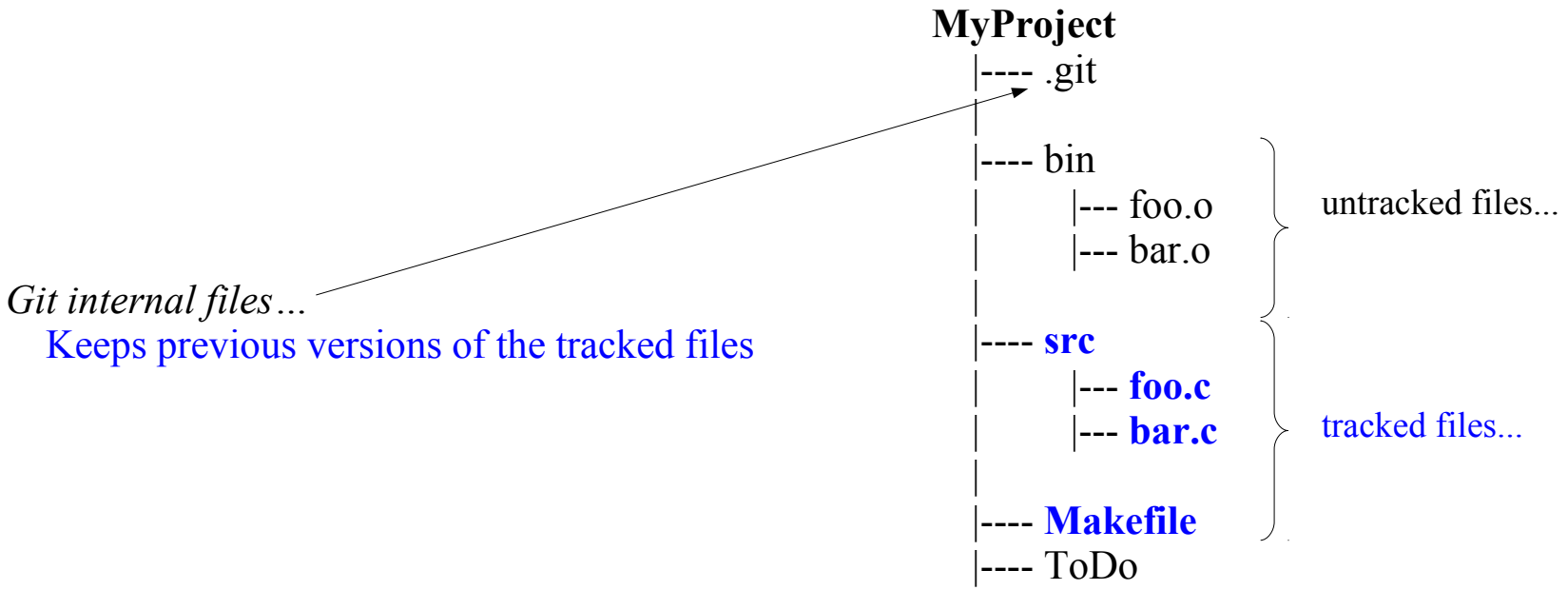
- GIT History – A saga of transactions
  - Tracks changes to a directory (its files and subdirectories)
  - A transaction: do some work, then commit or abort



GIT concepts

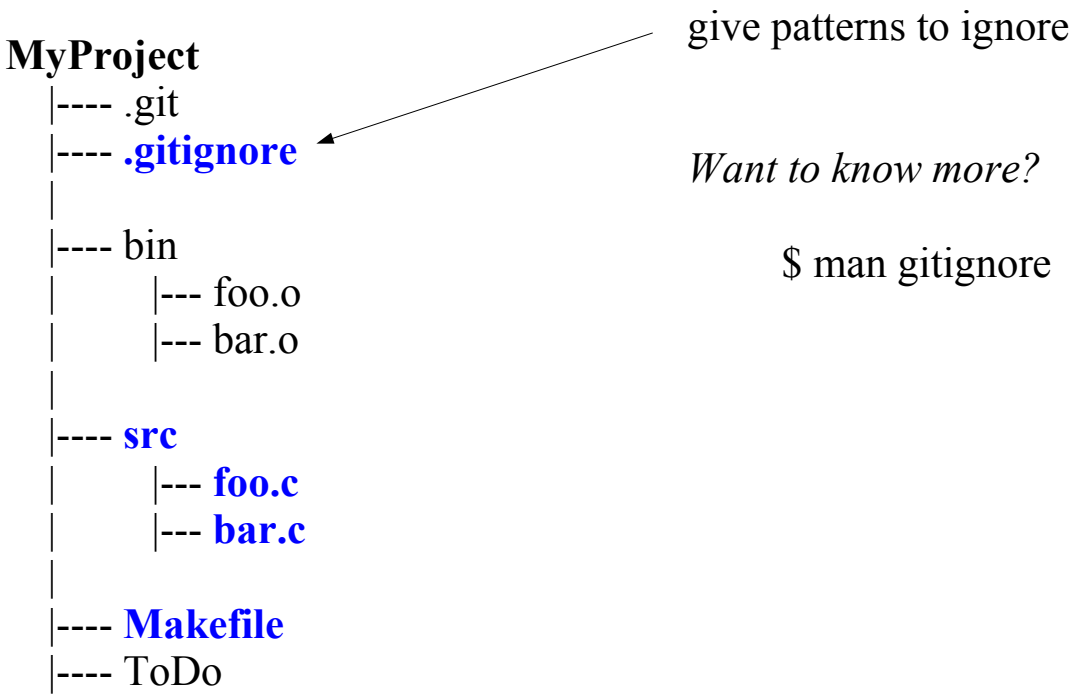
- GIT Repository
  - A repository manages one tree of directories and files
  - Git tracks the history of **certain files and directories**

```
$ mkdir MyProject
$ cd MyProject
$ git init
Initialized empty Git repository in ../MyProject/.git/
$
```



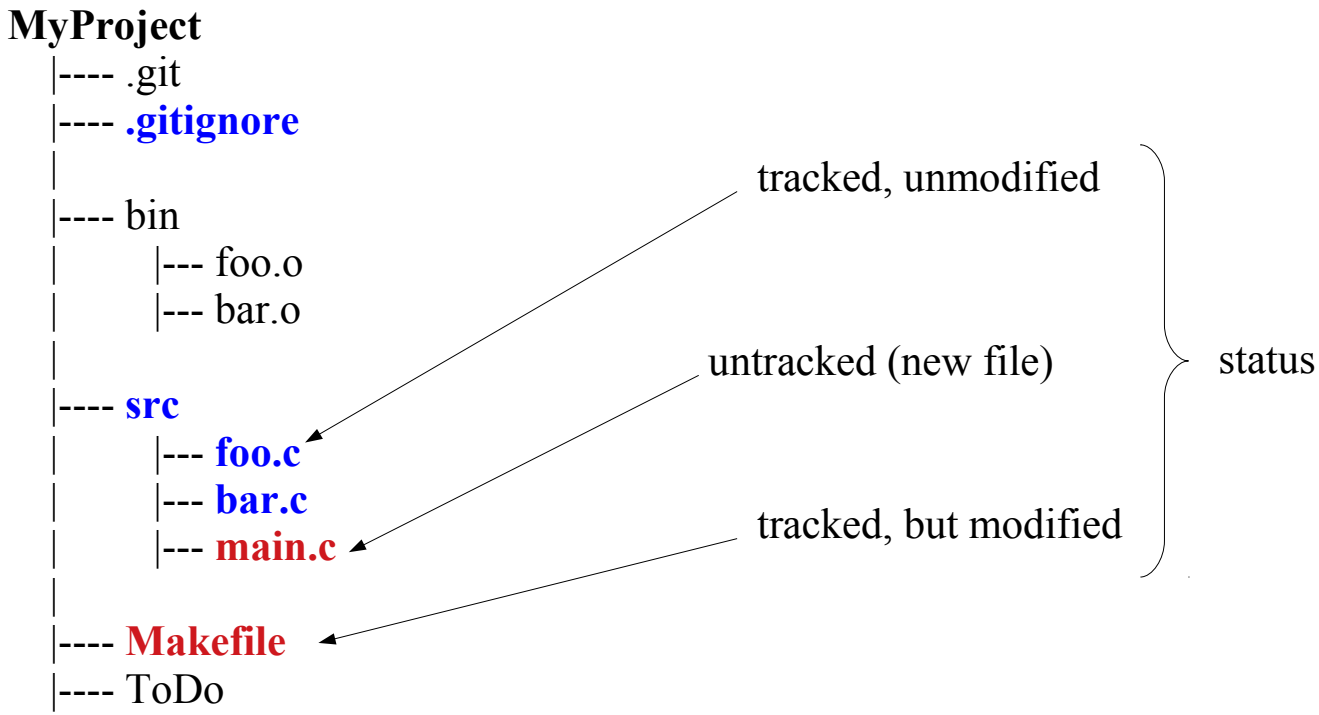
GIT concepts

- GIT Repository
  - A repository manages one tree of directories and files
  - Git tracks the history of **certain files and directories**



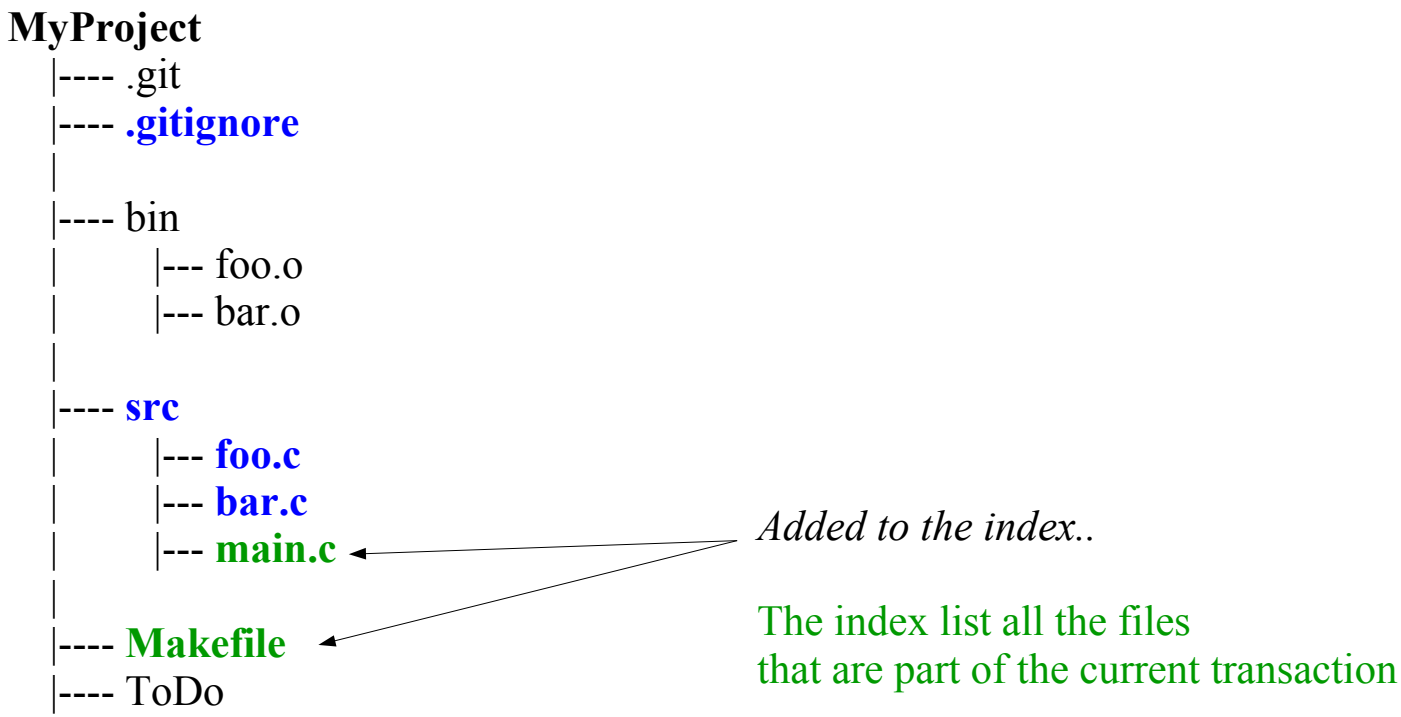
GIT concepts

- GIT Status
  - Git tracks the history of certain files and directories
  - **Each file has a status** – untracked, tracked, modified



