Versioning

Jaroslav Dytrych

Faculty of Information Technology, Brno University of Technology
Božetěchova 1/2. 612 66 Brno - Královo Pole
dytrych@fit.vutbr.cz

22. JULY 2021

Sharing data with team members

Means of sharing data with team members

- FTP/SFTP
- web
- version control system
- ...

- simplest and most available tool.

- simplest and most available tool.
- unsuitable for documents or source files that change
  - previous versions are lost.
  - conflicts (new version overwrites older, but edited version someone else had locally).
  - simultaneous write access?

- simplest and most available tool.
- unsuitable for documents or source files that change
  - previous versions are lost.
  - conflicts (new version overwrites older, but edited version someone else had locally).
  - simultaneous write access?
- suitable for sharing data that are created and don't change
  - can be downloaded via HTTP (simple link sharing).
  - effective even for large amounts of data.

- simplest and most available tool.
- unsuitable for documents or source files that change
  - previous versions are lost.
  - conflicts (new version overwrites older, but edited version someone else had locally).
  - simultaneous write access?
- suitable for sharing data that are created and don't change
  - can be downloaded via HTTP (simple link sharing).
  - effective even for large amounts of data.
- permission issues!
  - if the FTP is hosted for free, everything can be accessed via web by default.
  - competition can still find your data even without a link.

- a number of information and file sharing services available.

- a number of information and file sharing services available.
- data storage can be created even on the project web.

- a number of information and file sharing services available.
- data storage can be created even on the project web.
- similar to FTP, but usually with file size restrictions.
  - usually only suitable for published files.

- a number of information and file sharing services available.
- data storage can be created even on the project web.
- similar to FTP, but usually with file size restrictions.
  - usually only suitable for published files.
- always make sure your data are in safety and secured.

Other file sharing services

- Dropbox
- Google Drive
- TeamDrive

- Be mindful of restrictions and paid services.
- They don't always fulfil your expectations (the option to return to previous state may not be enough).

Version control system

Version control system

- Labeling:
  - RCS – Revision Control System (also name of the tool)
  - VCS – Version Control System
- intended for management and sharing of documents/programs source files, that change frequently throughout the development time or are developed by more than one person.

Version control system

- Labeling:
  - RCS – Revision Control System (also name of the tool)
  - VCS – Version Control System
- intended for management and sharing of documents/programs source files, that change frequently throughout the development time or are developed by more than one person.
- older versions of files can be found.
- conflicts can be resolved (new version overwritten by an older, edited version).
- tracks every change, so that the project can return to any previous state, when it still worked.
- multiple development branches (e.g. stable branch and experimental branch, or special version for a specific customer), switching between branches and merging them.

Version control system

- Labeling:
  - RCS – Revision Control System (also name of the tool)
  - VCS – Version Control System
- intended for management and sharing of documents/programs source files, that change frequently throughout the development time or are developed by more than one person.
- older versions of files can be found.
- conflicts can be resolved (new version overwritten by an older, edited version).
- tracks every change, so that the project can return to any previous state, when it still worked.
- multiple development branches (e.g. stable branch and experimental branch, or special version for a specific customer), switching between branches and merging them.
- unsuitable for large files with testing or other data, that don't need to be versioned (forces other team members to download them).

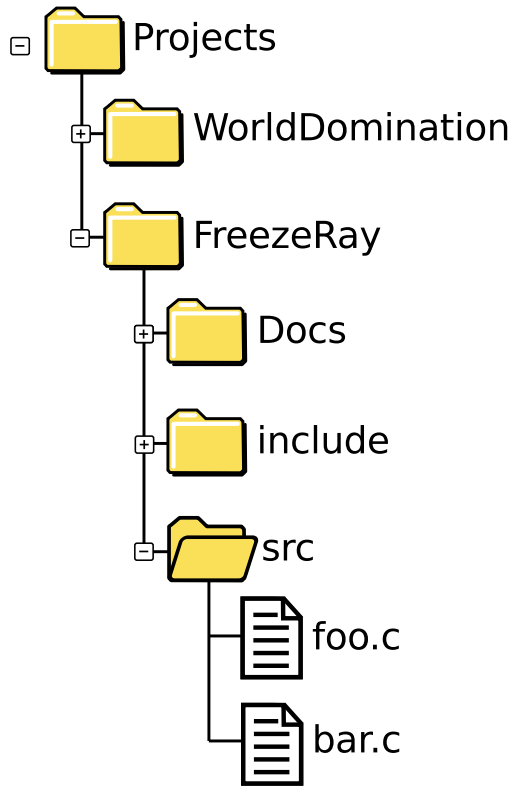
- Obsolete:
  - 1972 SCCS – only 1 developer in a single directory
  - 1980 RCS
  - 1986 CVS – multiple developers, central server
  - 1999 Subversion (SVN) – atomical commits
- Older, underused (more difficult to use, not so scalable), distributed:
  - 2001 Arch, Monotone
  - 2002 Darcs
- Current, distributed (and supported in IDEs):
  - 2005 Git – created by Linus Torvalds for Linux kernel
  - 2005 Mercurial (hg) – Matt Mackall, Git alternative
  - 2005 Bazaar (bzr) – Canonical, evolution of Arch, more options, but slower

Basic terms

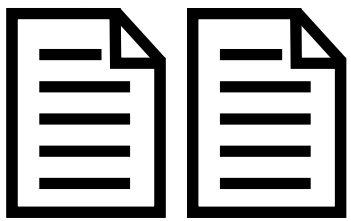
- **Working tree**
  - directory structure with source files (exists even without VCS)
- **Staging area**
  - space for whatever is to be commited (saved to repository), that means added, removed, renamed or changed files
  - does not have to contain all changes in working tree
- **Commit**
  - set of changes within a repository (unit of developer's work)
  - defines a specific state of a project (version)
- **Branch**
  - user-named set of commits
  - branches allow for parallel development (different development routes)
- **Repository**
  - database containing history of a project
- **Cloned repository**
  - copy of repository with working tree
- **Bare repository**
  - copy of repository without working tree

Working Tree

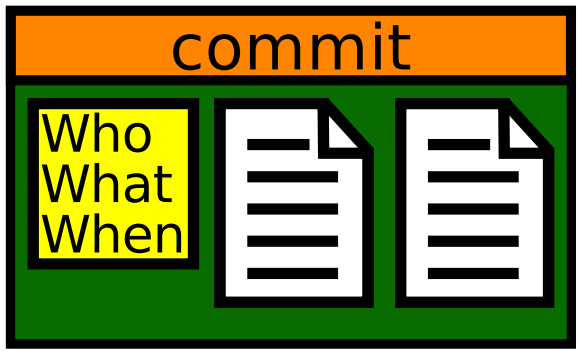
- directory structure
- source files



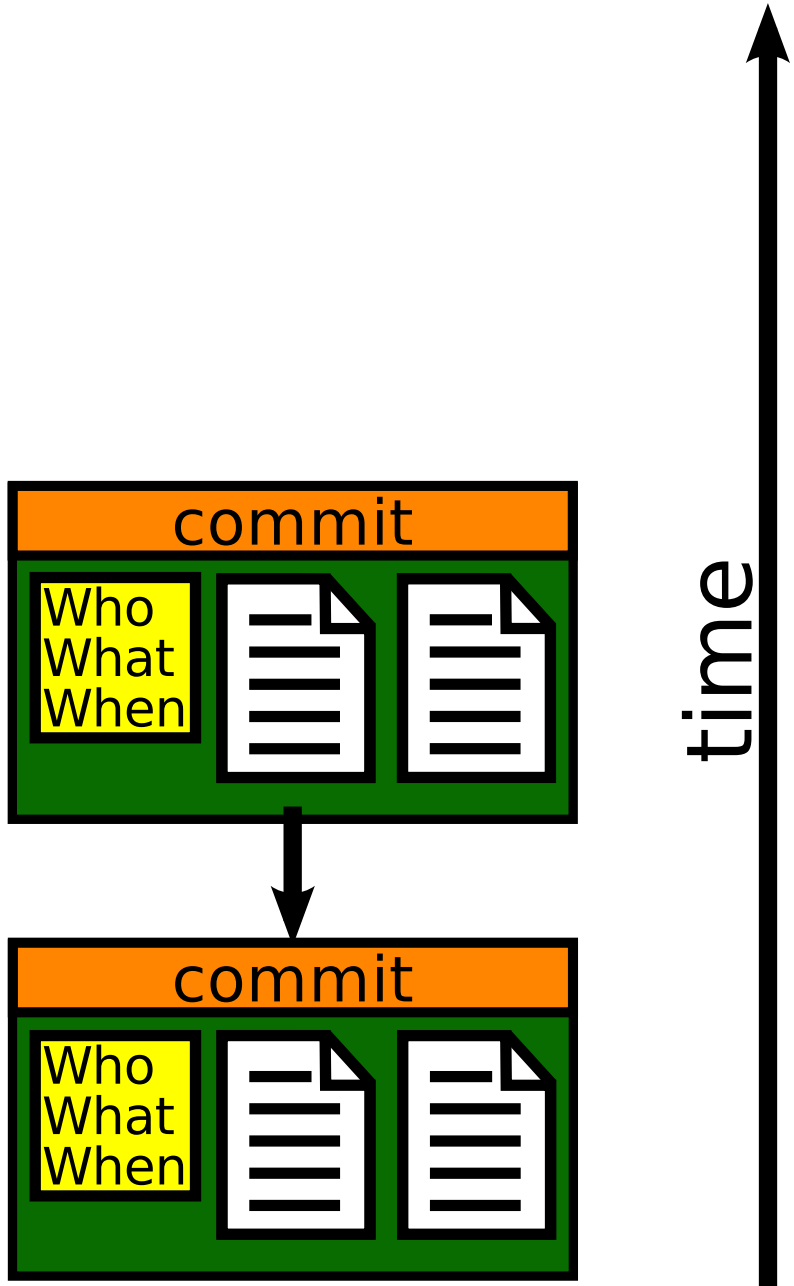
- File contents



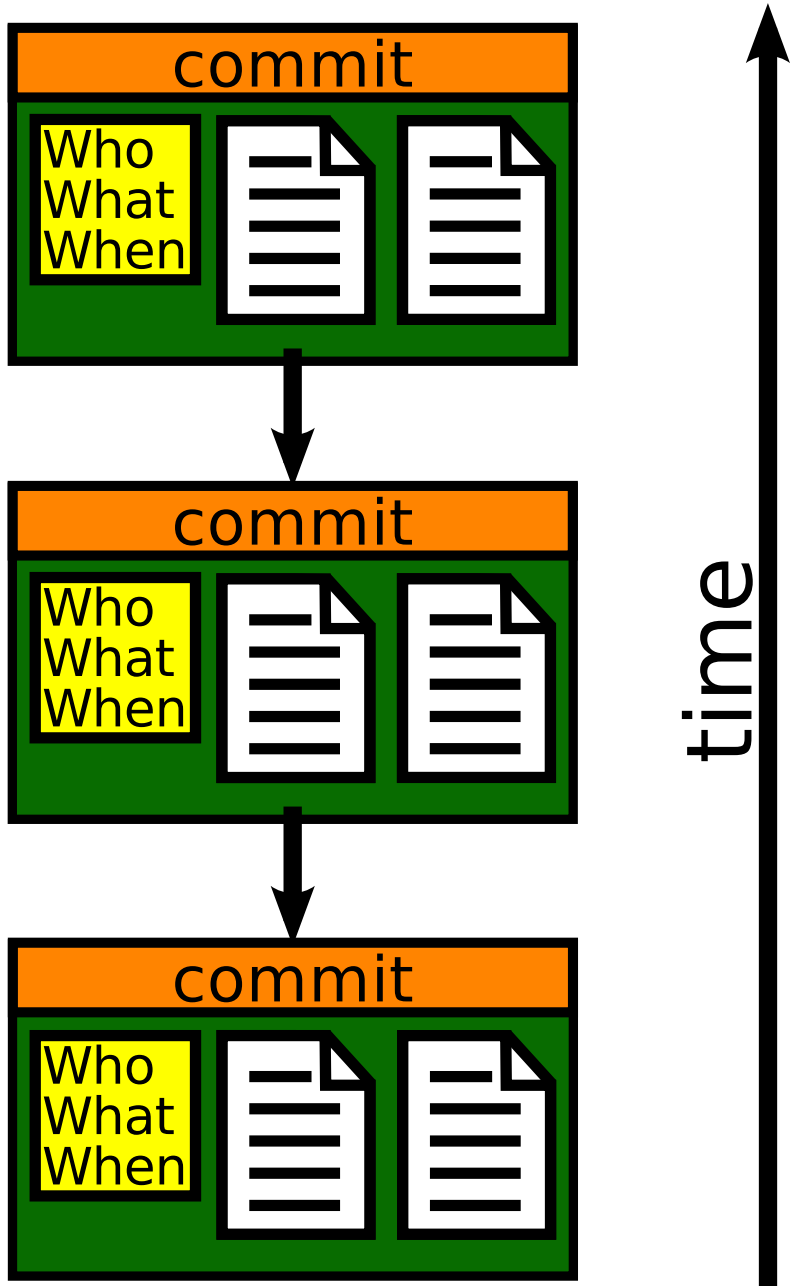
- File contents
- “Commits”



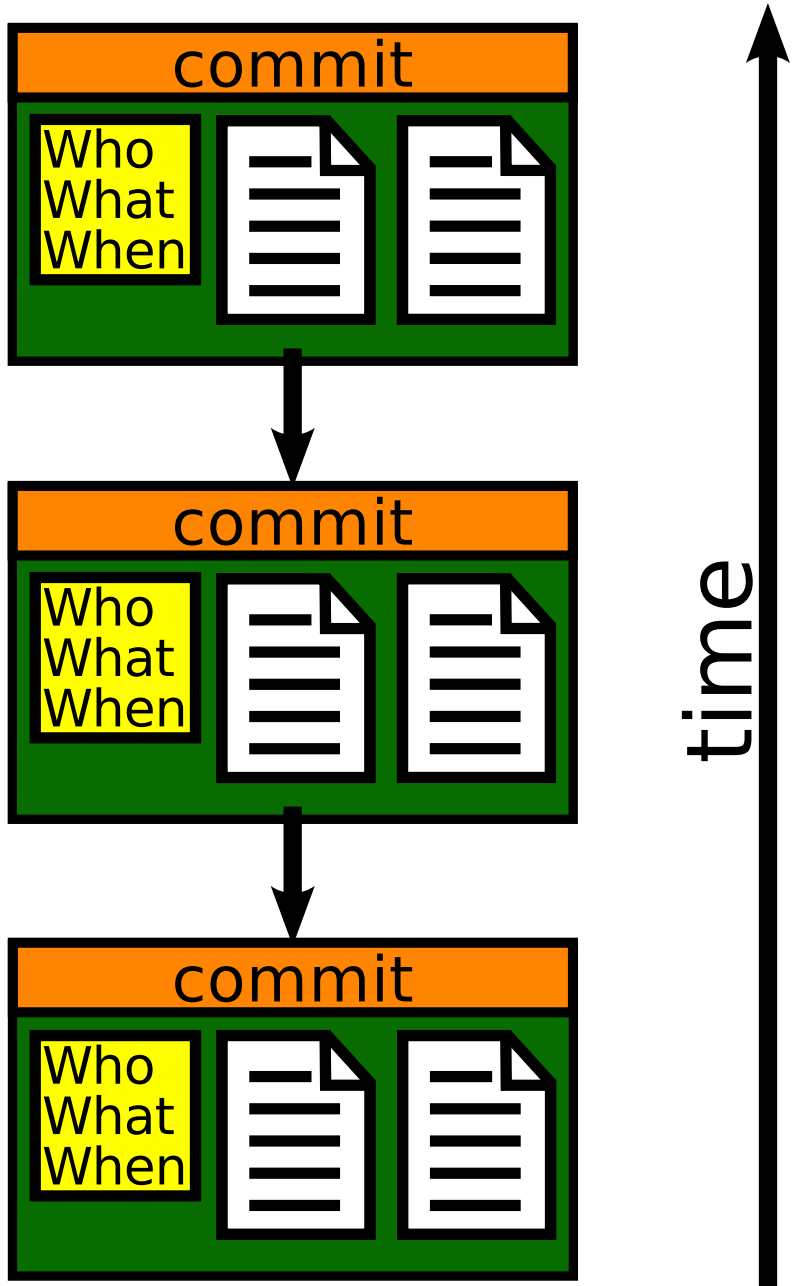
- File contents
- “Commits”
- “Ancestry chain”