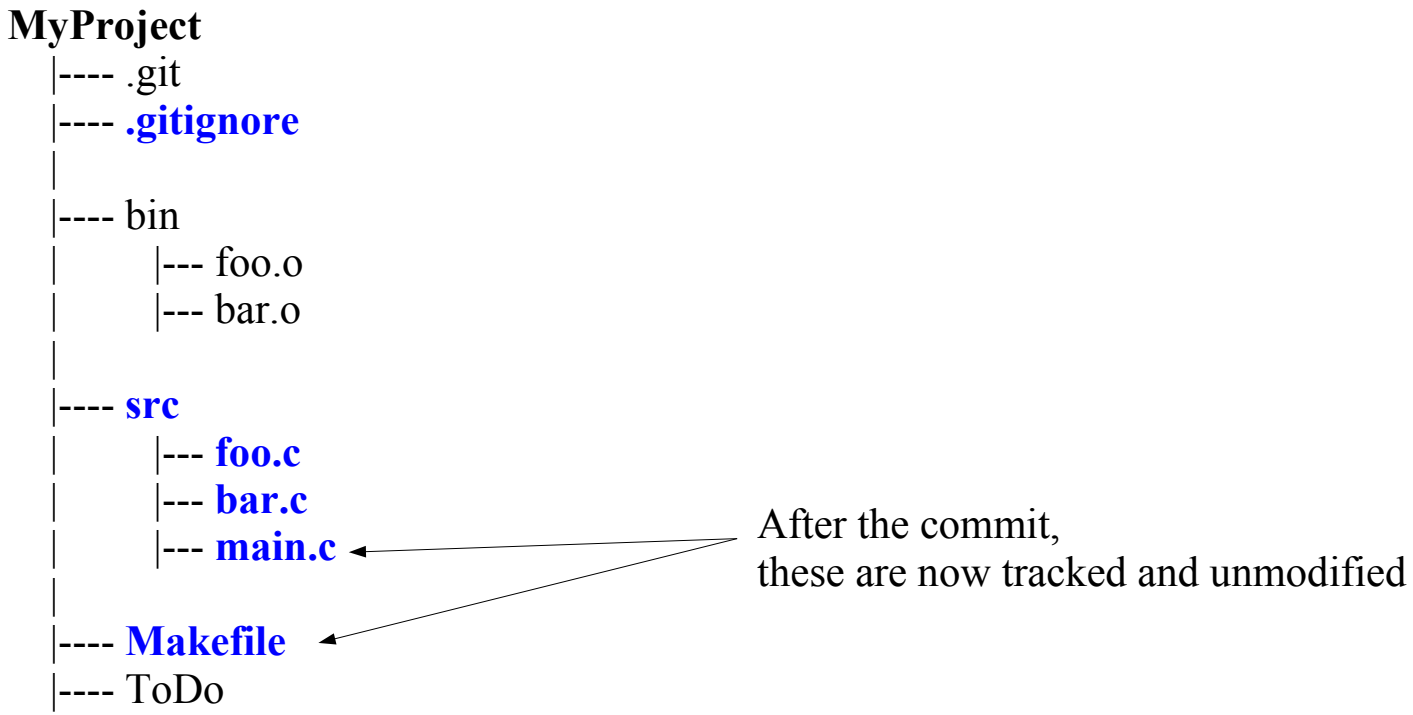
GIT concepts

- GIT Status
  - Git tracks the history of certain files and directories
  - **Each file has a status** – untracked, tracked, modified, **indexed**



GIT concepts

- GIT Status
  - Git tracks the history of certain files and directories
  - Each file has a status – untracked, tracked, modified, indexed
  - **Commit**: save changes for all the files in the index



GIT concepts – Recap

- Four commands
  - **Git-init**
    - Create a repository
  - **Git-status**
    - Give you the status of files
  - **Git-add**
    - Add files to the index
    - Modified or untracked files
  - **Git-commit**
    - Commit changes to indexed files

Want to know more?

$ man git-init
$ man git-status
$ man git-add
$ man git-commit

GIT concepts – Recap

- Four commands
  - **Git-init**
    - Create a repository
  - **Git-status**
    - Give you the status of files
  - **Git-add**
    - Add files to the index
    - Modified or untracked files
  - **Git-commit**
    - Commit changes to indexed files

```
$ mkdir MyProject
$ cd MyProject
$ git init
Initialized empty Git repository in ../MyProject/.git/
$
```

GIT concepts – Recap

- Four commands
  - **Git-init**
    - Create a repository
  - **Git-status**
    - Give you the status of files
  - **Git-add**
    - Add files to the index
    - Modified or untracked files
  - **Git-commit**
    - Commit changes to indexed files

$ ...
Add main.c
Modify Makefile

$ **git status**
Modified: Makefile
Untracked: main.c

GIT concepts – Recap

- Four commands
  - **Git-init**
    - Create a repository
  - **Git-status**
    - Give you the status of files
  - **Git-add**
    - Add files to the index
    - Modified or untracked files
  - **Git-commit**
    - Commit changes to indexed files

$ ...
Add main.c
Modify Makefile

$ **git status**
Modified: Makefile
Untracked: main.c

$ **git add –all**
$ **git status**
Makefile
main.c

GIT concepts – Recap

- Four commands
  - **Git-init**
    - Create a repository
  - **Git-status**
    - Give you the status of files
  - **Git-add**
    - Add files to the index
    - Modified or untracked files
  - **Git-commit**
    - Commit changes to indexed files

$ ...
Add main.c
Modify Makefile

$ **git status**
Modified: Makefile
Untracked: main.c

$ **git add –all**
$ **git status**
Makefile
main.c

$ **git commit** -m “Added main.c”
$ git status
nothing to commit, working directory clean
$

GIT concepts – Recap

File System View

```
MyProject
    |---- .git
    |---- .gitignore
    |
    |---- bin
    |         |--- foo.o
    |         |--- bar.o
    |
    |---- src
    |         |--- foo.c
    |         |--- bar.c
    |         |--- main.c
    |
    |---- Makefile
    |---- ToDo
```

Developer View

MyProject
- bin
  - foo.o
  - bar.o
- **src**
  - **foo.c**
  - **bar.c**
  - **main.c**
- **Makefile**
- ToDo

Index View

Modified: **Makefile**

Modified: **foo.c**

Untracked: **main.c**

Repo View

History of all commits
And the previous versions
of tracked files

Using GIT for Oneself

- For what?
  - Permits trials and errors
  - Provides a safety net
- Nothing is free
  - Git maintains a history, so it requires space on your disk
  - It takes some practice to getting used it

Using GIT – Safety Net

- Local safety net
  - Git maintains a copy of your work in the ".git" directory
  - So if you corrupt or loose a file/directory, you can recover it
  - As long as you do not delete or damage the contents of the ".git" directory
- Remote safety net
  - This is a bit more advanced usage
  - You could clone your repository on a remote machine, used as a backed-up machine
  - Example
    - You could use your account at the UFR
    - Or use github
    - Or use another machine at home

Using GIT – Safety Net

- Git-checkout: revert changes
  - Revert **uncommitted** changes
  - Restore the contents of modified files
  - Restore **removed** or **renamed** files or directories

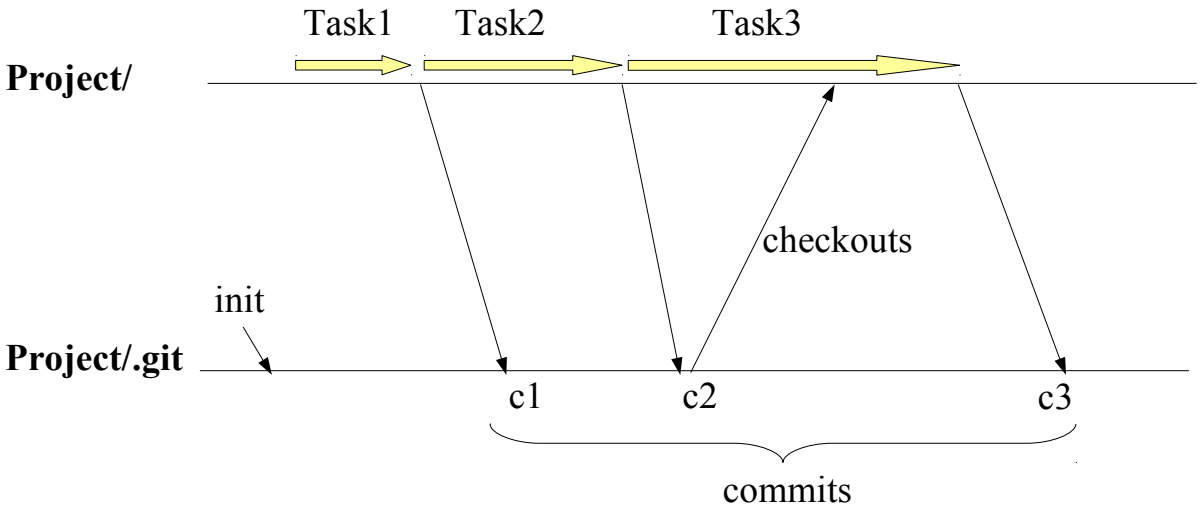
```
$ git add –all
$ git commit -m “Task2 done”

    Working on Task3
    Ouch:
      - f***ed up file Toto.java
      - removed        Titi.java

$ git checkout Toto.java
$ git checkout Titi.java

...Finish Task3

$ git add –all
$ git commit -m “Task3 done”
```



Using GIT – Safety Net

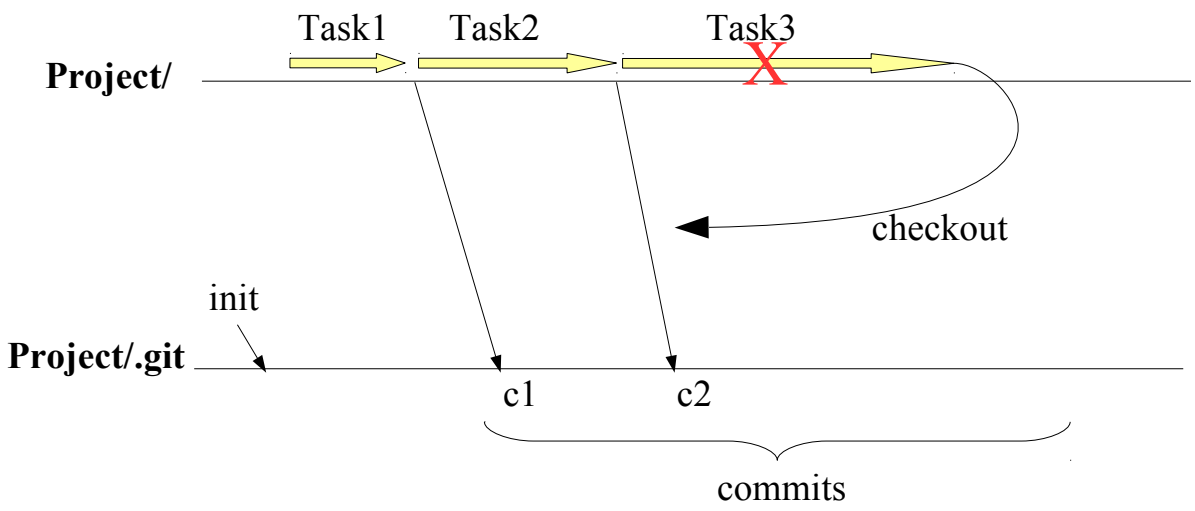
- Git-checkout: revert changes
  - Revert all **uncommitted** changes **only** in **tracked** files
  - Be careful here, you need to be sure before you ask for this… **it cannot be undone**

…Working on Task2

$ **git add** –all
$ **git commit** -m “Task2 done”

…Working on Task3
…@#^*#^$&@&#^%*&*
…OK, let‘s toss all that non sense…

$ **git checkout -f**



Using GIT – Safety Net

- Git-checkout: revert changes, back several commits
  - Revert all **uncommitted** changes in **tracked** files

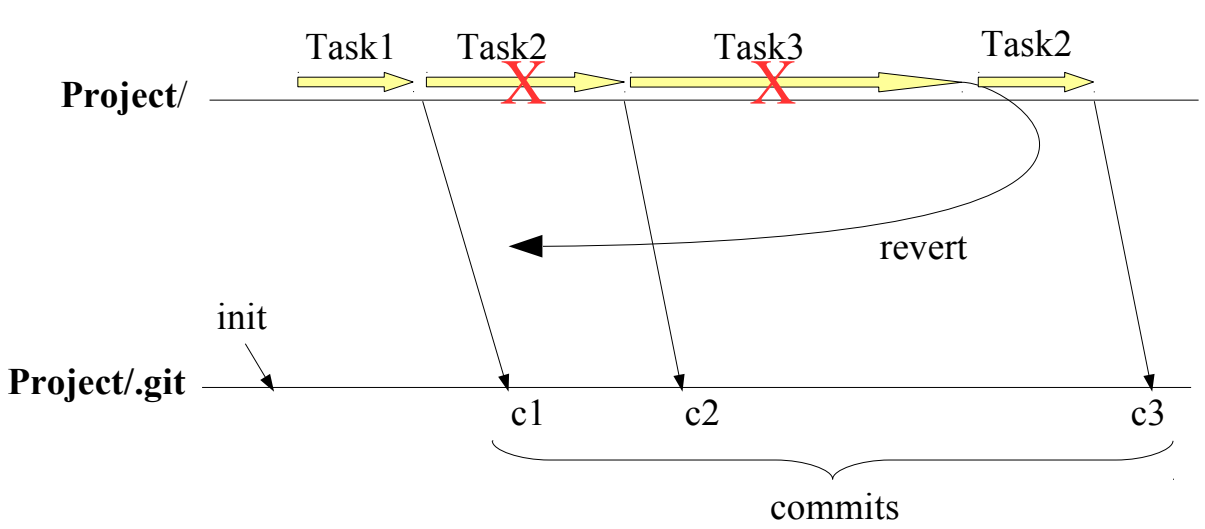
… Oh oh…
Task2 and Task3 were a mistake
Let's scratch that work.

```
$ git log
commit 75c027
  c2
commit 35c025
  c1
```