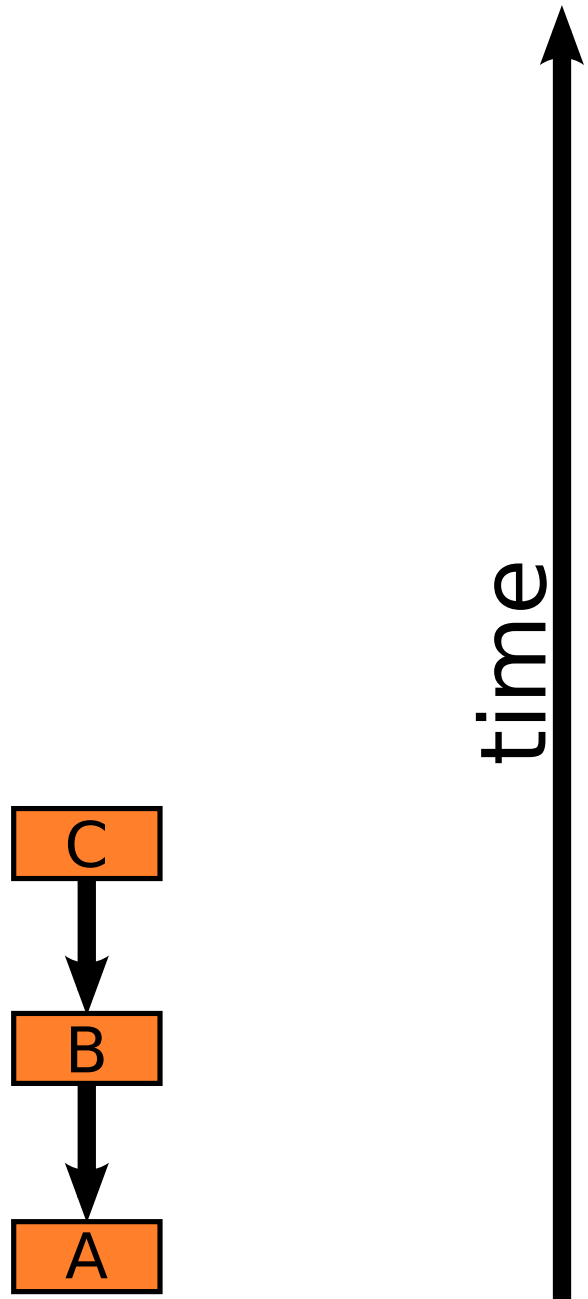
- File contents
- “Commits”
- “Ancestry chain”



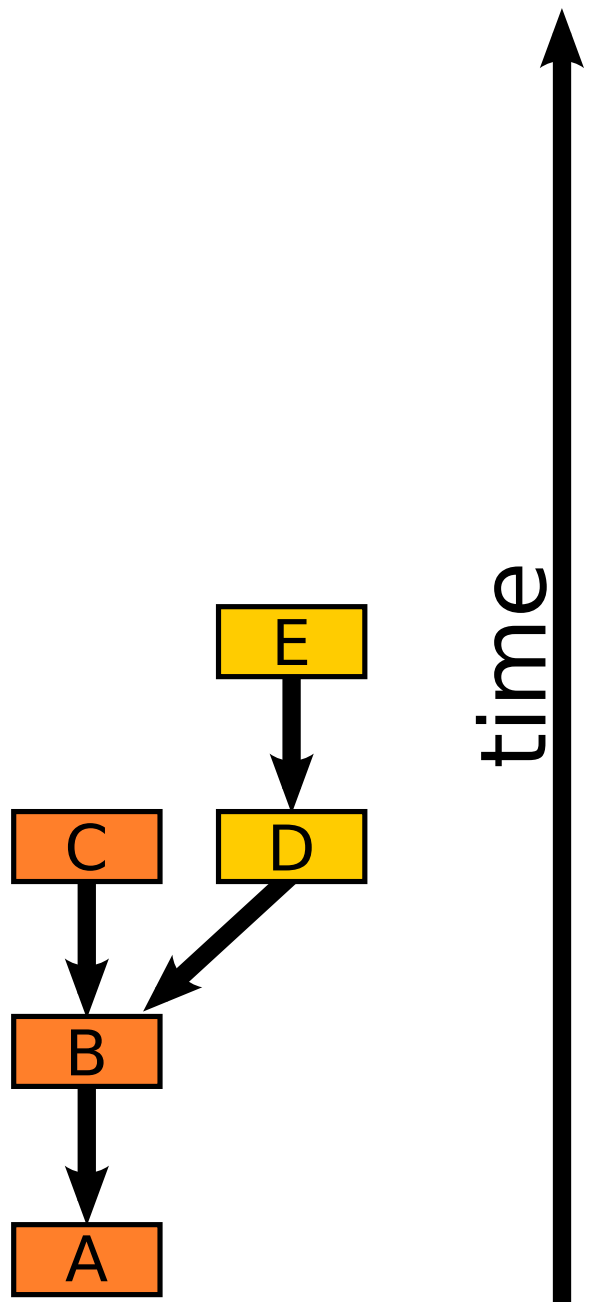
- File contents
- “Commits”
- “Ancestry chain”
  - Link to direct ancestor



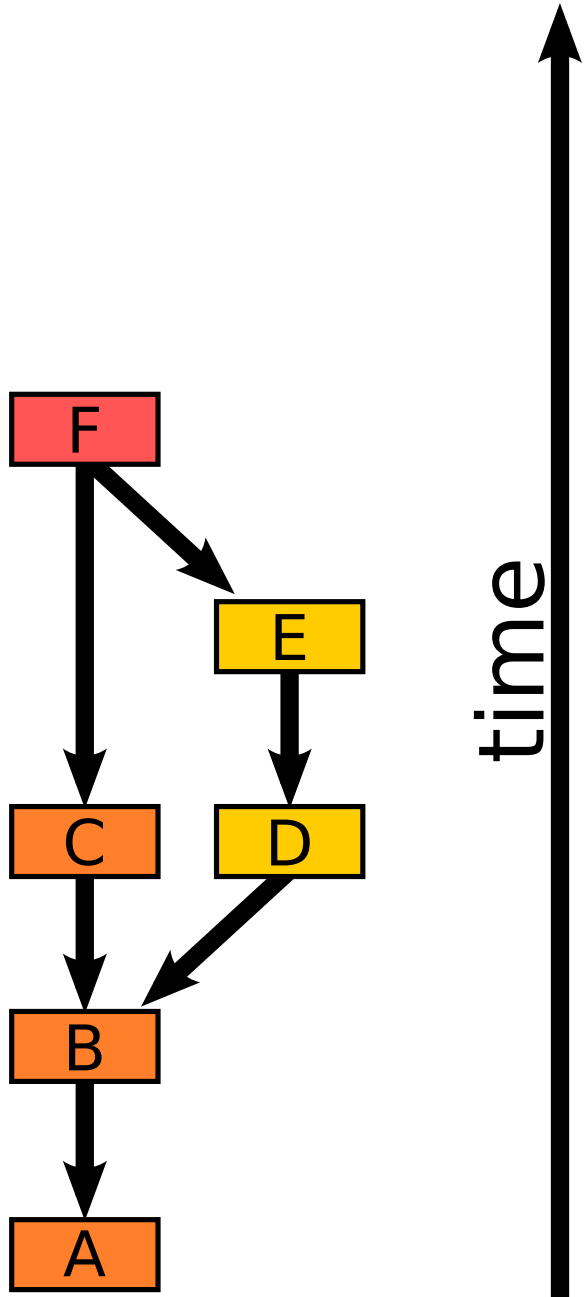
- File contents
- “Commits”
- “Ancestry chain”
  - Link to direct ancestor



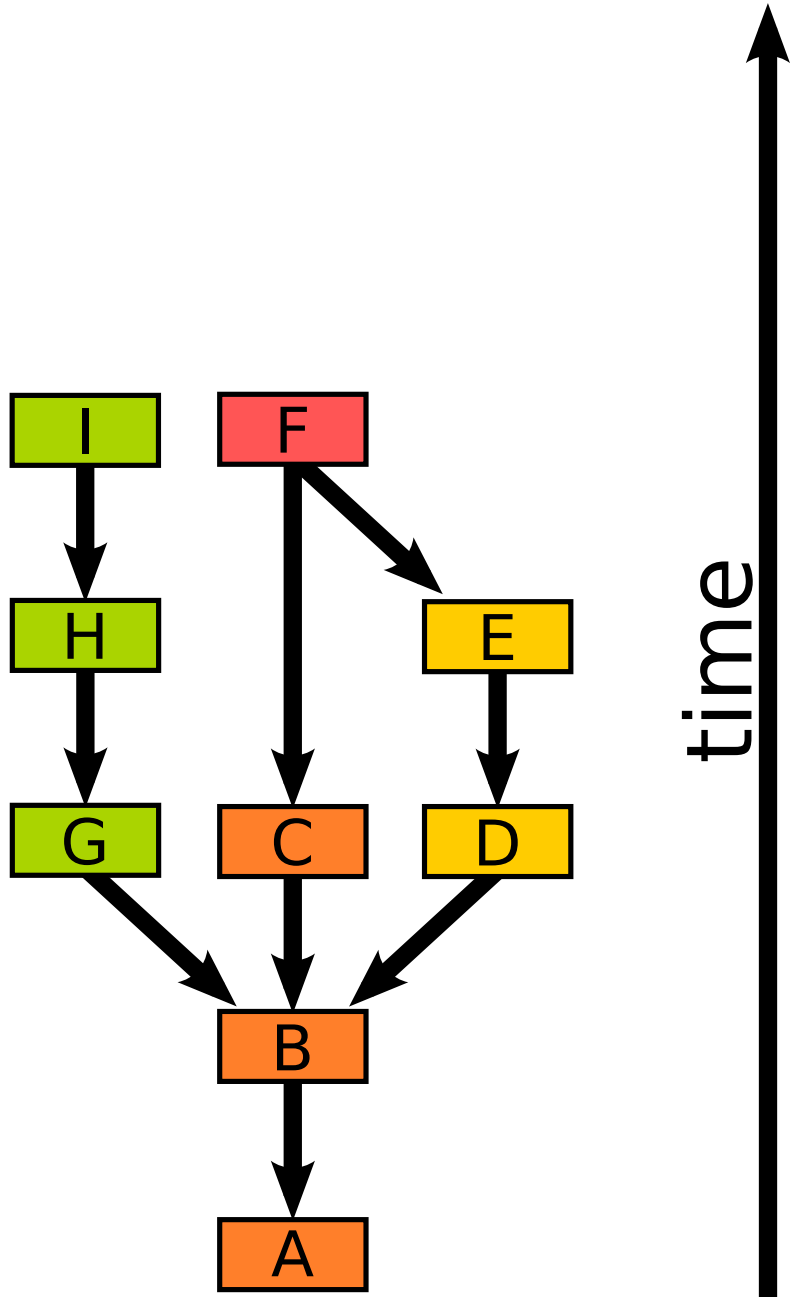
- File contents
- “Commits”
- “Ancestry chain”
  - Link to direct ancestor
  - branches



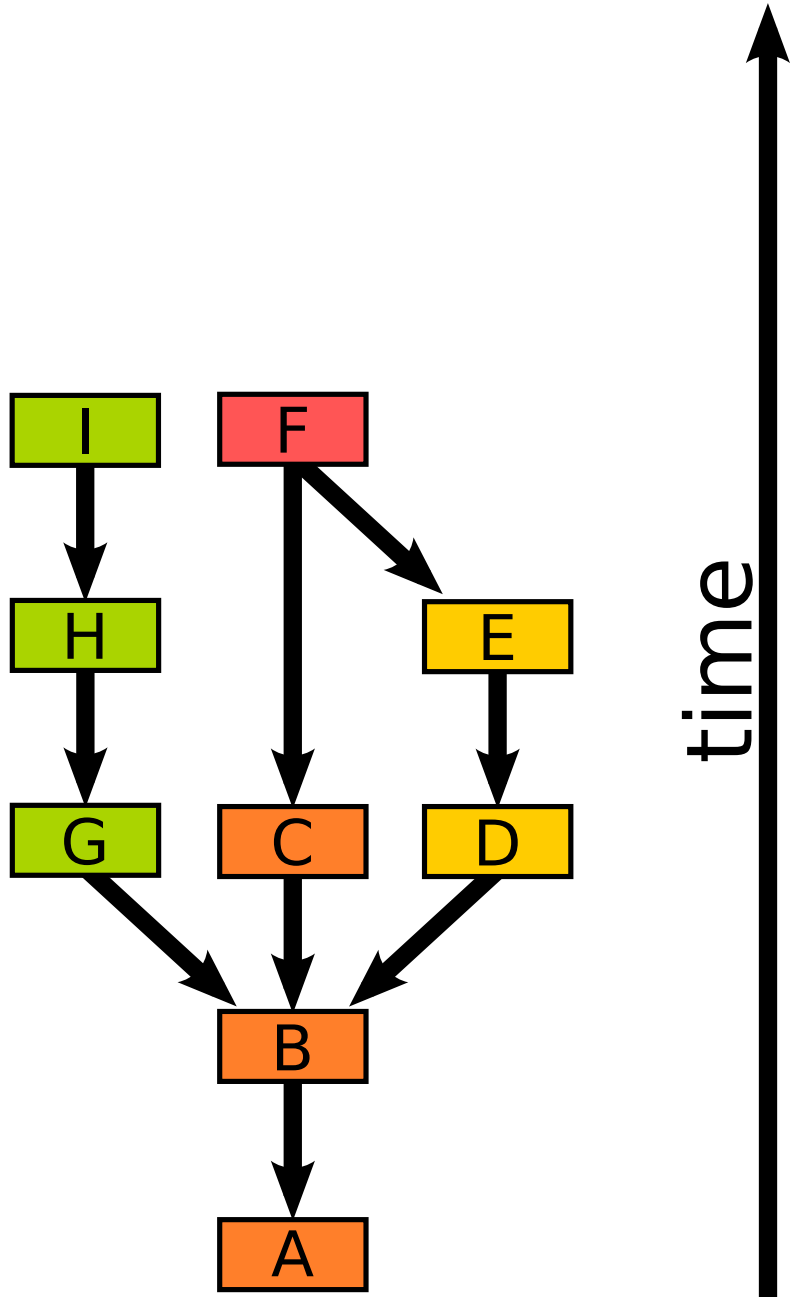
- File contents
- “Commits”
- “Ancestry chain”
  - Link to direct ancestor
  - branches
  - merges



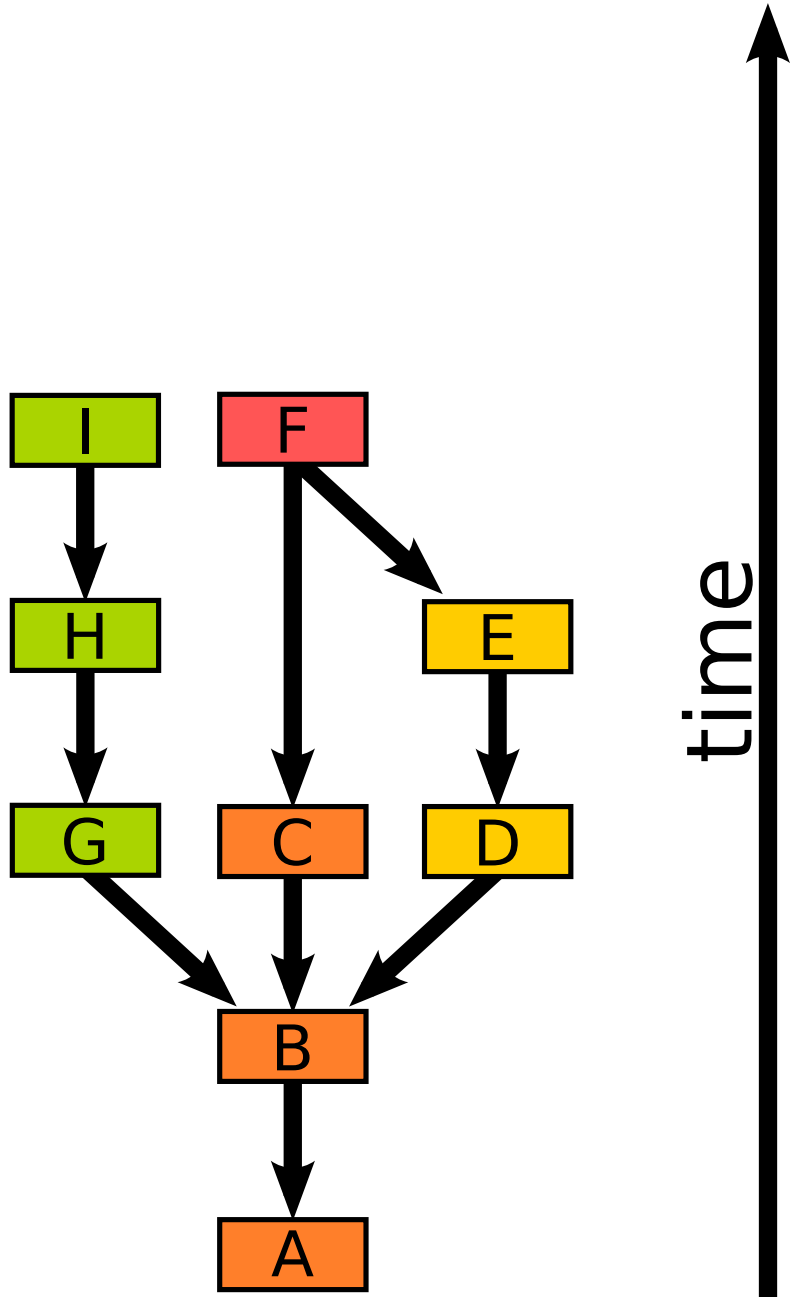
- File contents
- “Commits”
- “Ancestry chain”
  - Link to direct ancestor
  - branches
  - merges



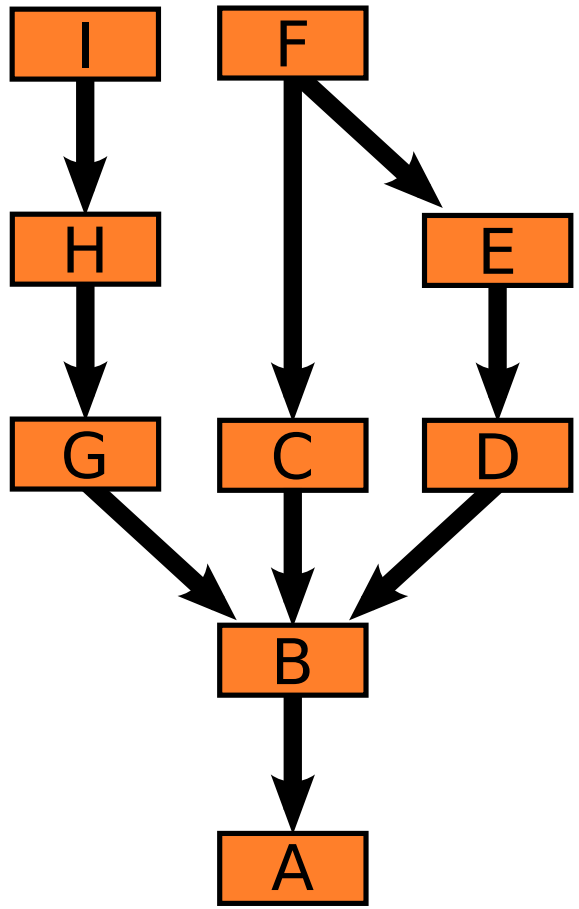
- File contents
- “Commits”
- “Ancestry chain”
  - Link to direct ancestor
  - branches
  - merges
- Head / Tip
  - youngest commit in branch



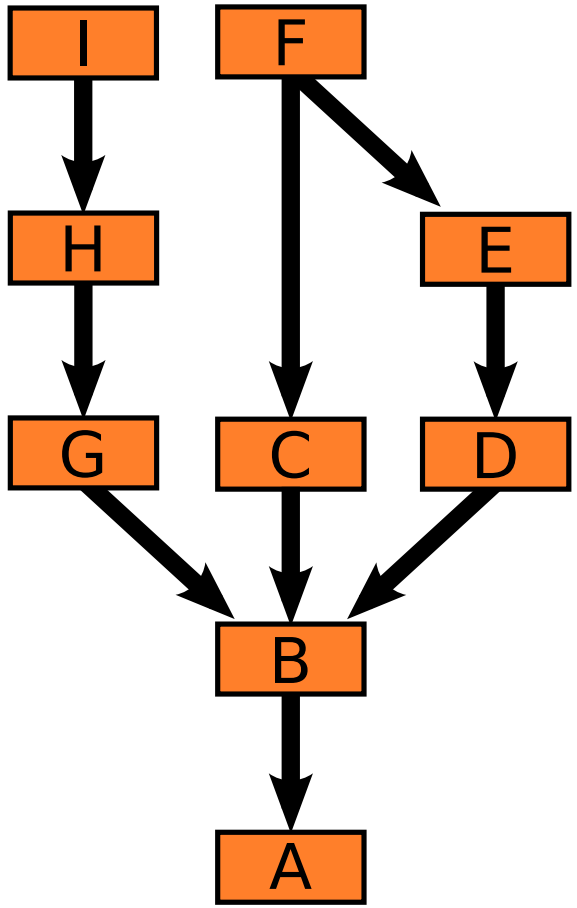
- File contents
- “Commits”
- “Ancestry chain”
  - Link to direct ancestor
  - branches
  - merges
- Head / Tip
  - youngest commit in branch
- Multi-head / Multi-tip repository
  - multiple unmerged branches



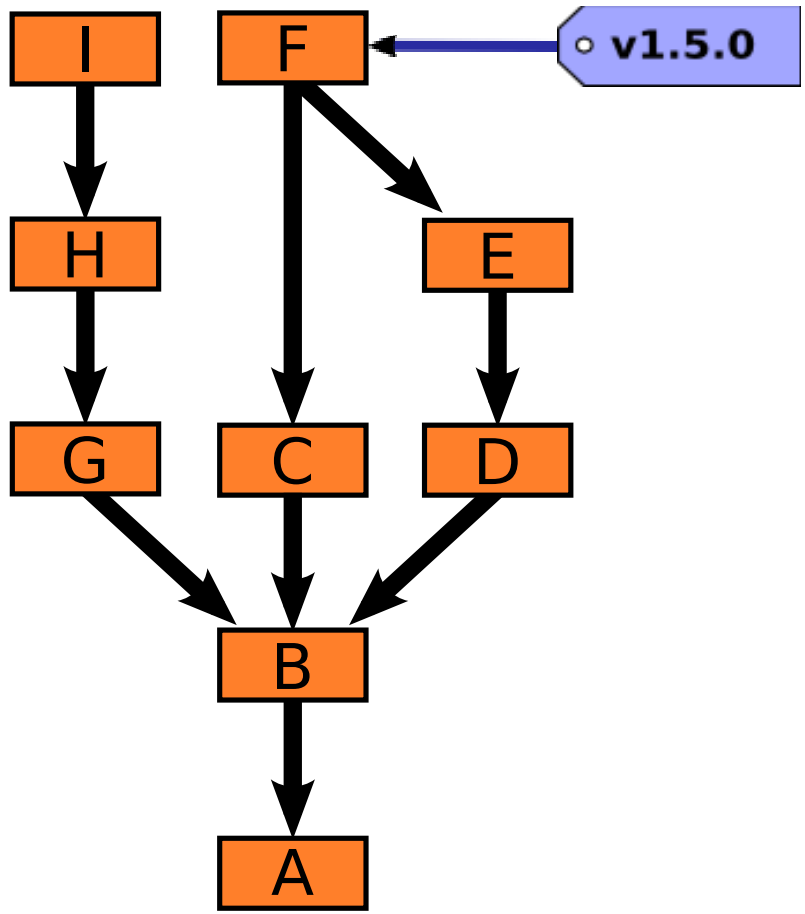
Directed acyclic graph (DAG)