$ **git revert** --no-commit **35c025**..HEAD

...now do it right

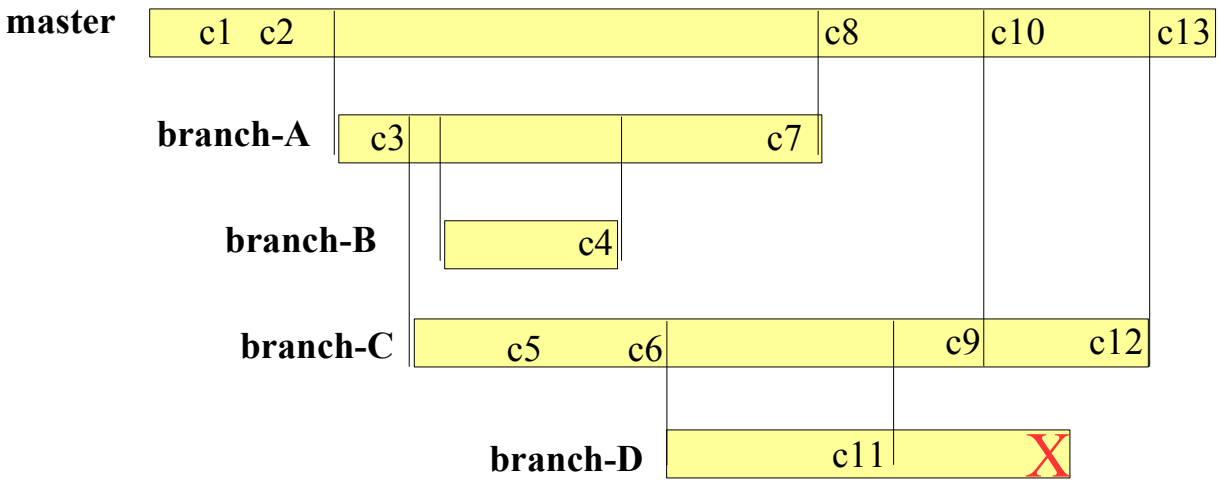
$ **git add** --all
$ **git commit** -m "Ouf… "



You can do it this way, but using branches is **much** easier...

Using GIT Branches

- GIT history
  - Branch: fork and merge your work
  - Yields a tree of commits, on various branches

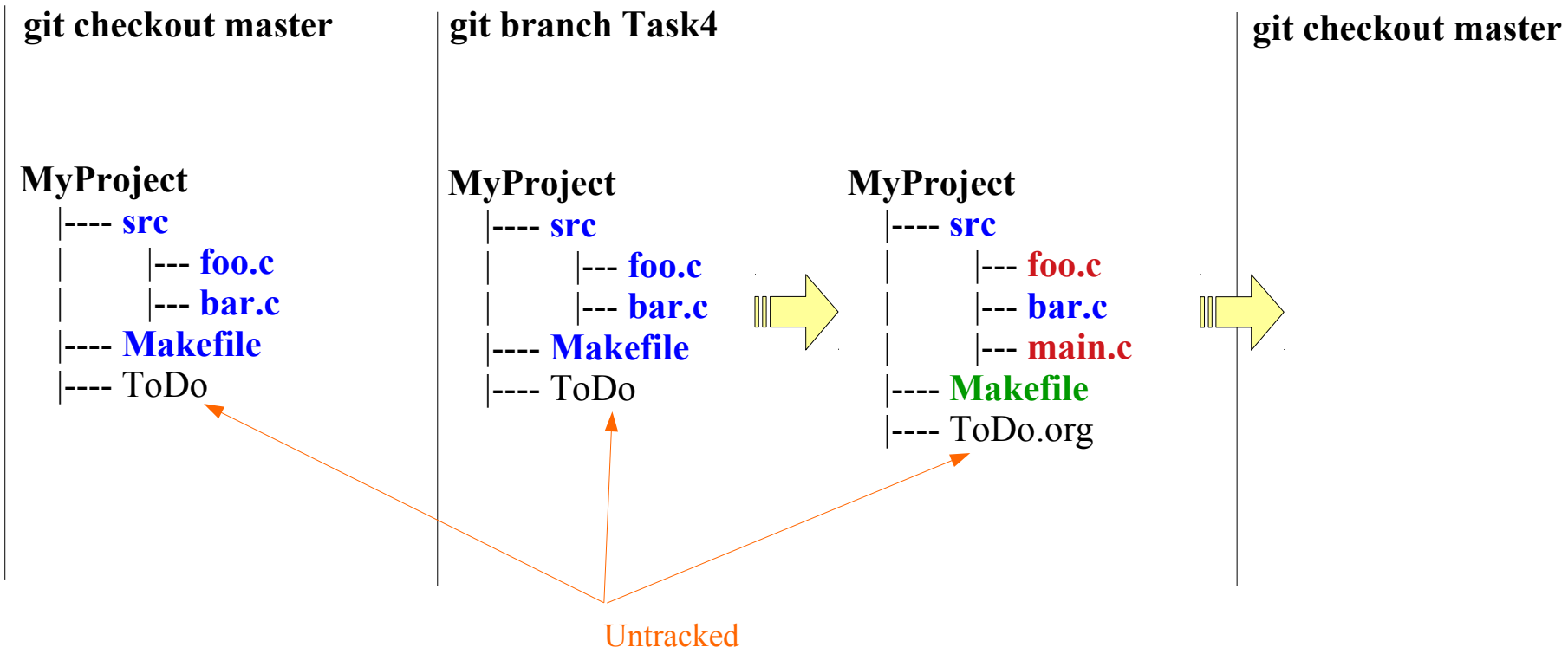


WARNING:
this may become complex…

ADVICE:
keep it simple

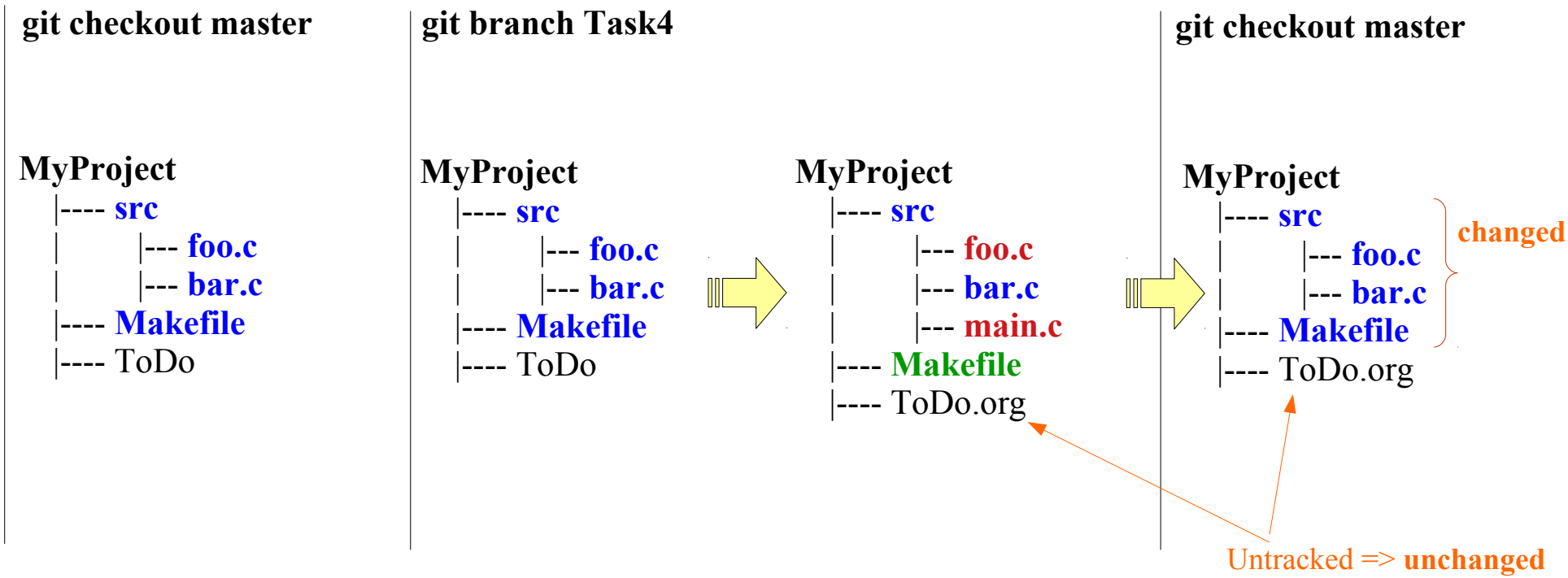
Using GIT branches

- Switching between branches
  - Changes the contents of **tracked** files
  - Leaves untracked files untouched



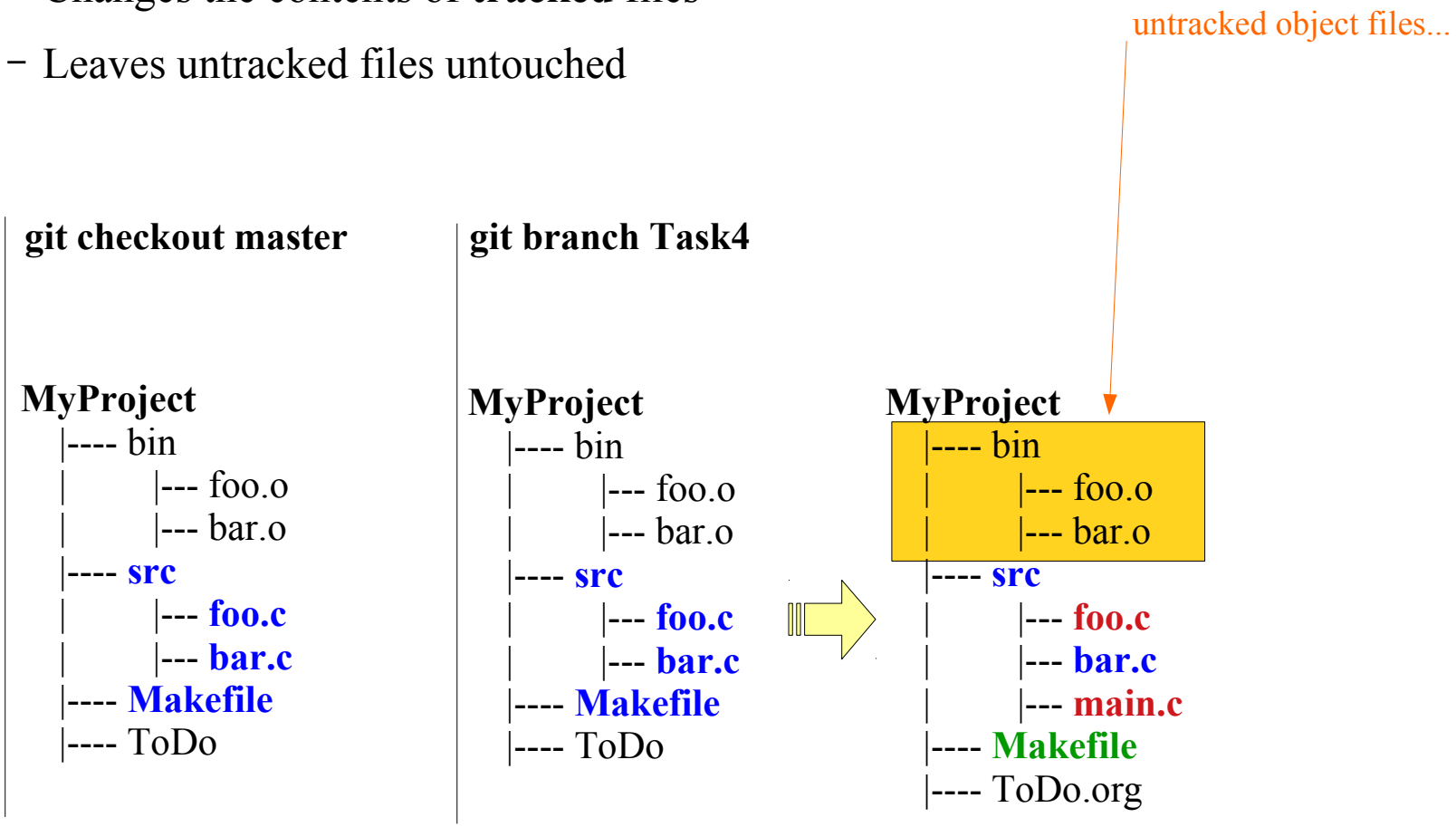
Using GIT branches

- Switching between branches
  - Changes the contents of **tracked** files
  - Leaves untracked files untouched