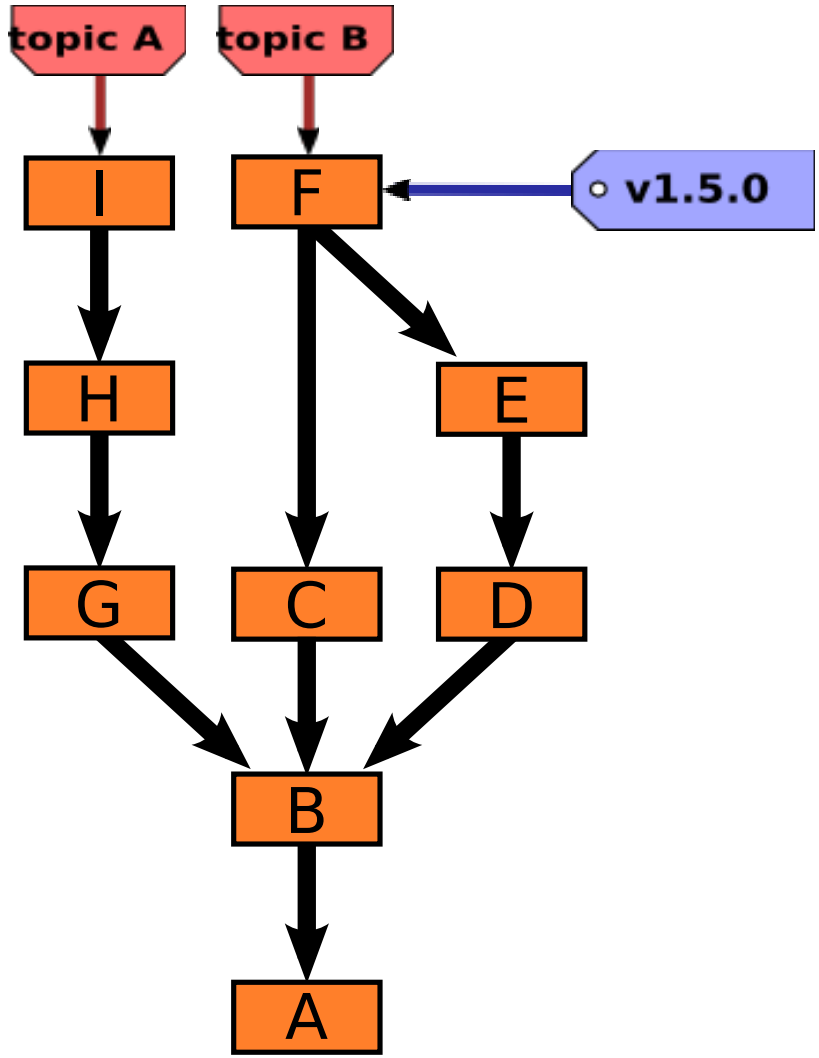
References are pointers to DAG



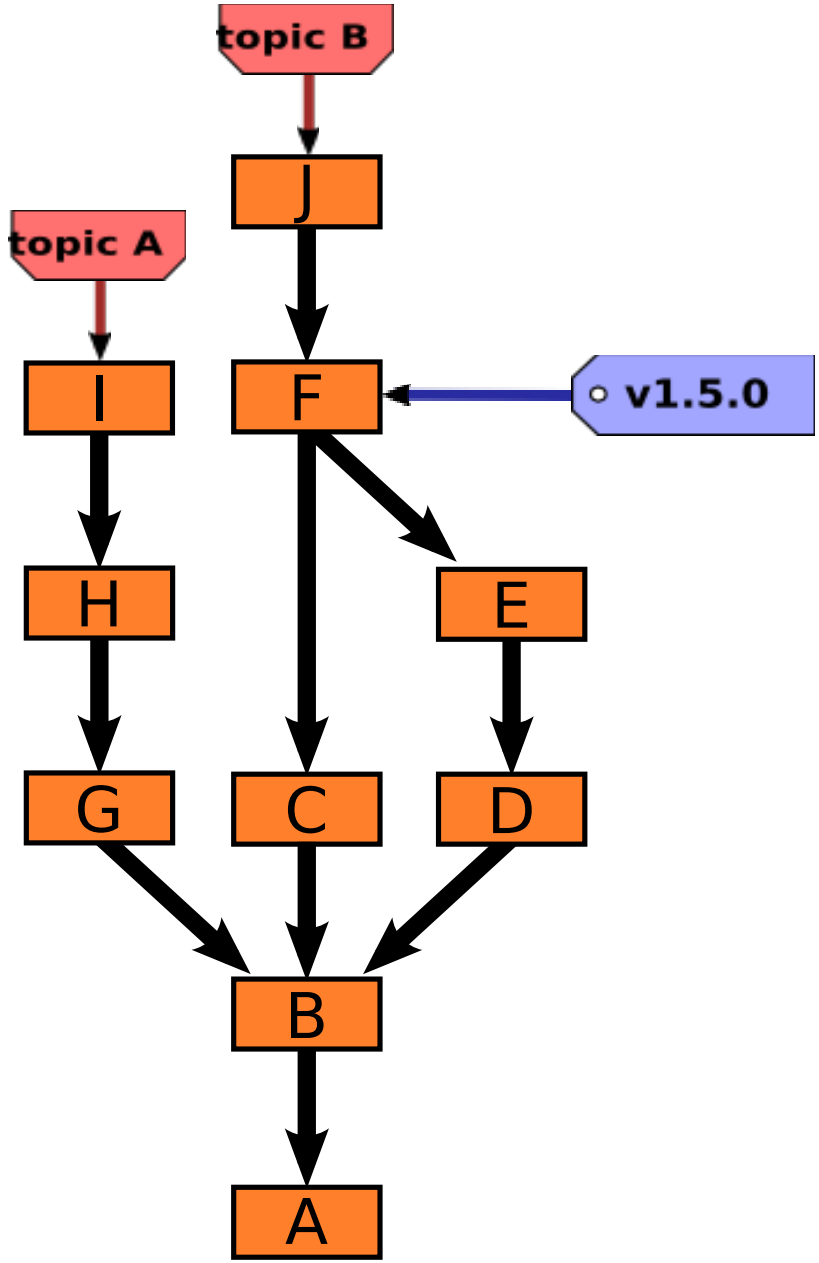
- tags
  - usually don't change



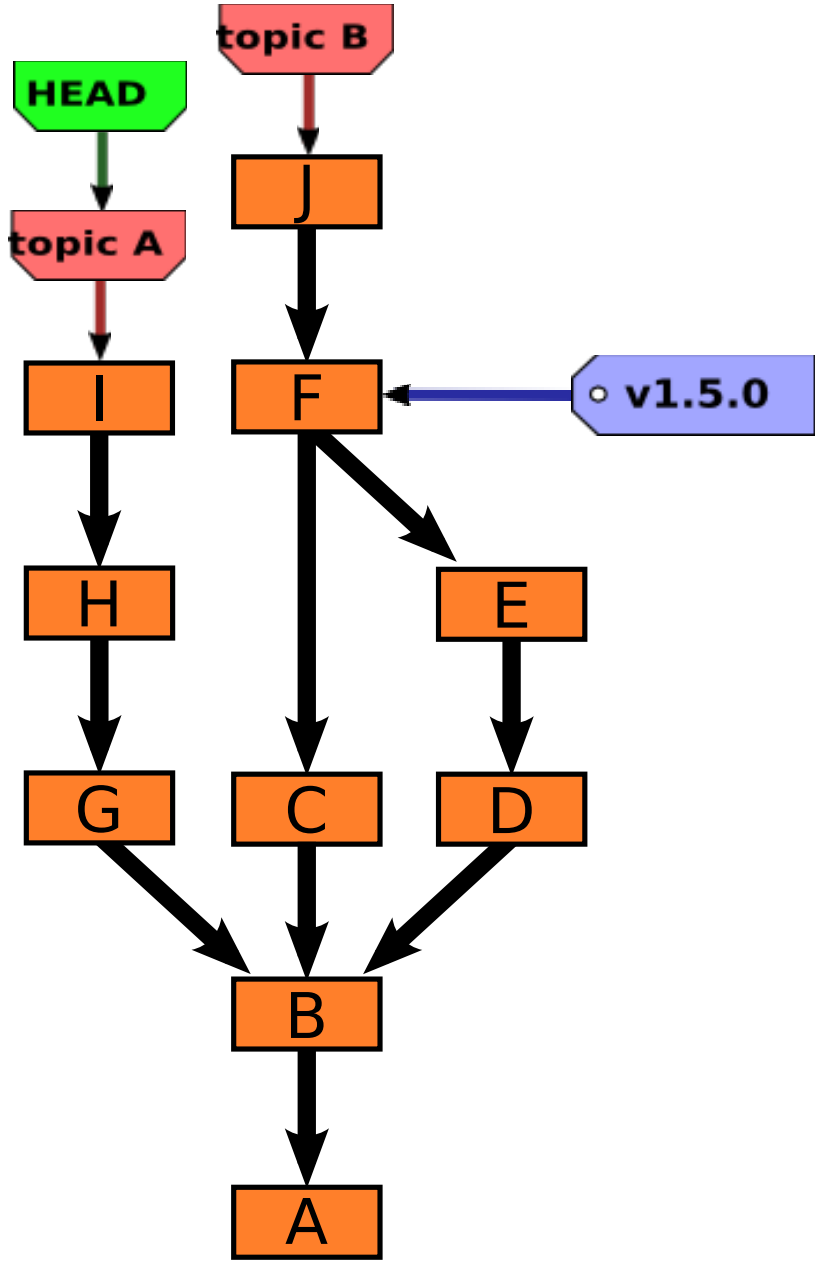
- tags
  - usually don't change
- branches



- tags
  - usually don't change
- branches

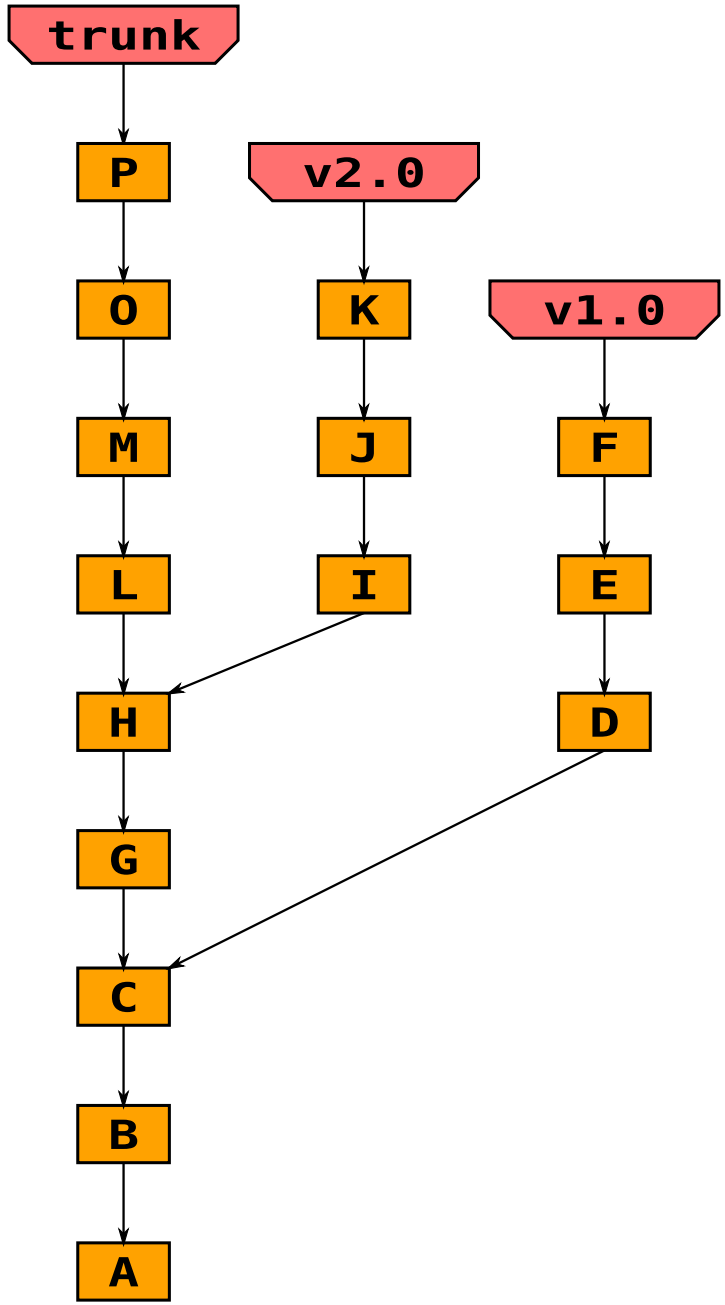


- tags
  - usually don't change
- branches
- current checkout
  - HEAD
  - link to branch,
    we're currently working on



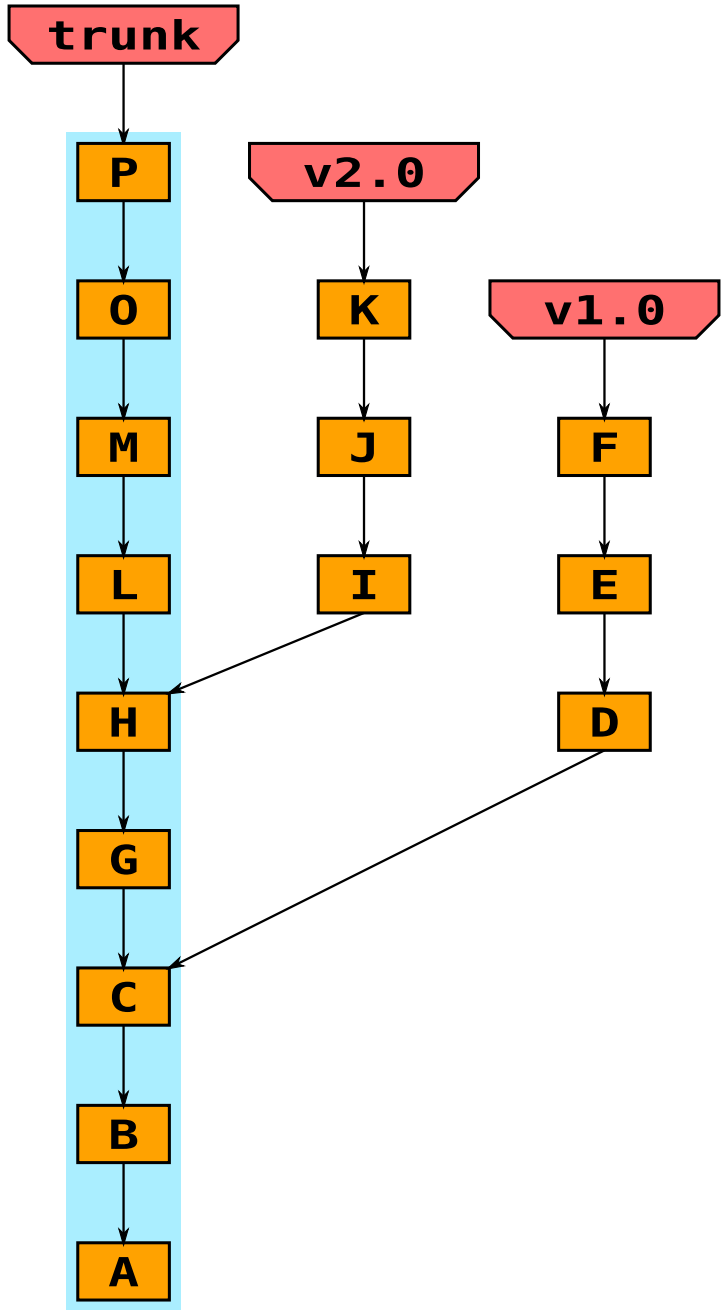
- Workflow stands for progression of work with the repository. Events typically repeat.
- The base flow is to make changes in working directory, add these changes to staging area and commit. Repeat.
- Usual:
  - Traditional workflow
  - Topic branches