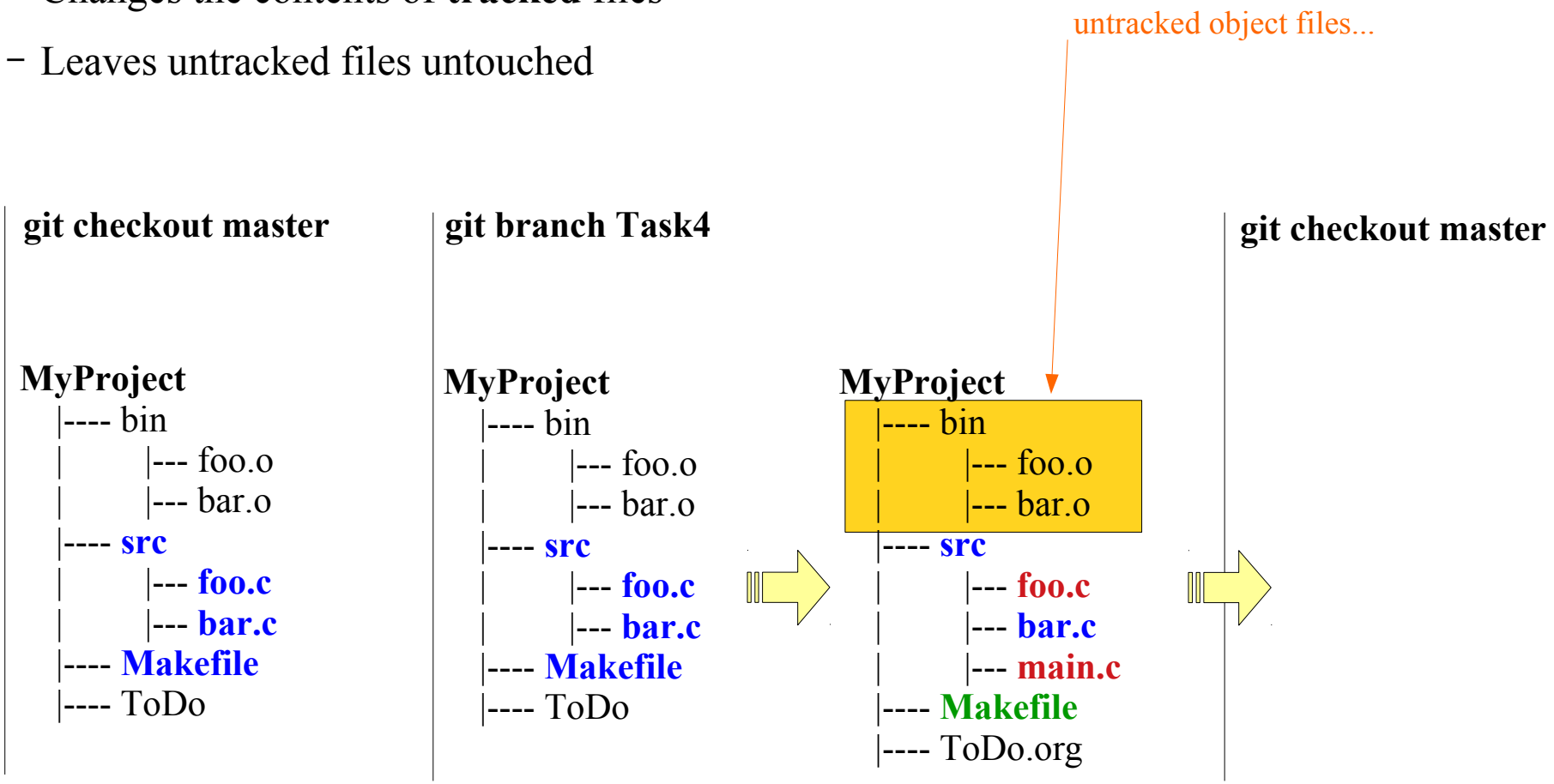
Using GIT branches

- Switching between branches
  - Changes the contents of **tracked** files
  - Leaves untracked files untouched



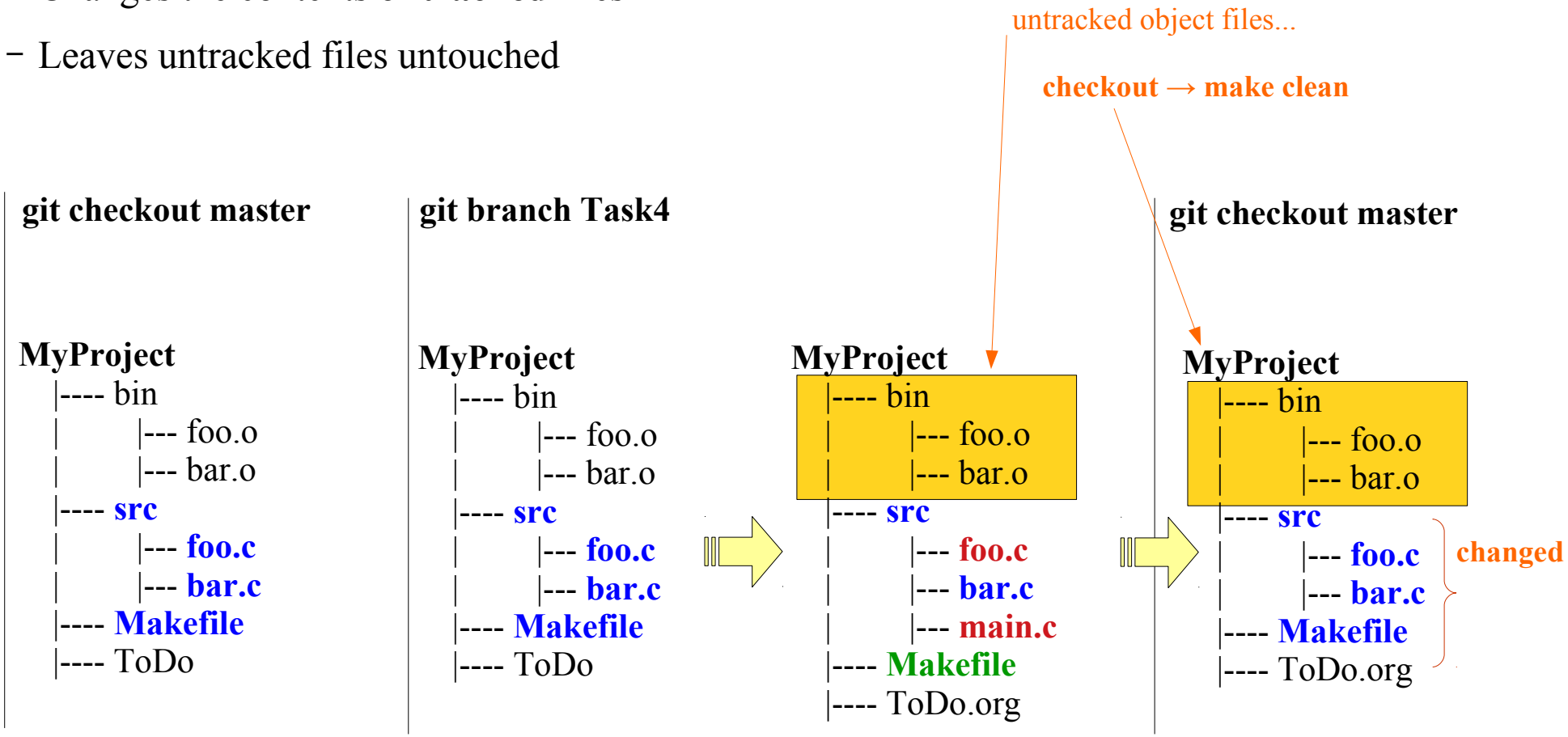
Using GIT branches

- Switching between branches
  - Changes the contents of **tracked** files
  - Leaves untracked files untouched



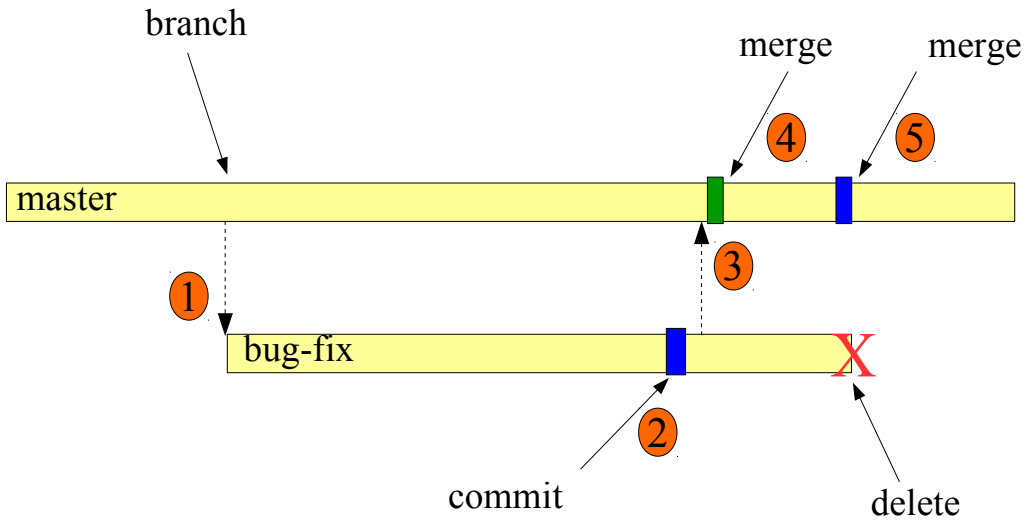
Using GIT branches

- Switching between branches
  - Changes the contents of **tracked** files
  - Leaves untracked files untouched



Using GIT branches

- The simplest pattern
  - Create a branch, merge it
  - Delete the branch if no longer needed



1. $ **git branch bug-fix**
   $ **git checkout bug-fix**

   ... work on fixing the bug

2. $ **git add** –all
   $ **git commit** -m “Bug fixed”

3. $ **git checkout master** ⟸ Come back on the branch *master*

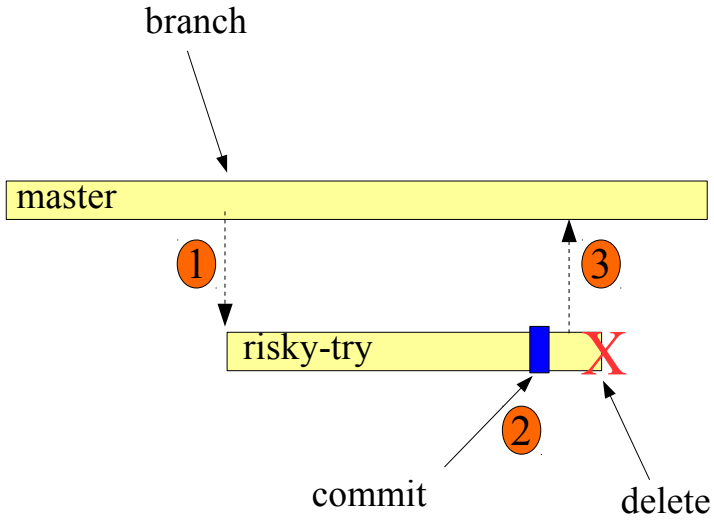
4. $ **git merge bug-fix** ⟸ Merge the branch bug-fix into master
   No conflicts, if master is <u>never</u> worked on

5. $ **git add** –all
   $ **git commit** -m “Bug fixed merged”

   $ **git branch -d bug-fix** ⟸ Optional delete of the branch

Using GIT branches

- The simplest pattern – A great safety net
  - Create a branch, do some work
  - Toss your work at any time...



1. $ **git checkout -b risky-try**

   ... work on trying something risky...

2. $ **git add** –all
   $ **git commit** -m “First commit”

   ... If you do not like it at some point

3. $ **git checkout master** ⟵ Come back on the branch *master*

   $ **git branch -d risky-try** ⟵ Optional delete of the branch

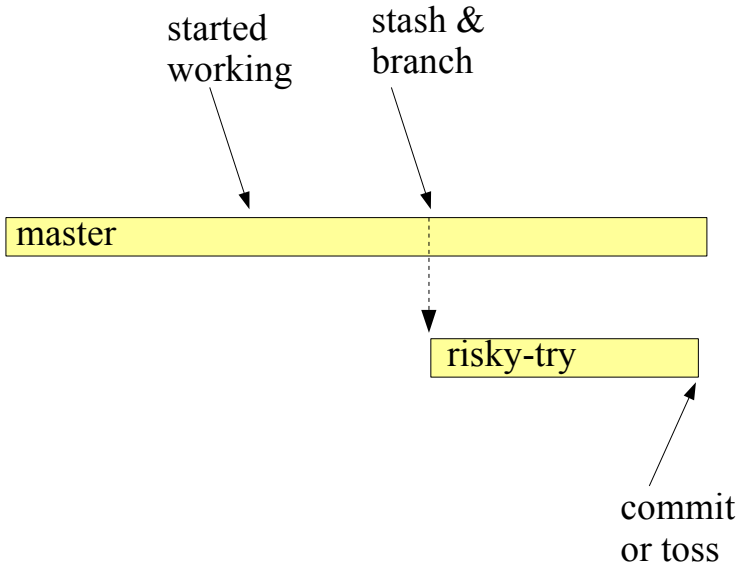
Using GIT branches

- What if your already started to work…
  - Then you realize this is going to be risky stuff…
  - No problem! (but avoid this situation nevertheless)

… started working on something…
… you are growing concerned…
… just too risky…

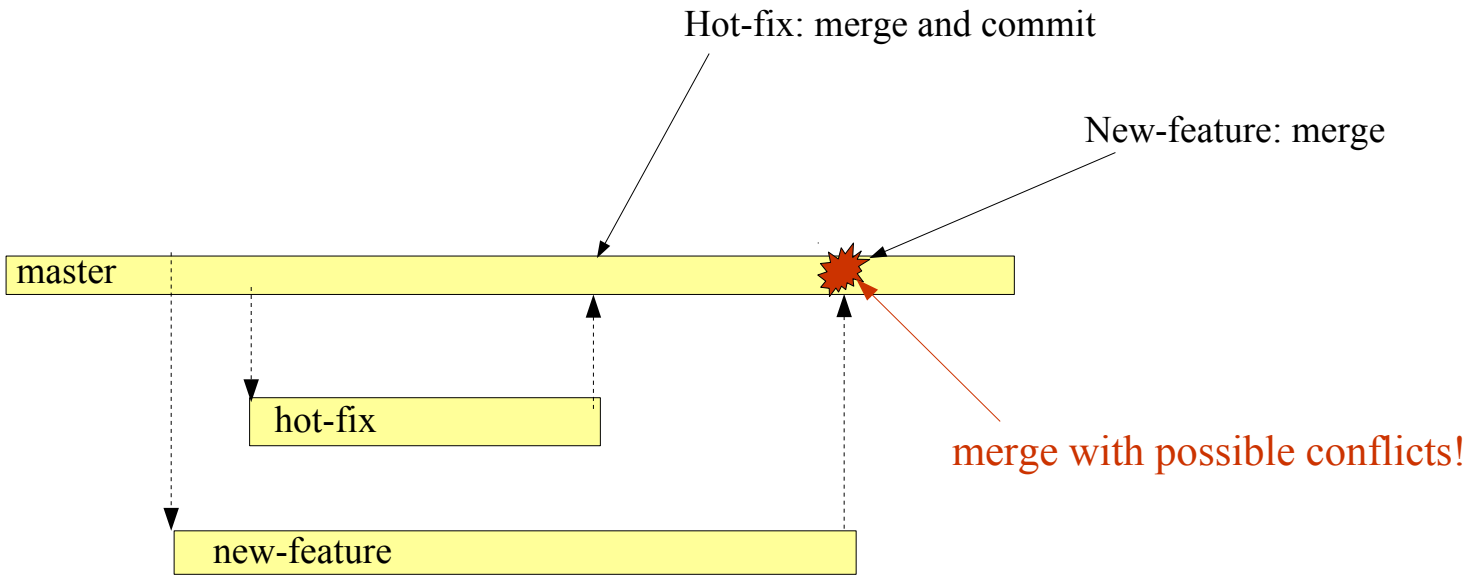
$ git stash
$ git checkout -b risky-try
$ git stash apply

… keep working… ⟵ Then comit or toss...



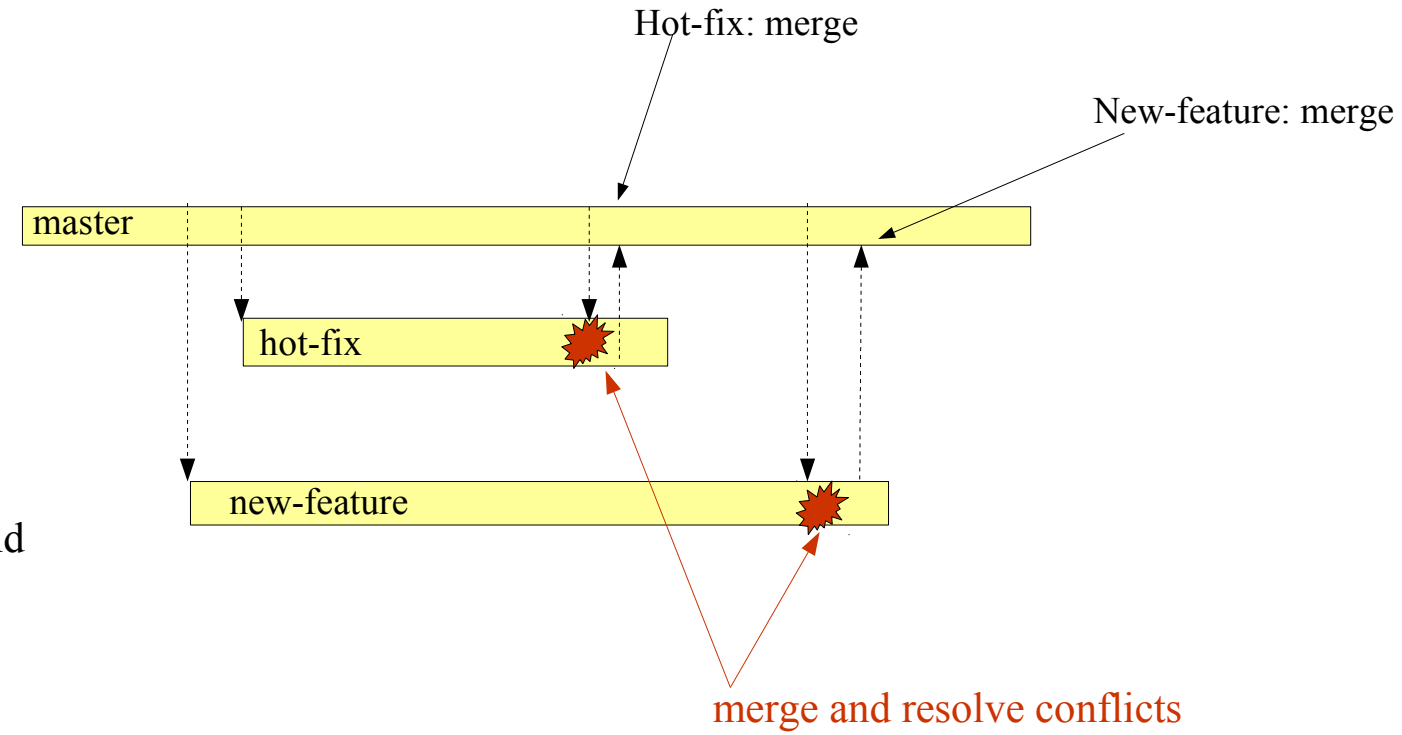
Using GIT branches

- Merge conflicts – the hot-fix syndrome
  - You are working on a new feature… hoping for a few days of quiet dev-time
  - Of course, a bug is found in your committed code → a hot-fix is necessary
  - When done with the hot-fix, you commit and merge onto master
  - Going to you new new feature…
  - *When done, can you merge to master?*



Using GIT branches

- Always merge *master* on your local branch first
  - Most updates on the same files are merged without conflicts
  - Sometimes a manual merge is necessary
  - You will need to practice it with your favorite merge tool

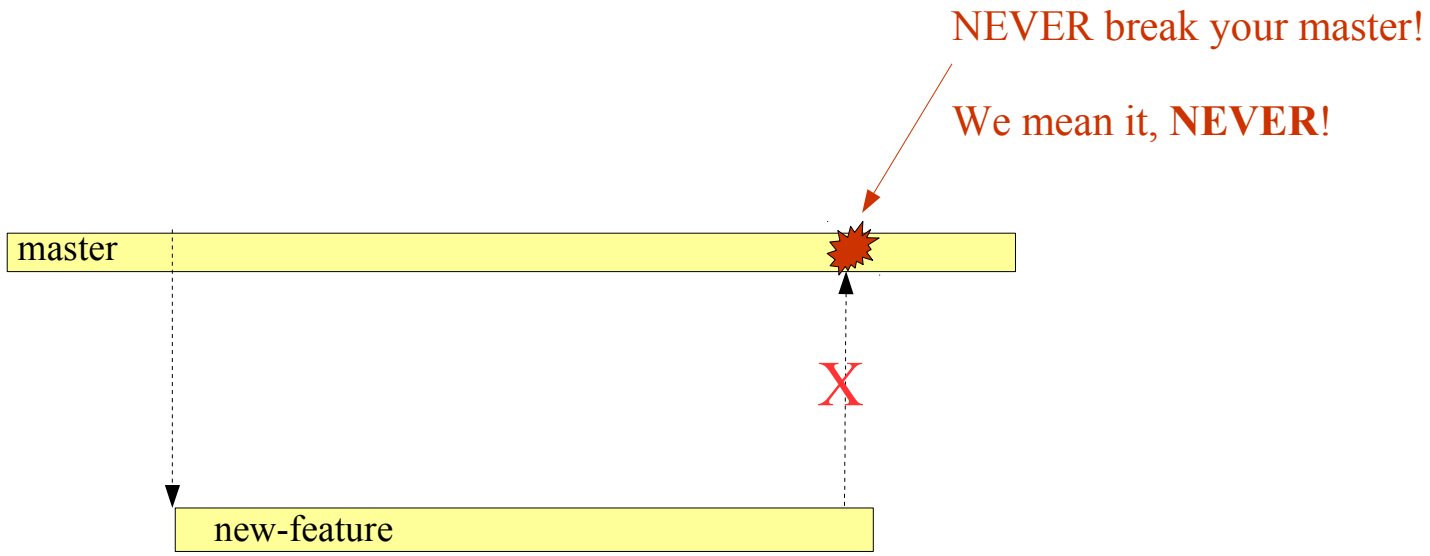


$ git mergetool --tool=meld

$ man git-mergetool

Using GIT branches

- Merge conflicts
  - **Never** merge on your master with potential conflicts
  - **...**

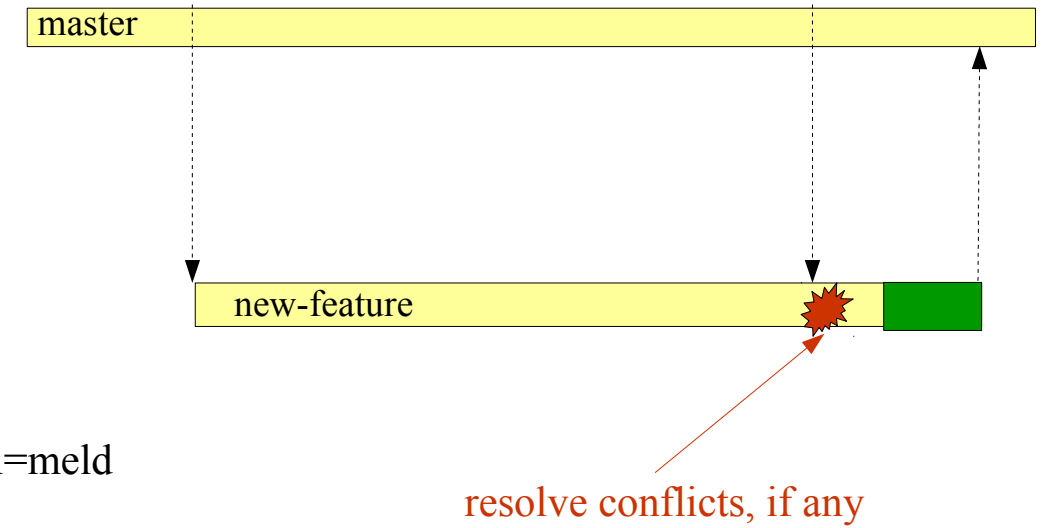



```
$ git mergetool --tool=meld
```

```
$ man git-mergetool
```

Using GIT branches

- **Never break your master branch**
  - **Never** merge on your master with potential conflicts
  - **Always** merge on your branch first
  - **...**

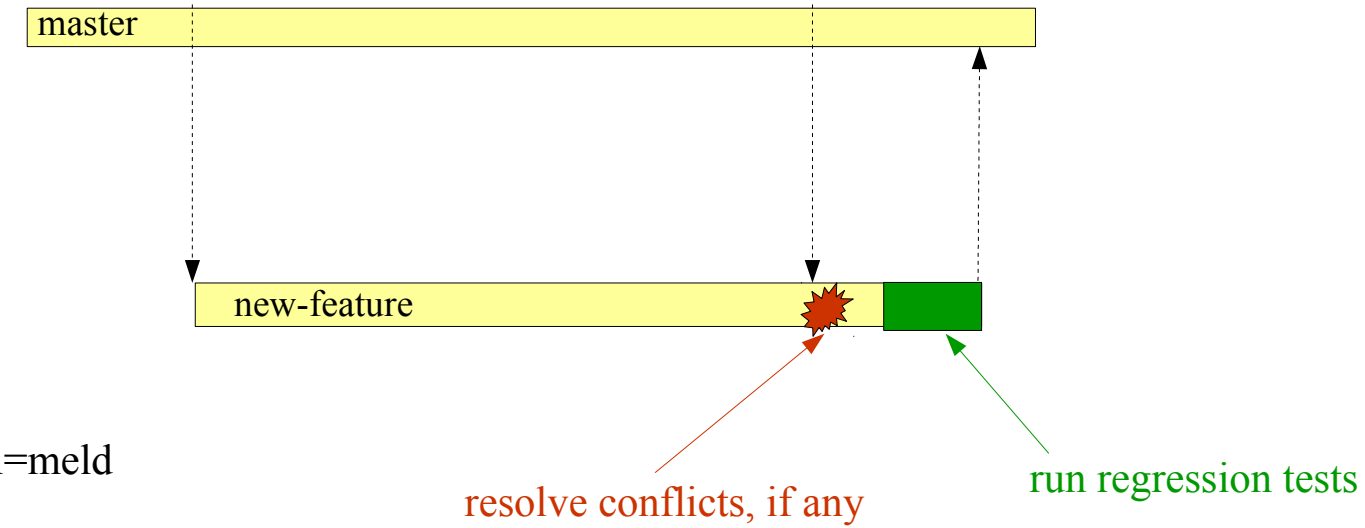


$ git mergetool --tool=meld

$ man git-mergetool

Using GIT branches

- **Never break your master branch**
  - **Never** merge on your master with potential conflicts
  - **Always** merge on your branch first
  - **Always** run regression tests before merging back on master