Traditional workflow



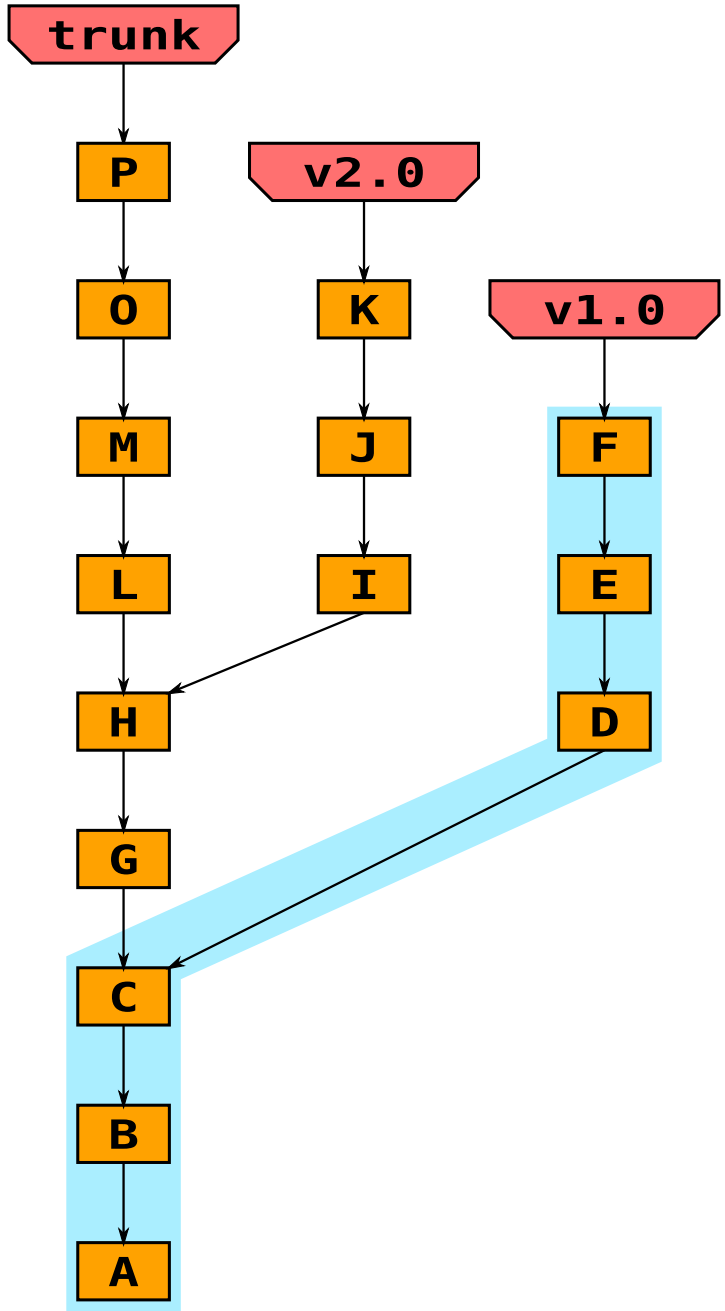
Traditional workflow

Main development branch

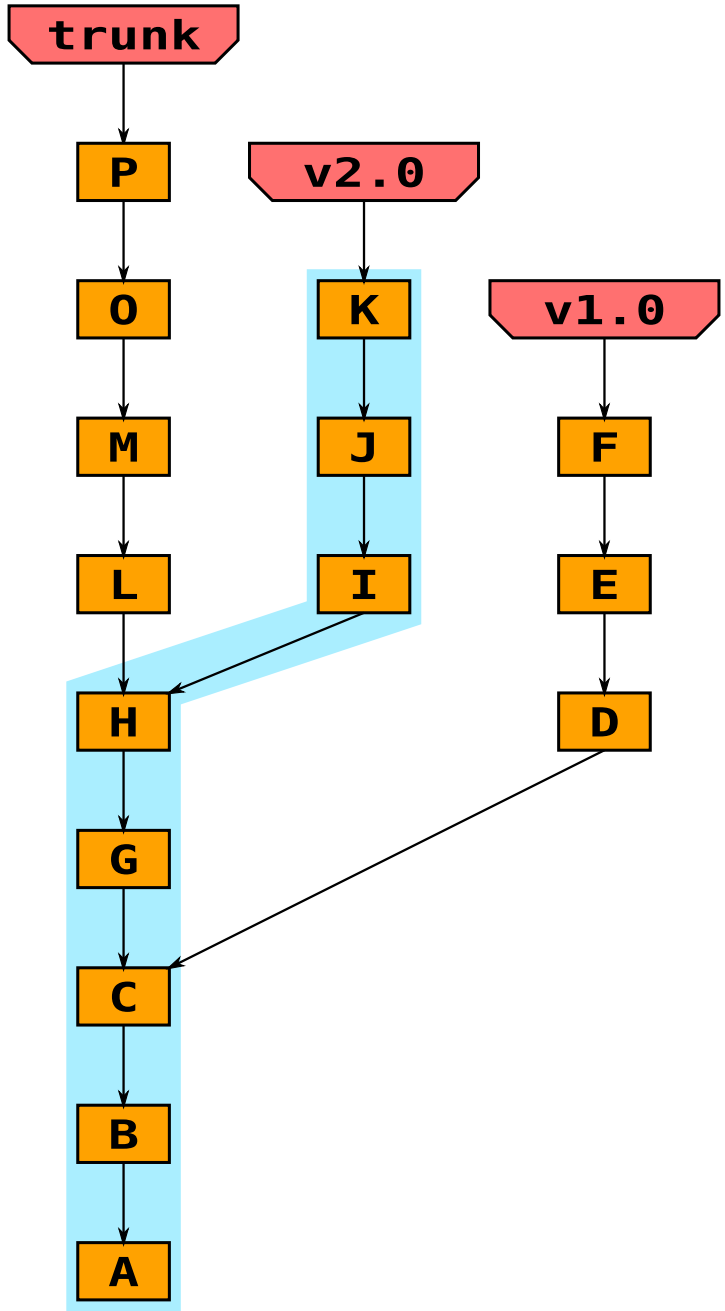


Traditional workflow

Historical versions

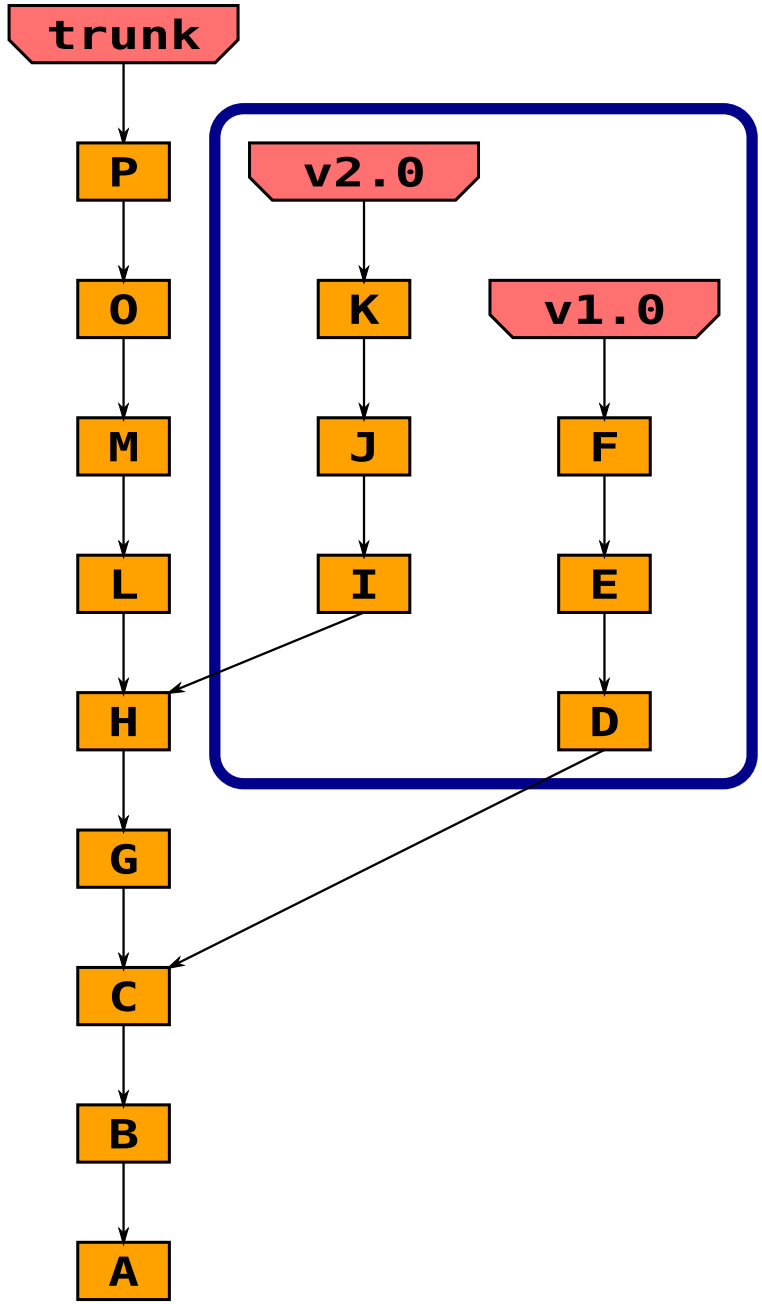


Historical versions



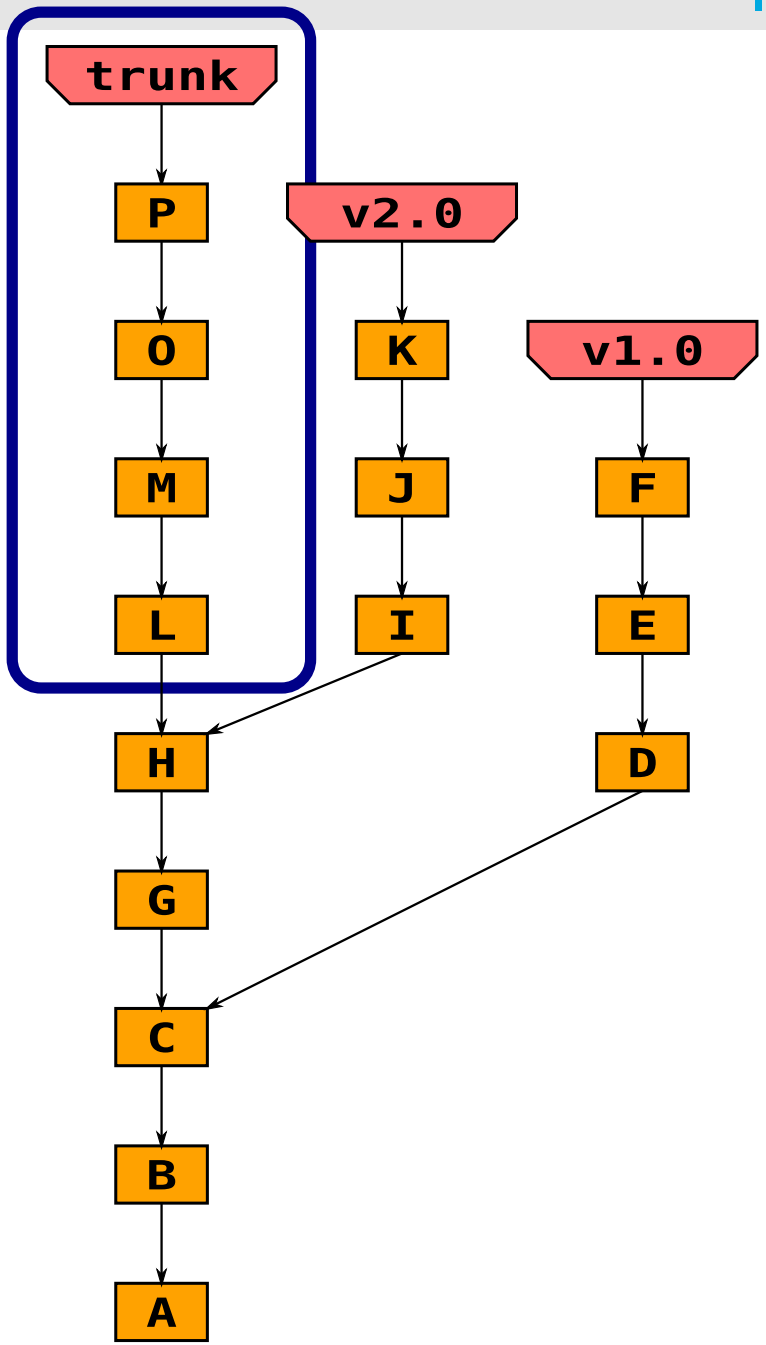
Traditional workflow

Only bugfixes



Traditional workflow

New functionality and bugfixes



Topic branches

- One branch for each topic

Topic branches

- One branch for each topic
- Frequent commits

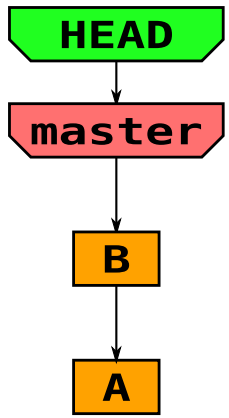
Topic branches

- One branch for each topic
- Frequent commits
- Short lifespan

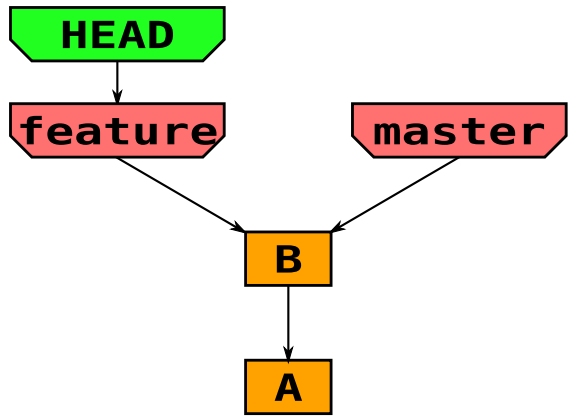
Topic branches

- One branch for each topic
- Frequent commits
- Short lifespan
- Finished topic merged to master

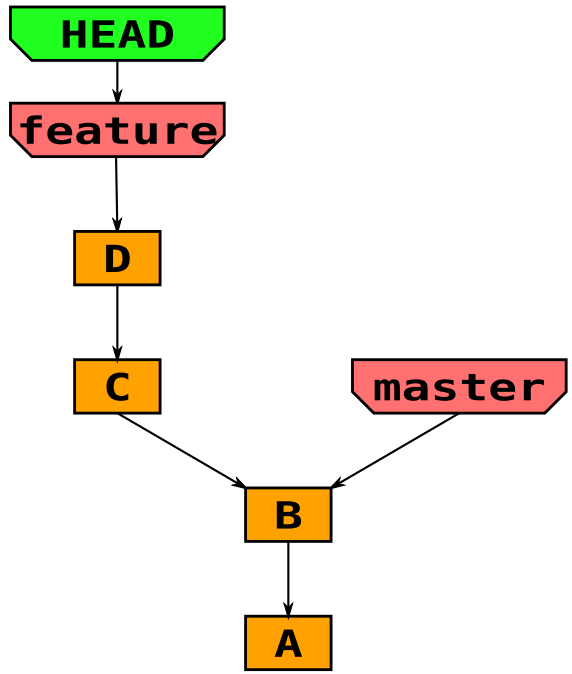
Current repository



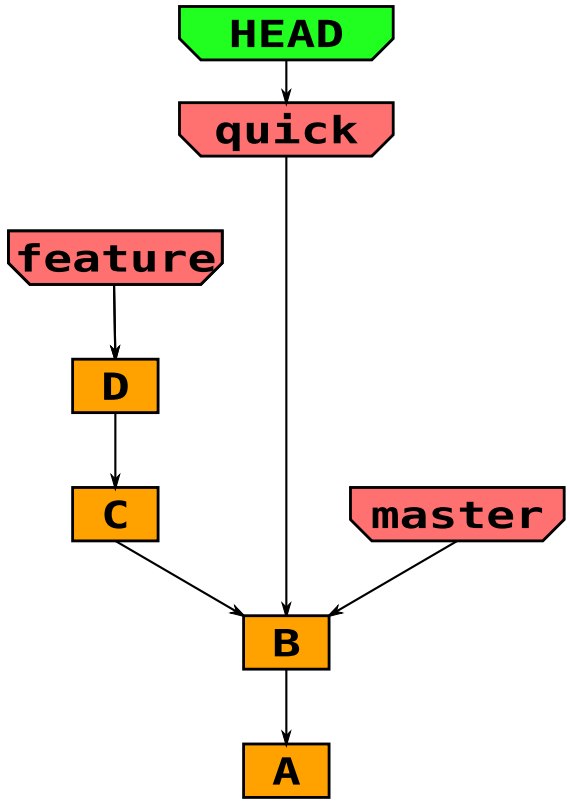
Create branch for feature.



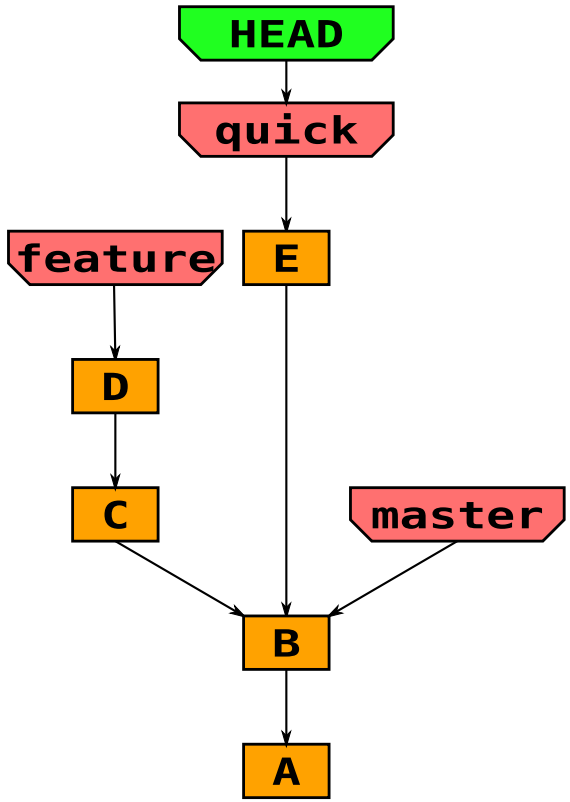
Develop feature.