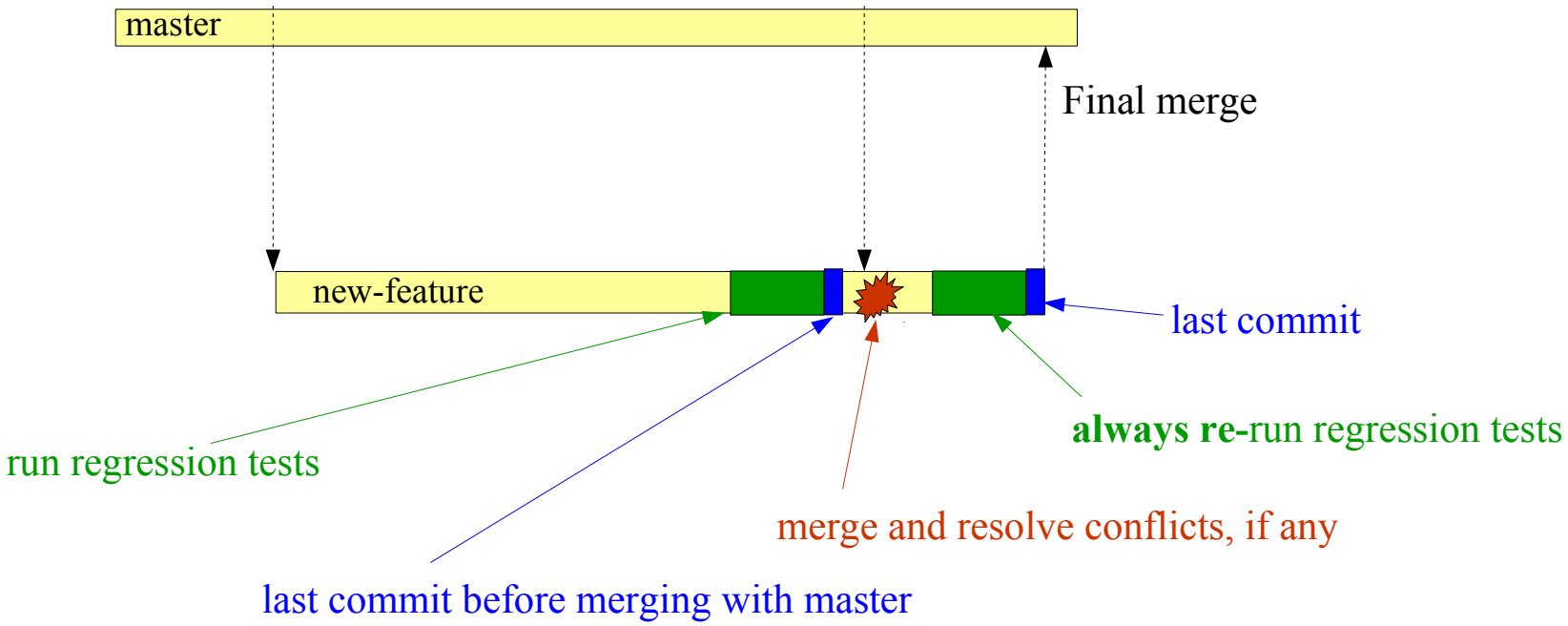


$ git mergetool --tool=meld

$ man git-mergetool

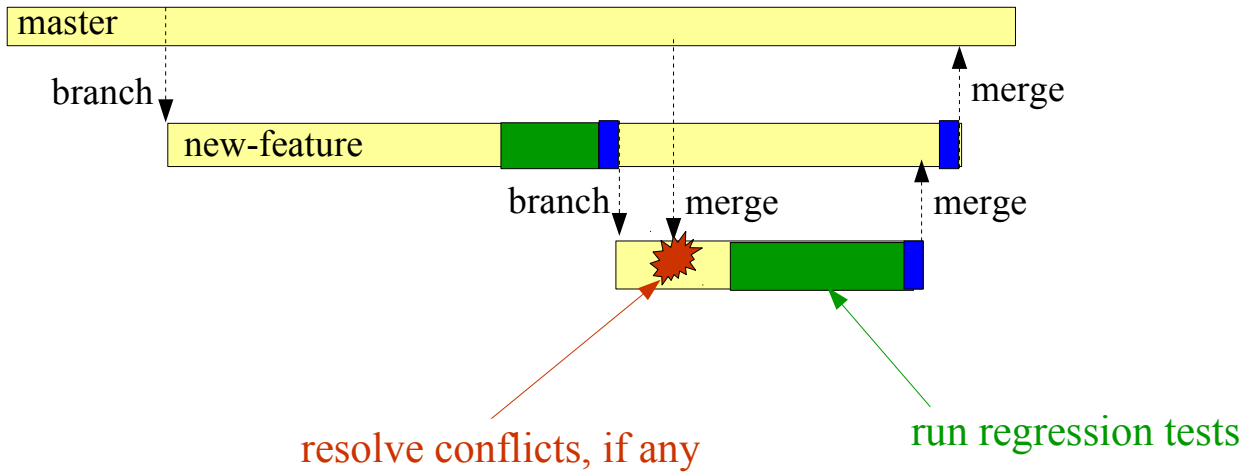
Using GIT branches

- Safe Merge Workflow
  - **(1) run regression tests,** they must pass before you can considering your work done
  - **(2) commit on your branch**
  - **(3) merge** the master branch on your branch
  - **(4) if the merge changed something → goto (1)**
  - **(5) merge** your branch in the master branch



Using GIT branches

- Safe Merge Workflow – Using a *"merge"* branch
  - Easy toss of the *"merge"* branch, if the merge should fail
  - Easier if you ever need to checkout back and forth across branches (hot-fix for examples)

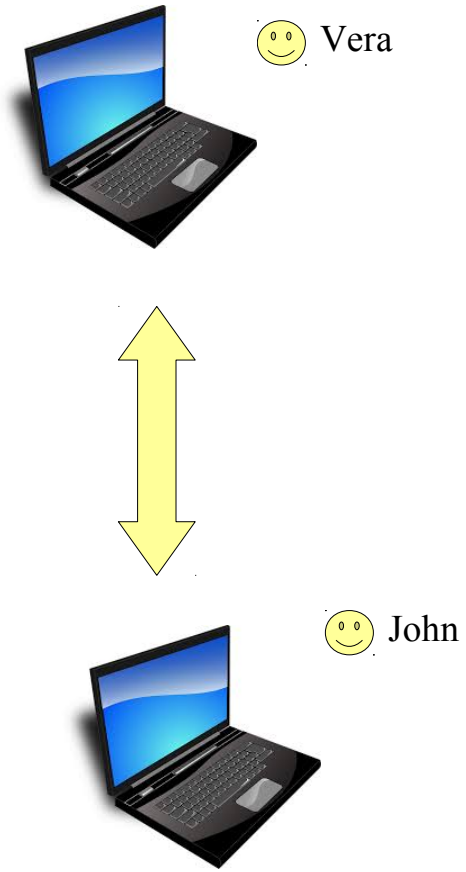


Using GIT – Practice makes perfect

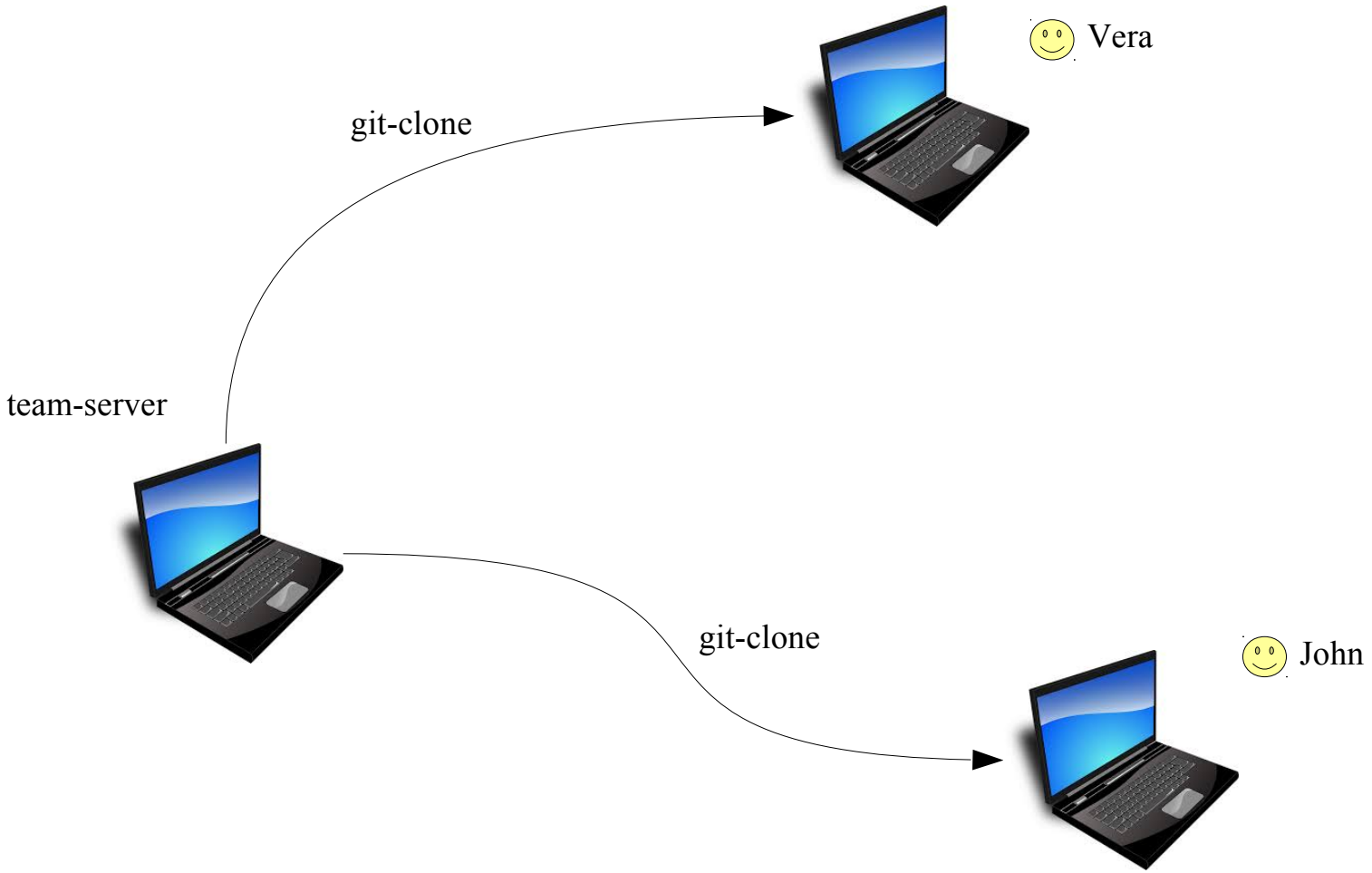
- **Do practice**
  - Again and again, until you are confortable with GIT
  - Scared of making a mistake – use branches
  - Really scared – try it on a copy of your project first
- **Do not procrastinate**
  - Until you have a team project
- **Keep it simple**
  - **You will be happy you did, believe us**
- **Keep doing your backups**
  - Better safe than sorry
  - Even if you use git clones…

Using GIT – Team Cooperation

- Vera and John need to work together on a project
  - They want to divide the work into features
  - Develop features independently
  - Merge features incrementally
- Distributed Git
  - Powerful and flexible tool
- Challenges
  - Keep it simple! Really.
  - Focus on keeping merges small and thus easy