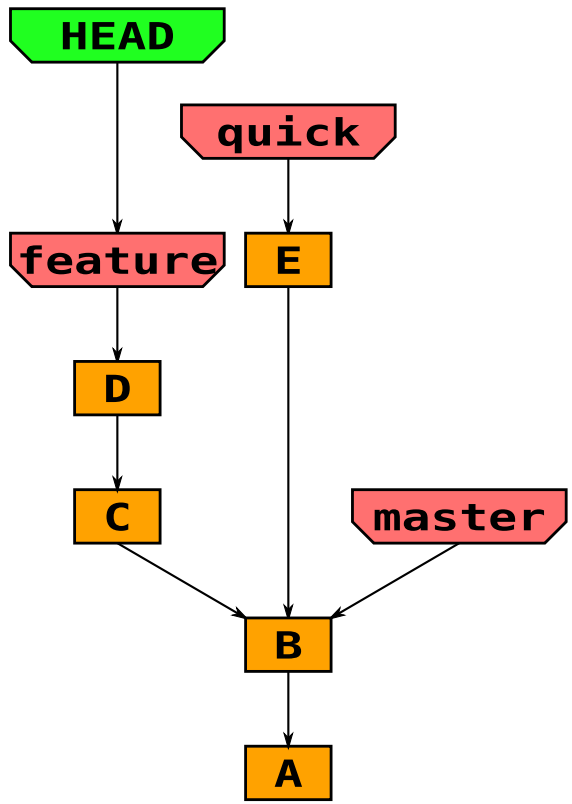
We've identified a bug,
that hinders the development,
and so we fix it now.



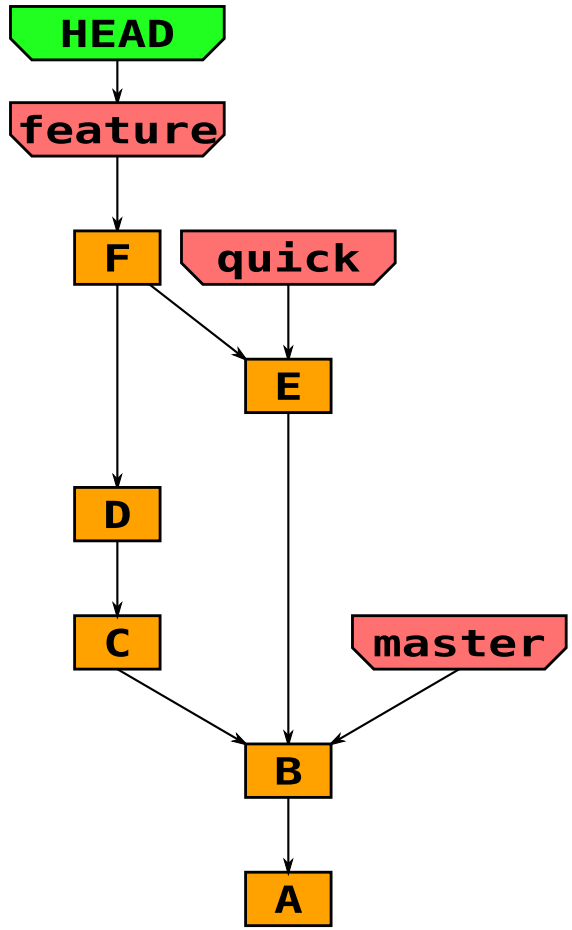
Quick bugfix.



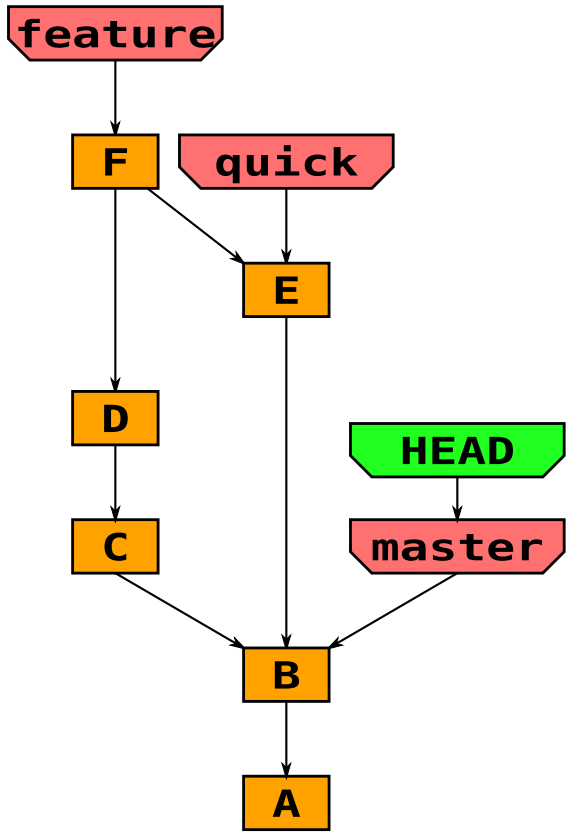
Back to developing feature.



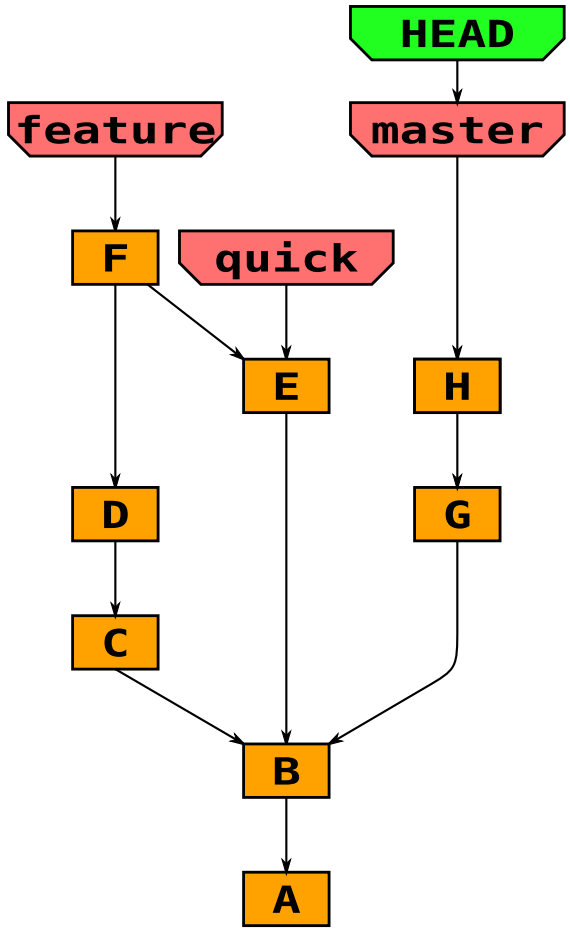
Merge bugfix.



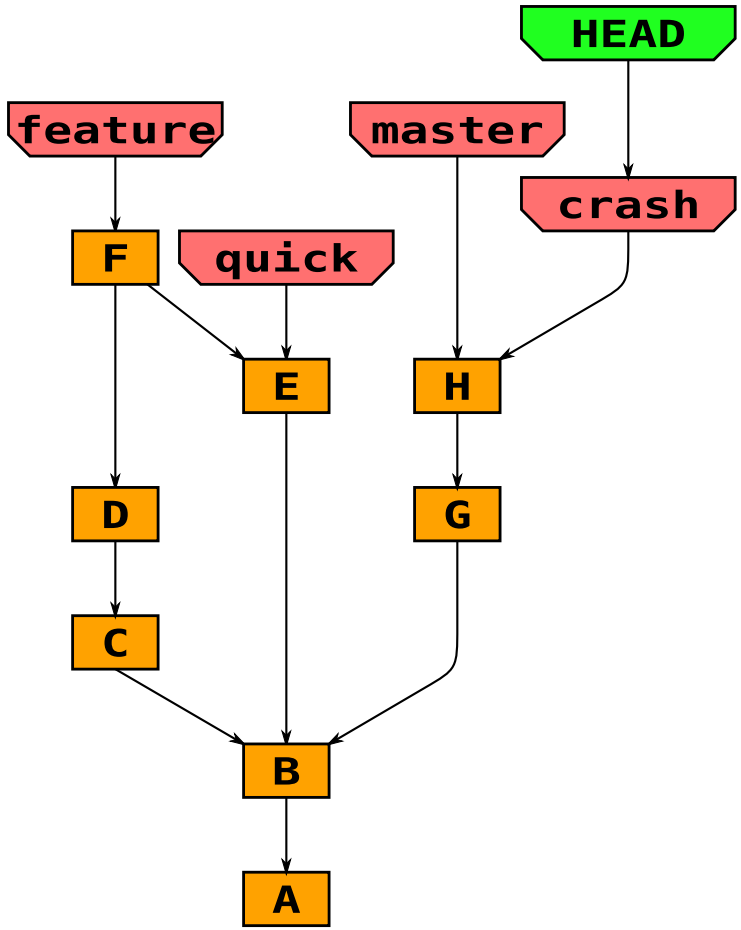
Upstream crashes!



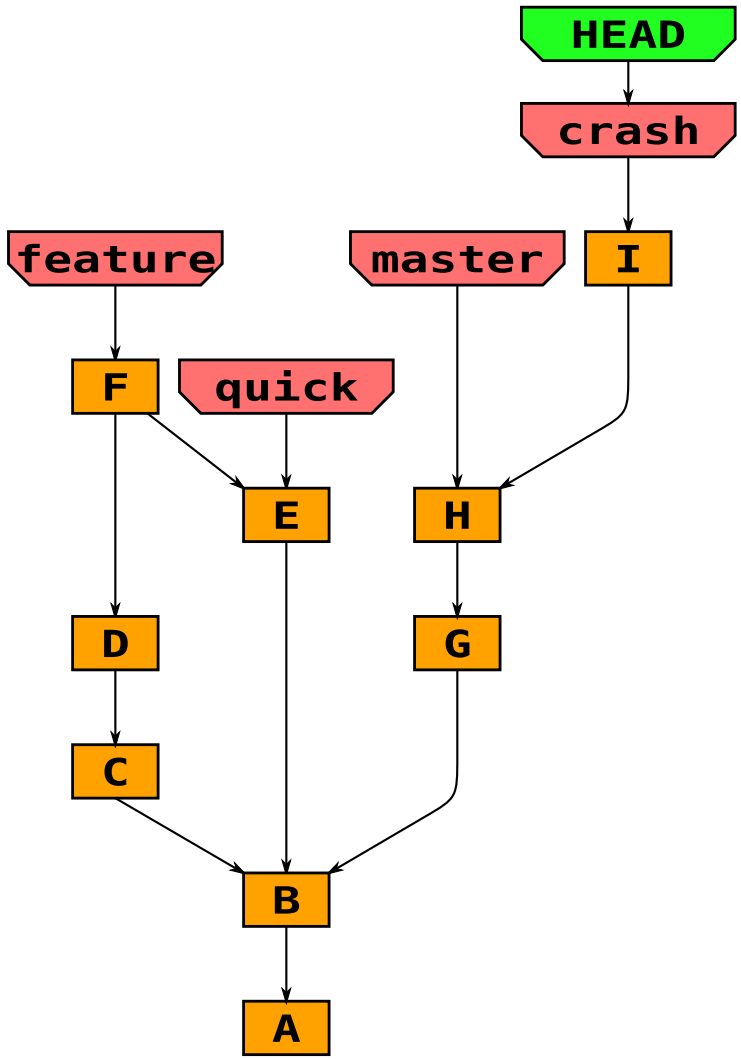
First we update master with upstream version.



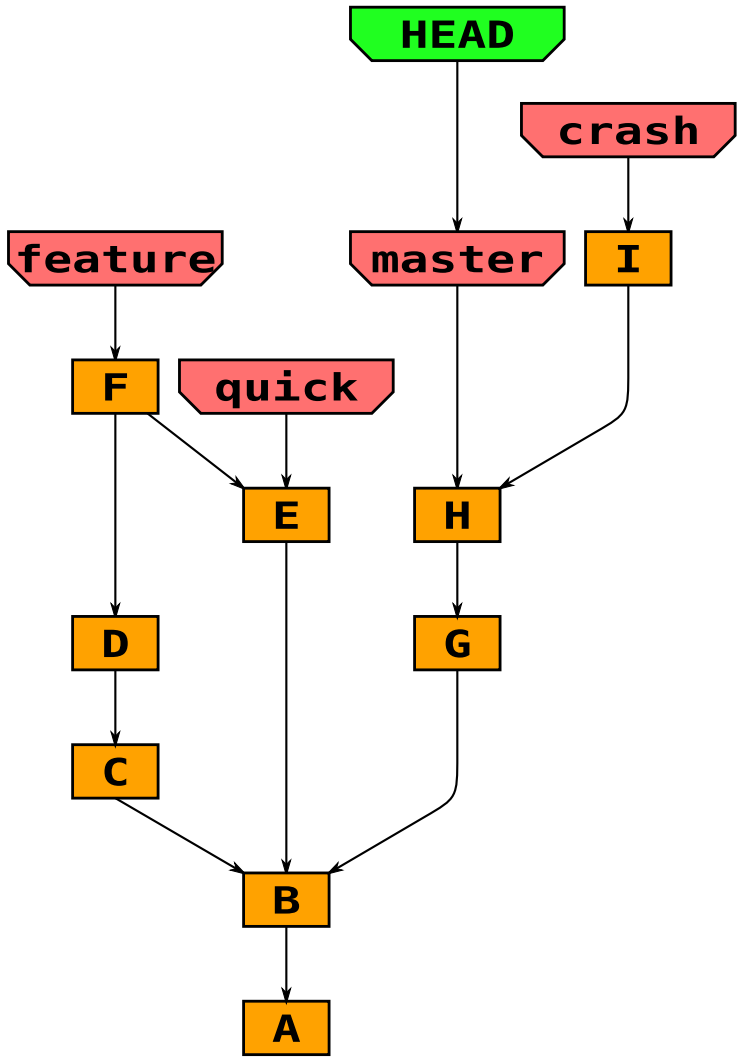
We create a branch for the error crash.



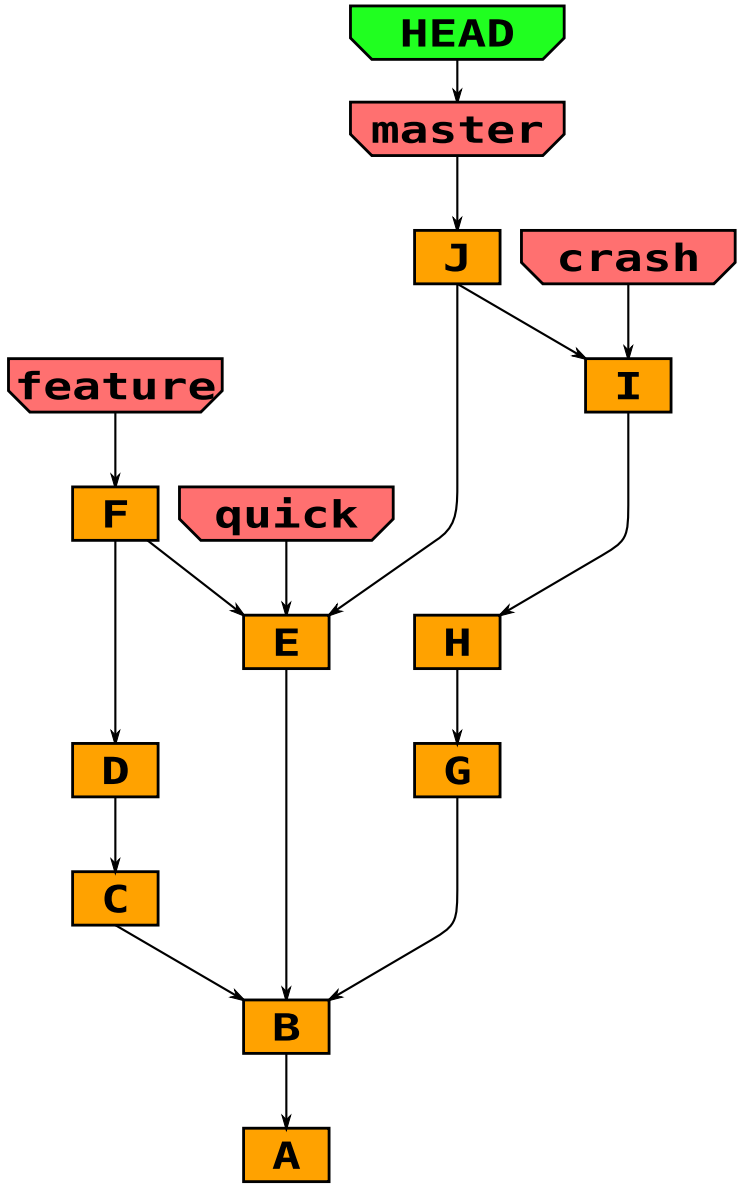
Fix the error crash.



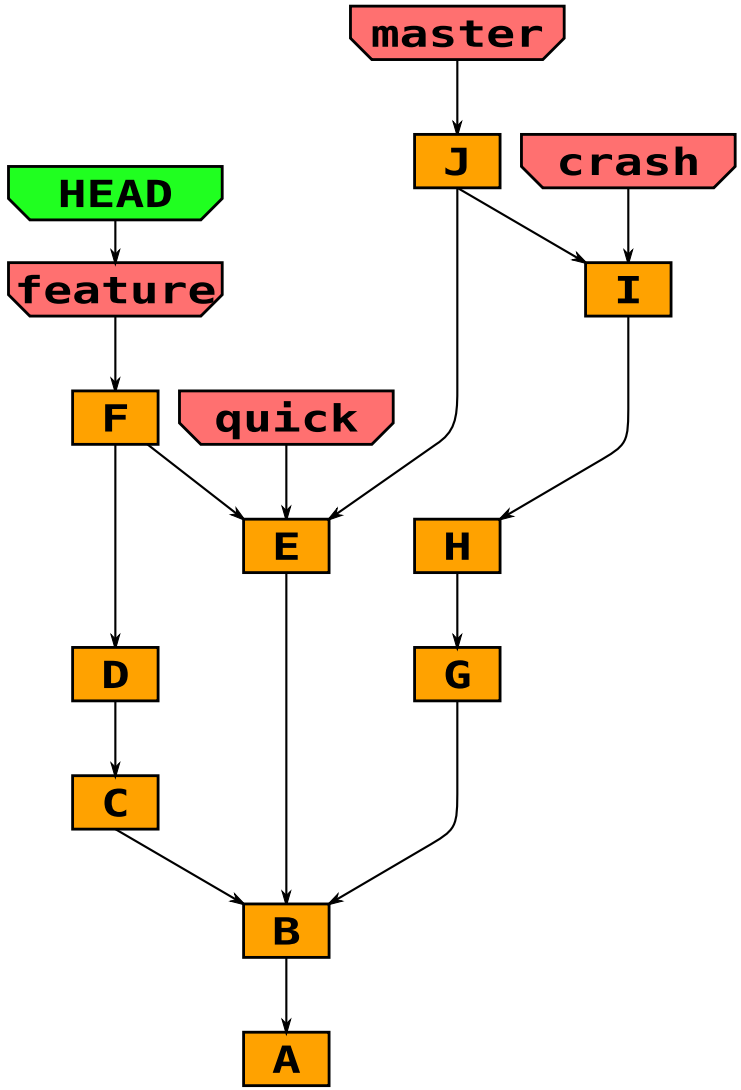
And now we will integrate.



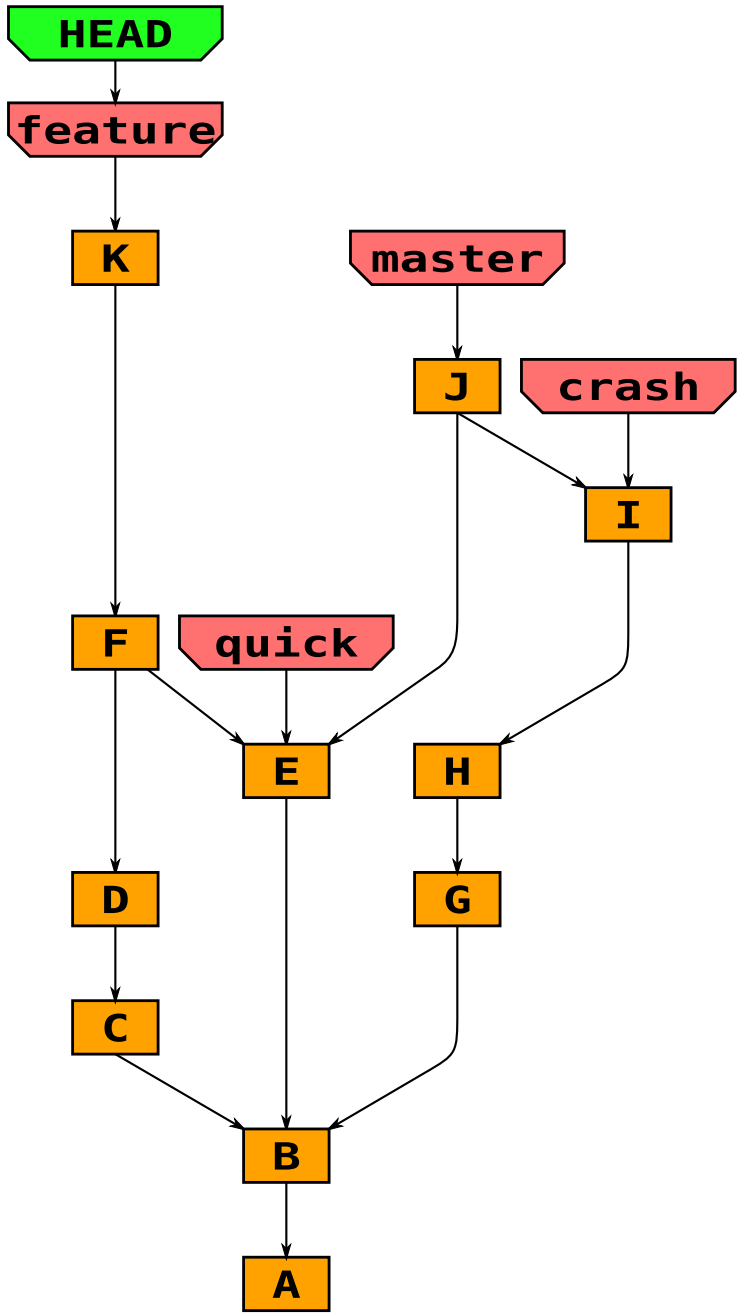
Integrate fixes of both errors quick and crash to master.



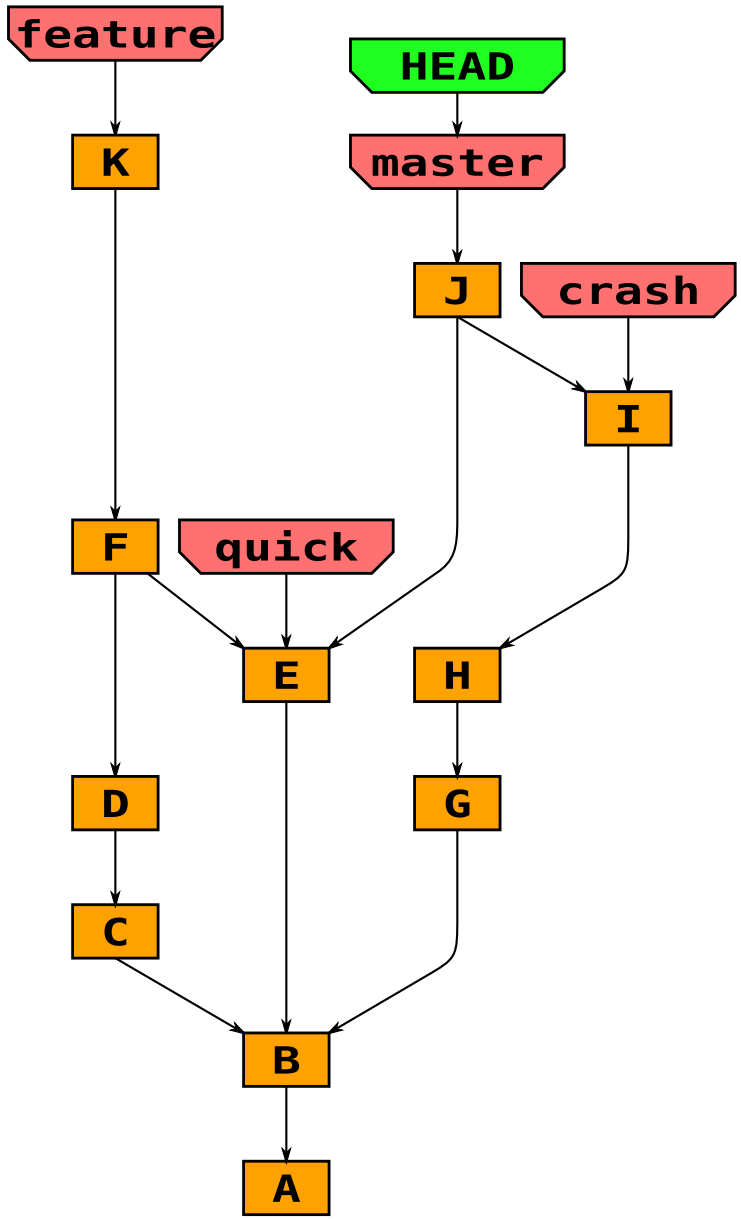
And back to developing our feature.



Finish the feature.

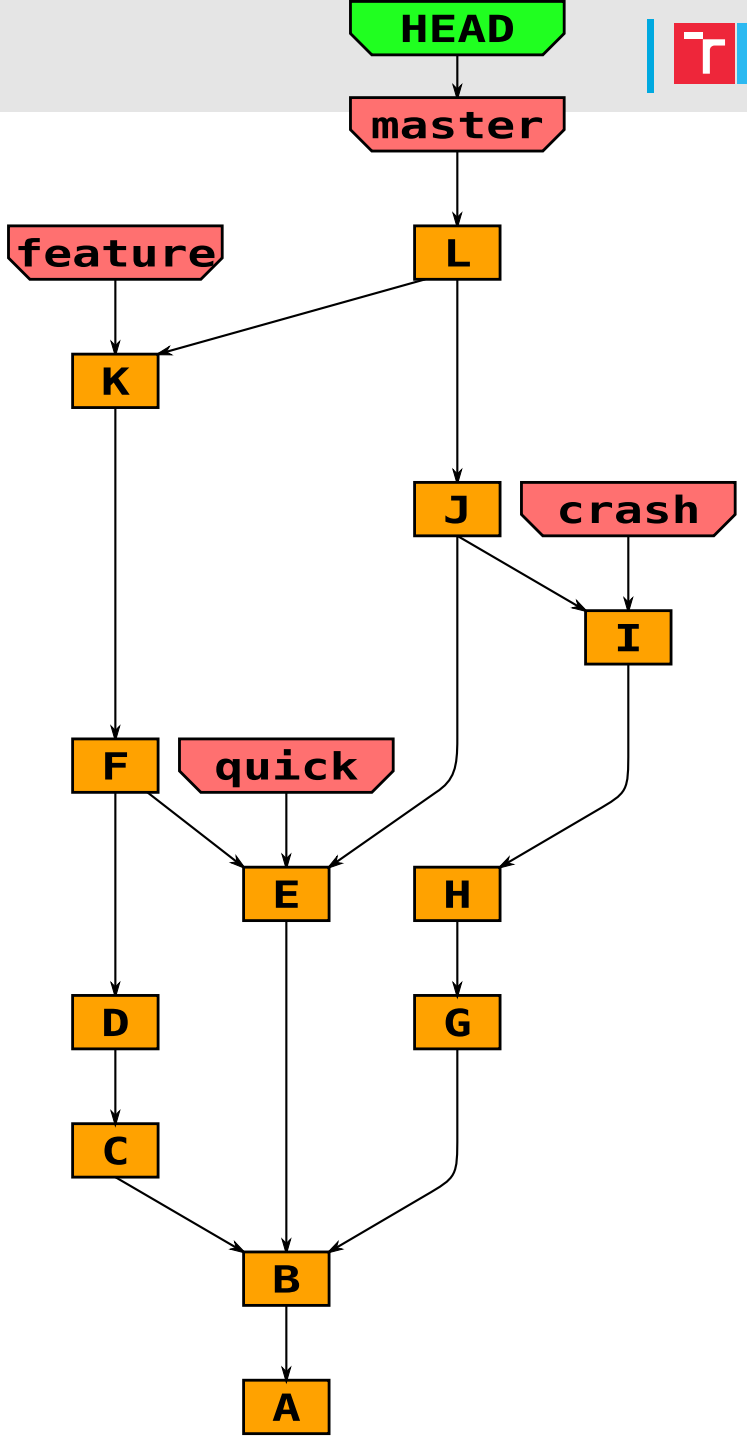


Switch to master.