Using GIT – Team Cooperation



Using GIT – Team Cooperation

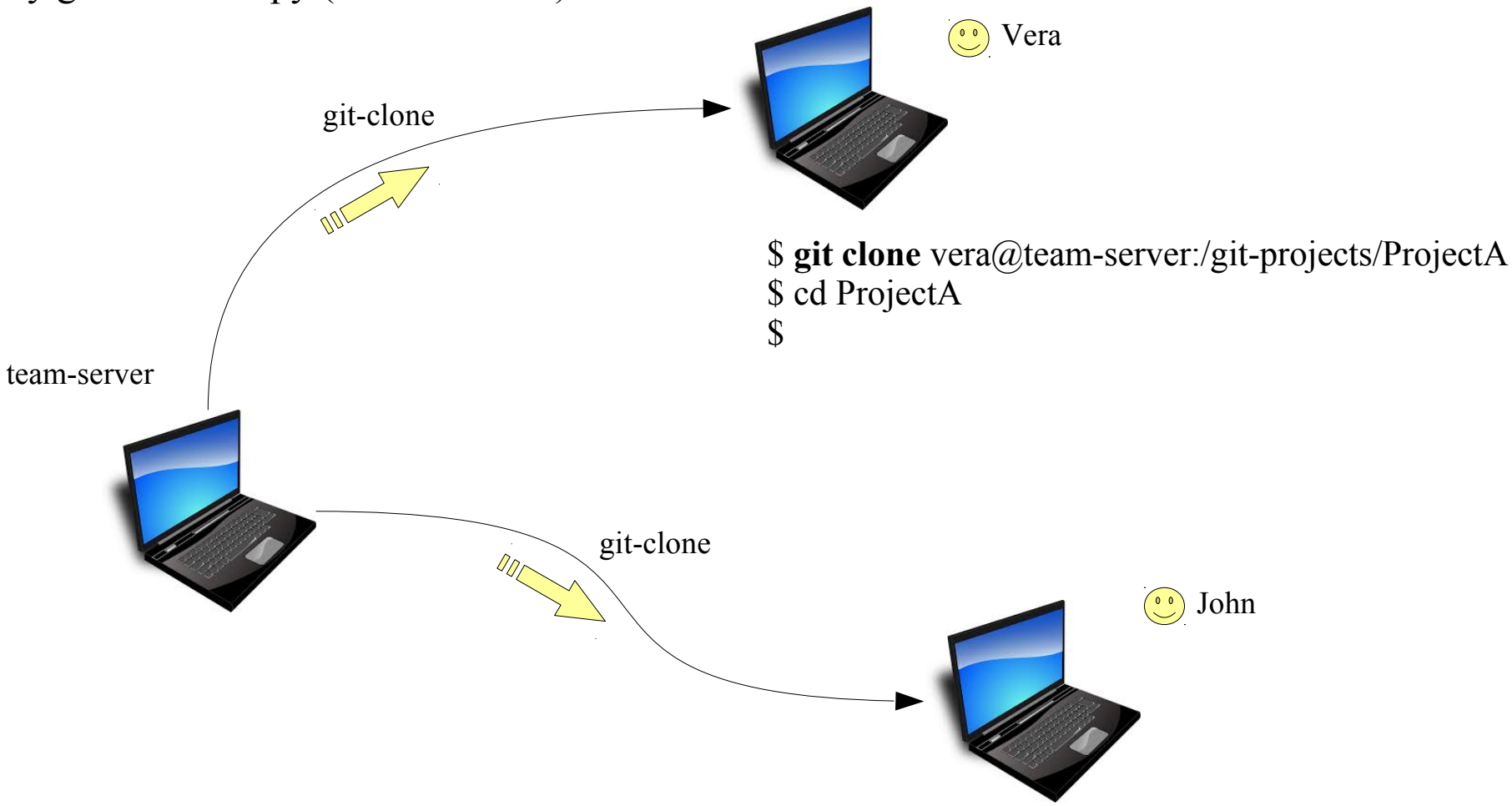
- Cloning repositories across machines
  - Create a repository (git-init), as a **shared bare** repository (--shared --bare)

team-server

```
$ mkdir -p /git-projects/ProjectA
$ cd /git-projects/ProjectA
$ git init --shared --bare
Initialized empty shared Git repository in /git-projects/ProjectA
$
```

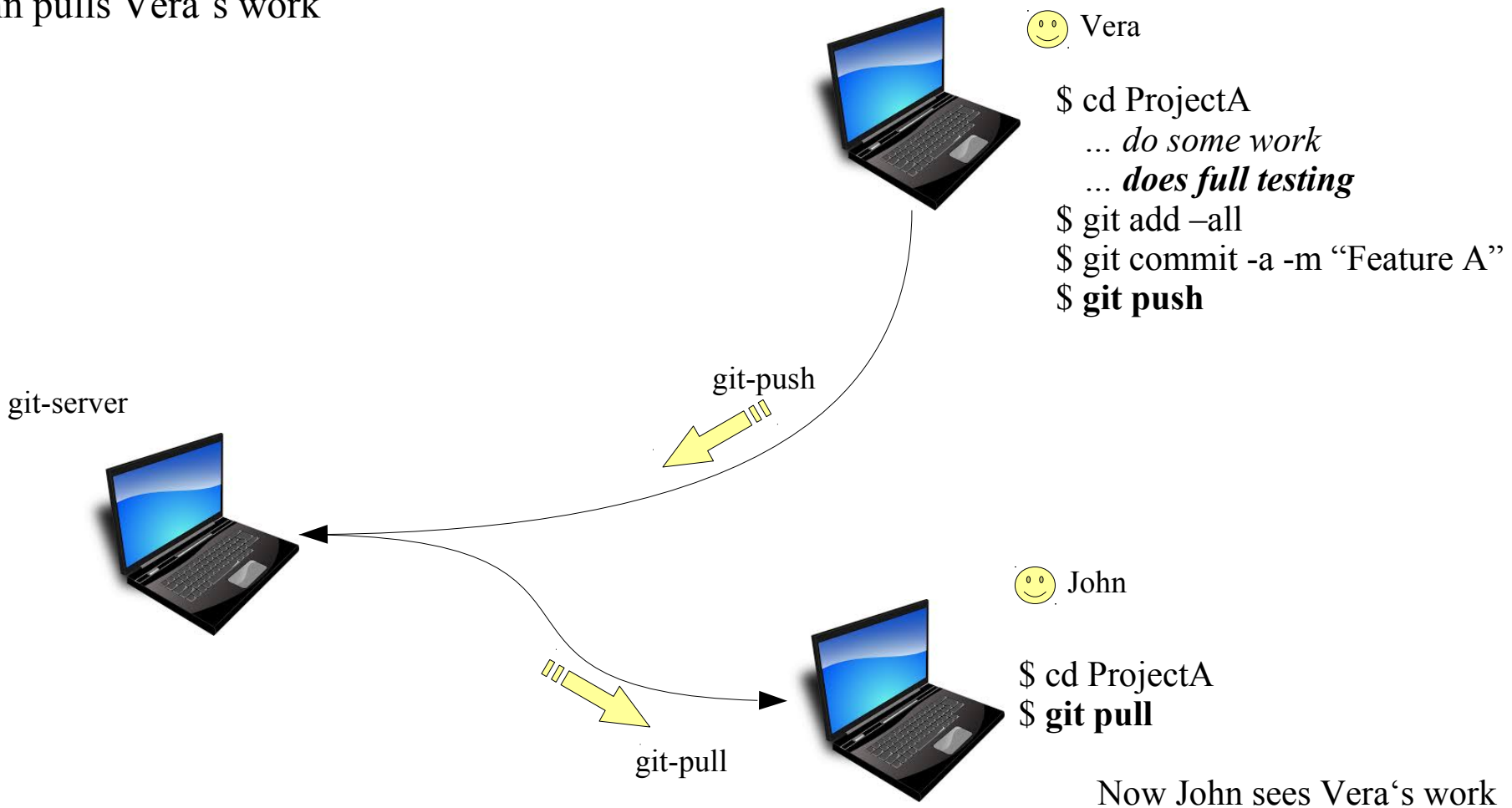
Using GIT – Team Cooperation

- Cloning repositories across machines
  - Team members clone that repository
  - They get a local copy (called a clone)



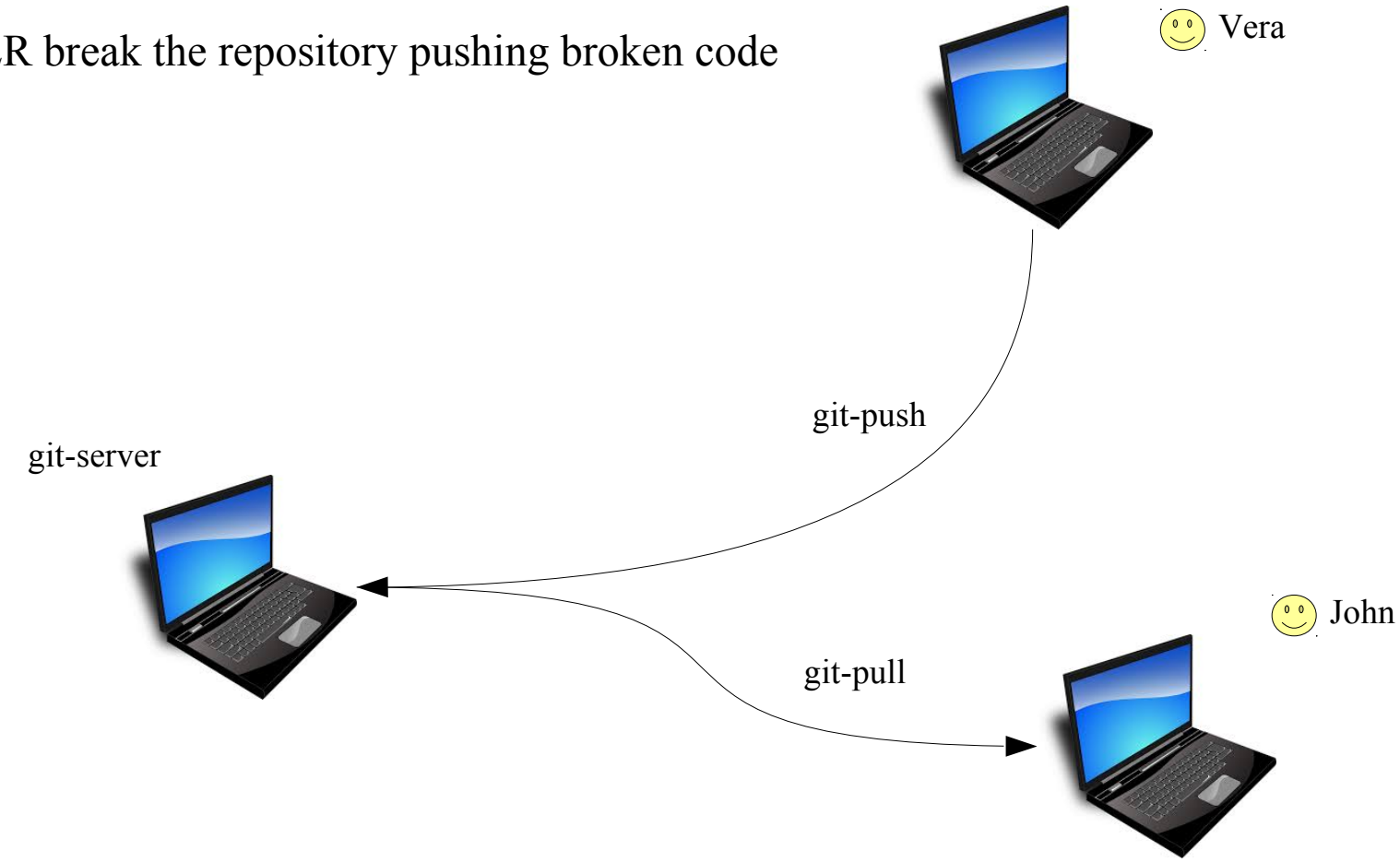
Using GIT – Team Cooperation

- Cooperation basics
  - Vera does some work and publishes it
  - John pulls Vera's work

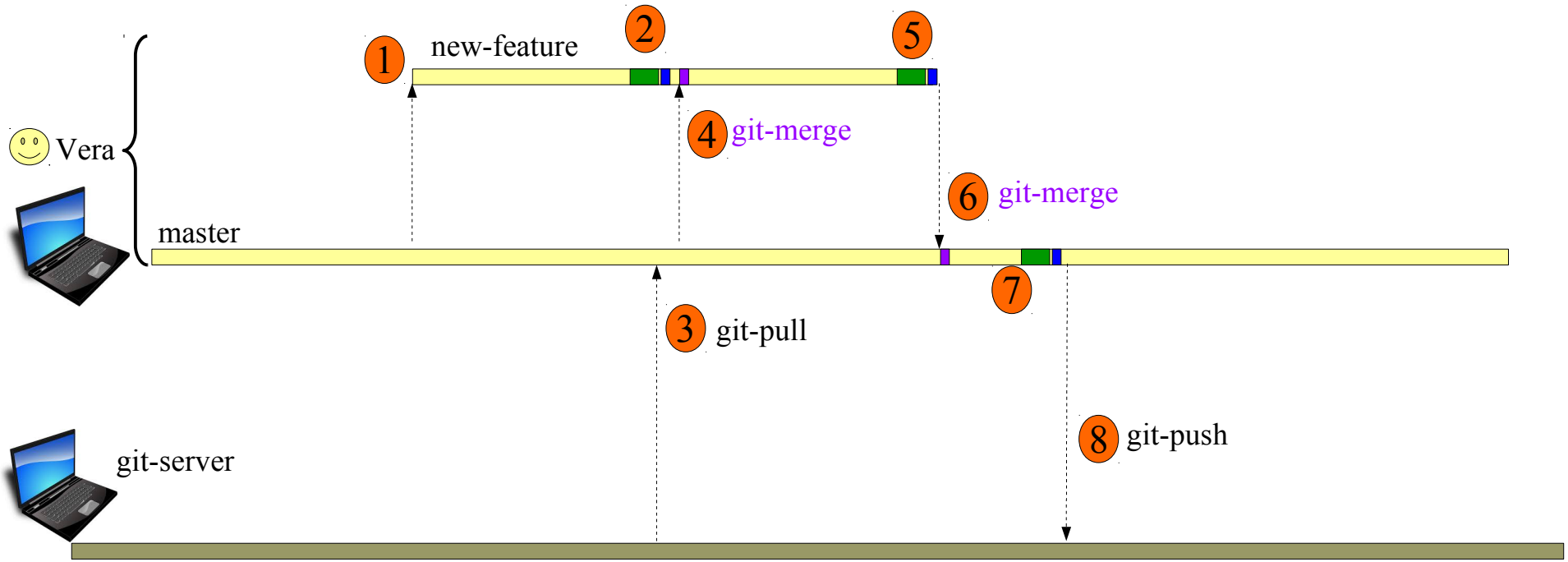


Using GIT – Team Cooperation

- Challenges
  - ALWAYS follow the same push-pull workflow
  - STRIVE to keep merges small and thus easy
  - NEVER break the repository pushing broken code



Using GIT – Team Cooperation

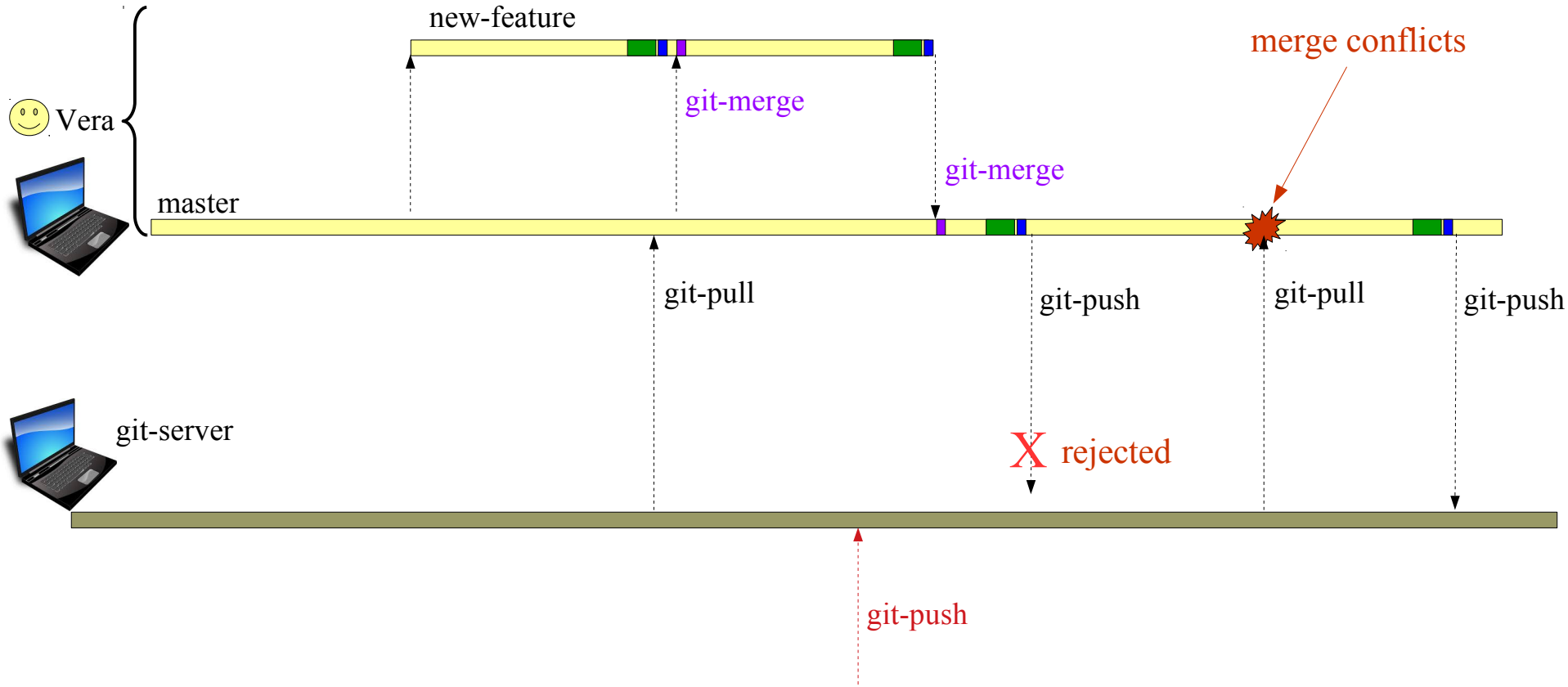


Vera's Workflow:

(1) create a branch to work in	git checkout -b vera.feature.a
(2) fully test and then commit	git commit ...
(3) git pull from server clone	git pull --all
(4) merge master on your branch	git merge master
(5) fully test and then commit	
(6) git-merge on your master	git checkout master; git merge vera.feature.a
(7) tests and commit on master	
(8) git push	

Using GIT – Team Cooperation

- Git-branch:

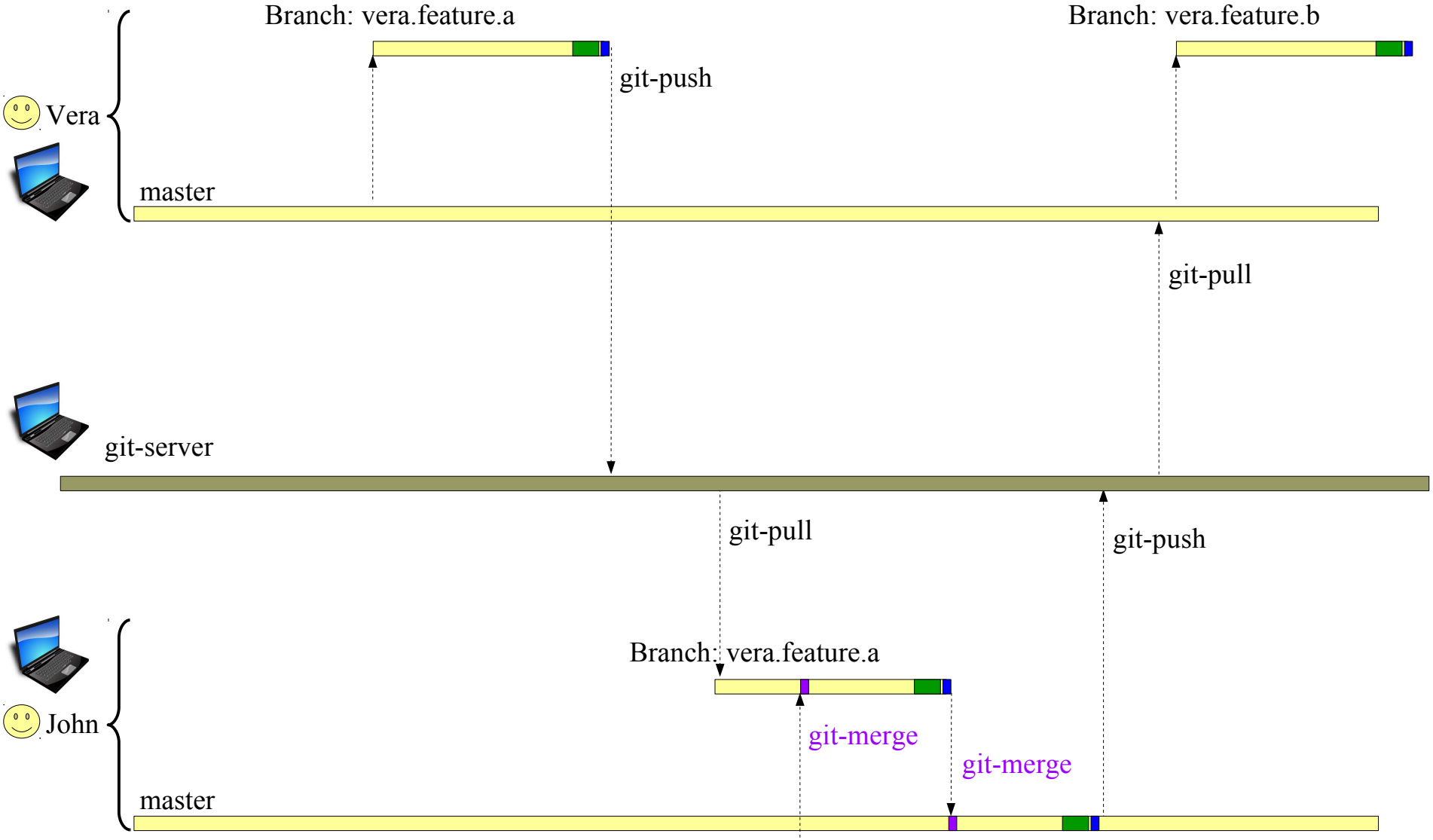


Using GIT – Team Cooperation

- Challenge of rejected git-push
  - You may break the master on your clone…
  - But you can still fix it, test it, commit it, and then push it
- The pull-request pattern
  - Only one team member does all merges on master
  - Other team developers pull the master when starting a branch, from master, for a new feature

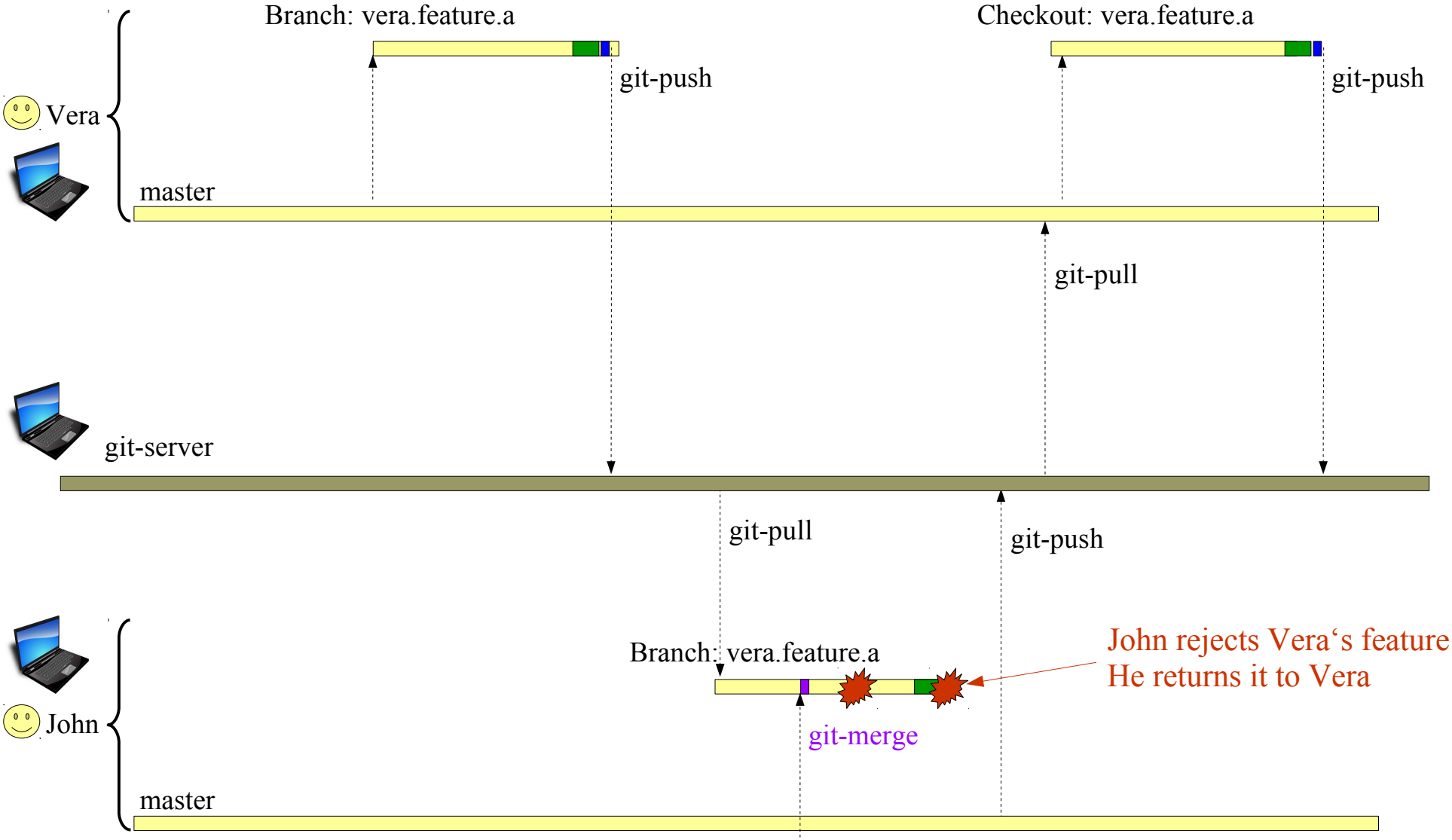
Using GIT – Team Cooperation

- Git-branch:



Using GIT – Team Cooperation

- Git-branch:



Conclusion

- Git
  - Powerful source control…
  - So keep it simple… even simpler… really… we mean it!
- When to use it?
  - Use it for your own personal development
  - Use it for team development, but requires rigorous testing

Practice it to learn it.

Practice some more before using it on real projects...

And do not forget, keep it simple. Really simple.

Don‘t forget to backup your server repo…