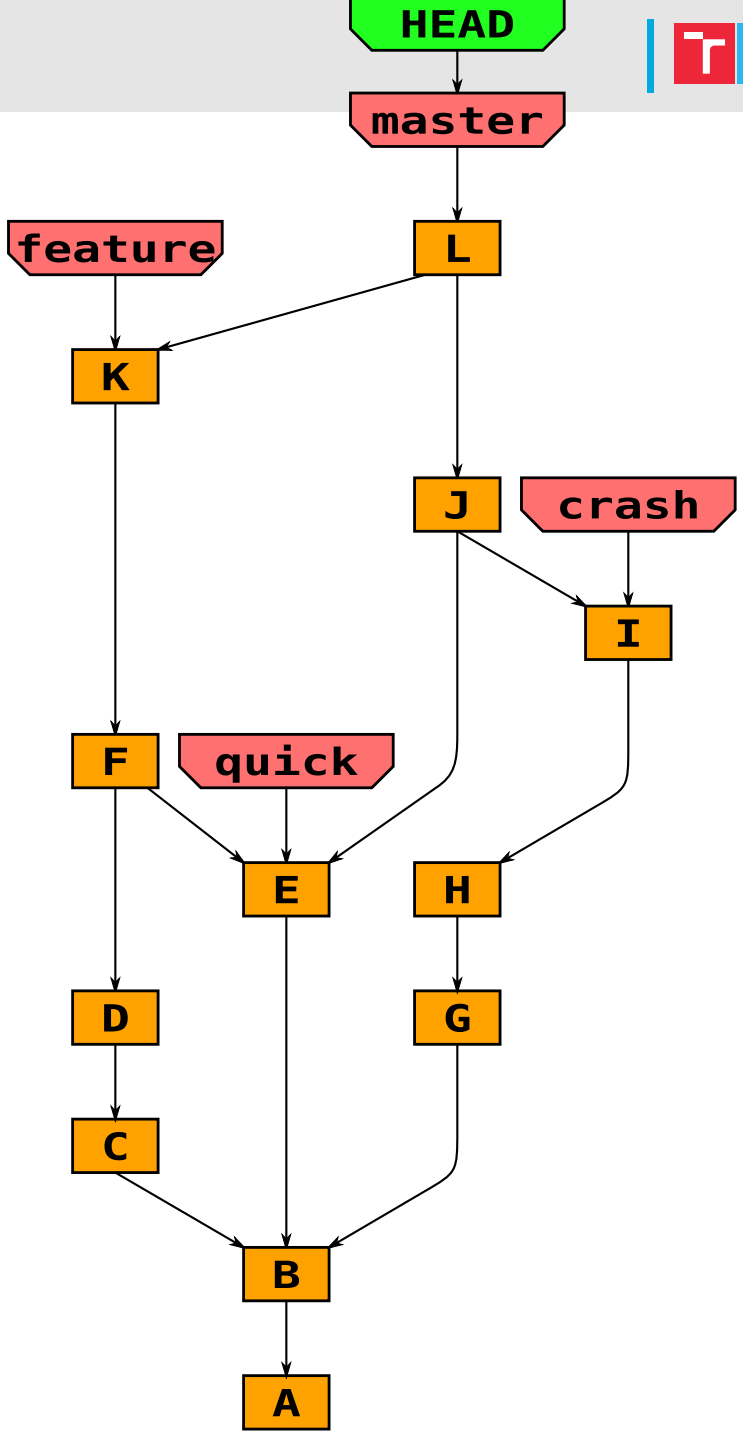


Topic branches

Merge feature to master.

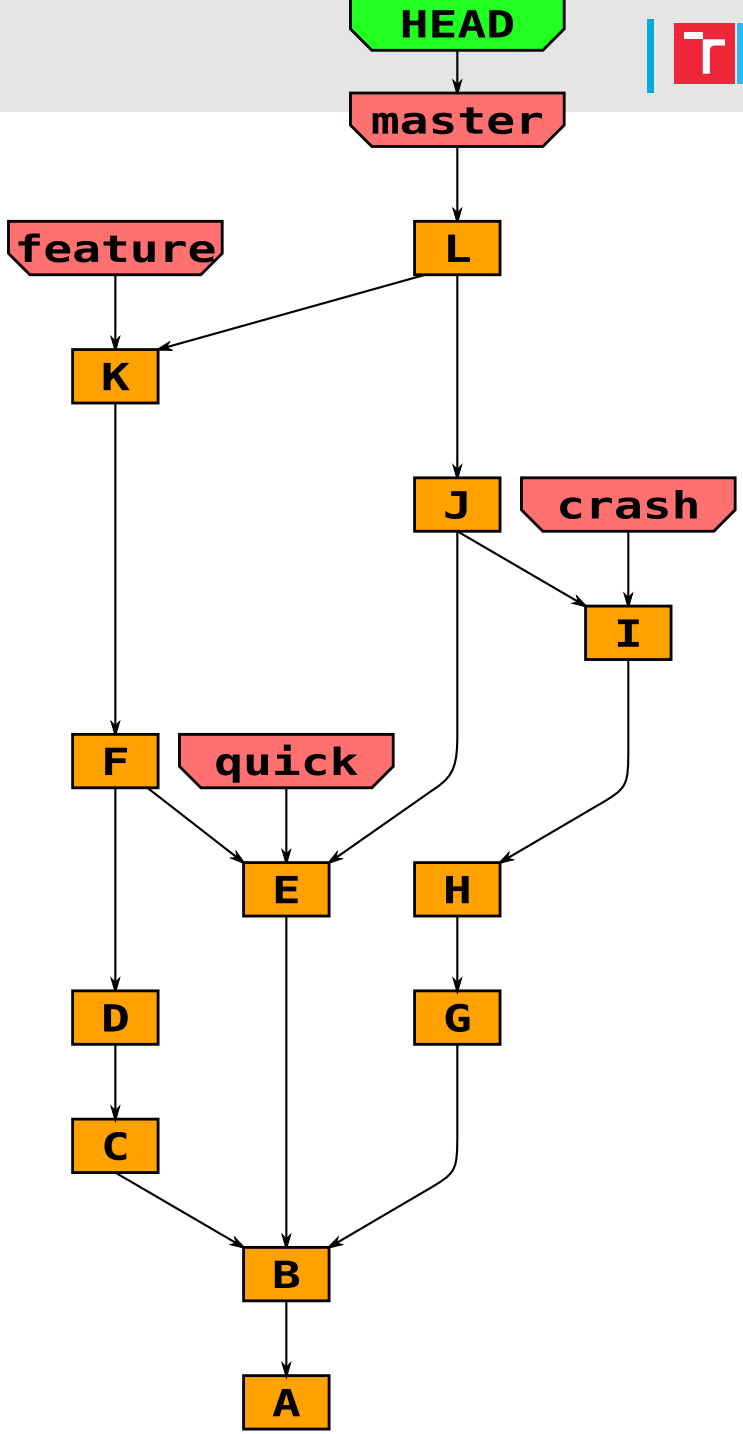


Topic branches



Topic branches

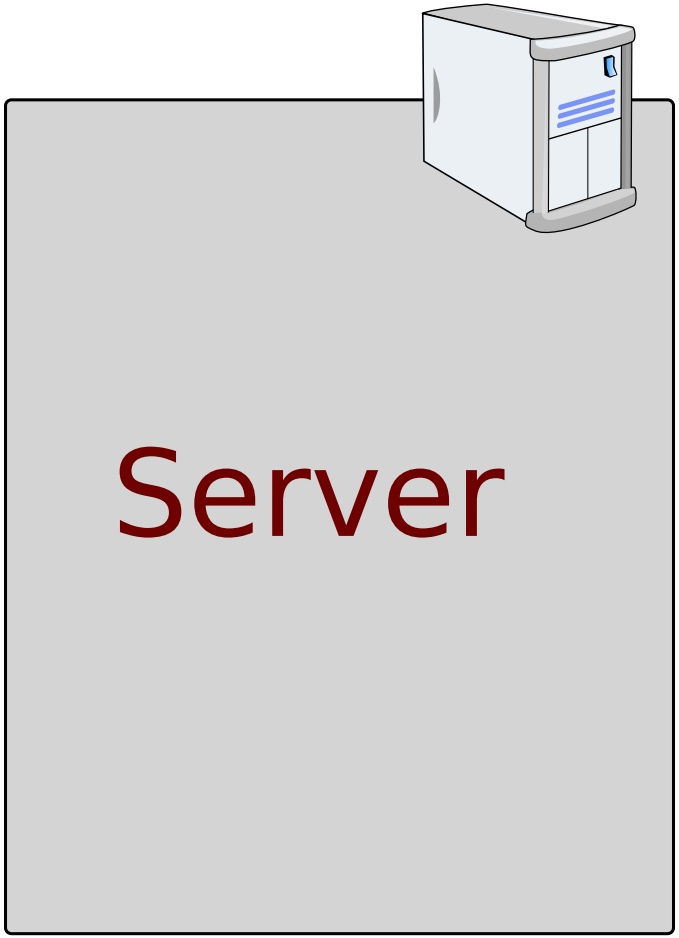
- Clean
- Revision (before merging)
- Merge can be postponed



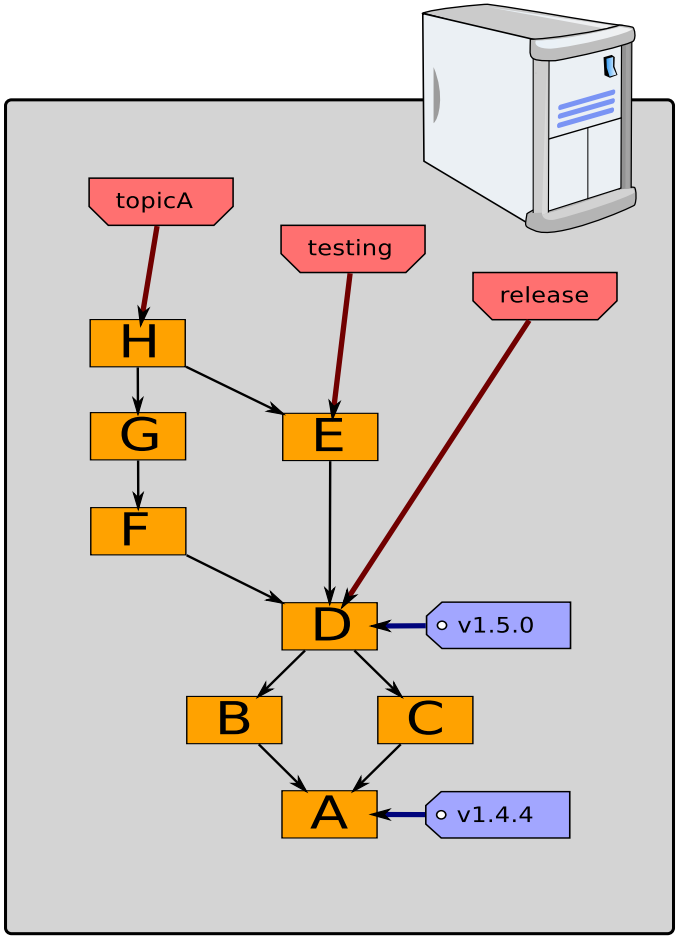
Version control systems

- Centralized
- Distributed

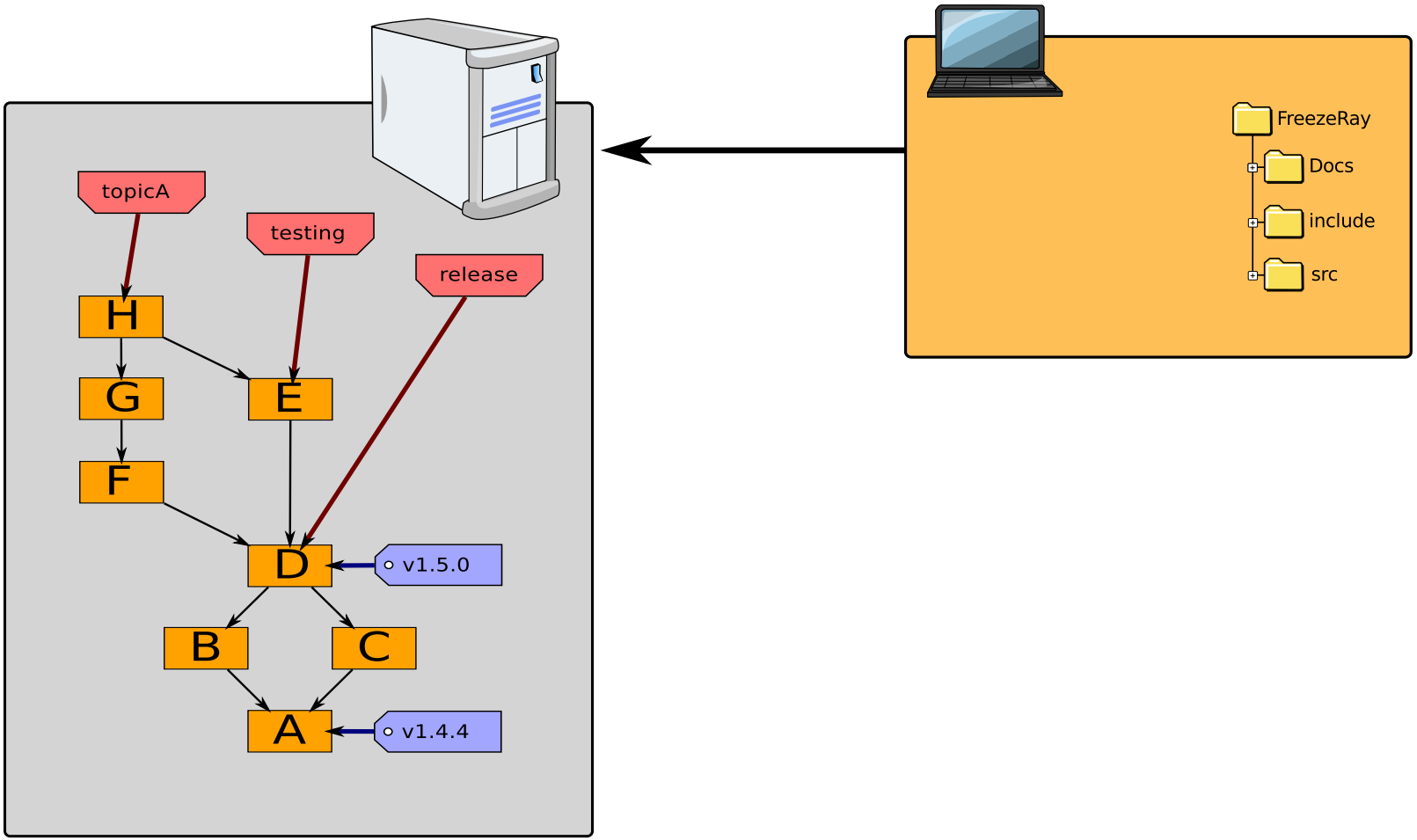
Centralized version control



Centralized version control

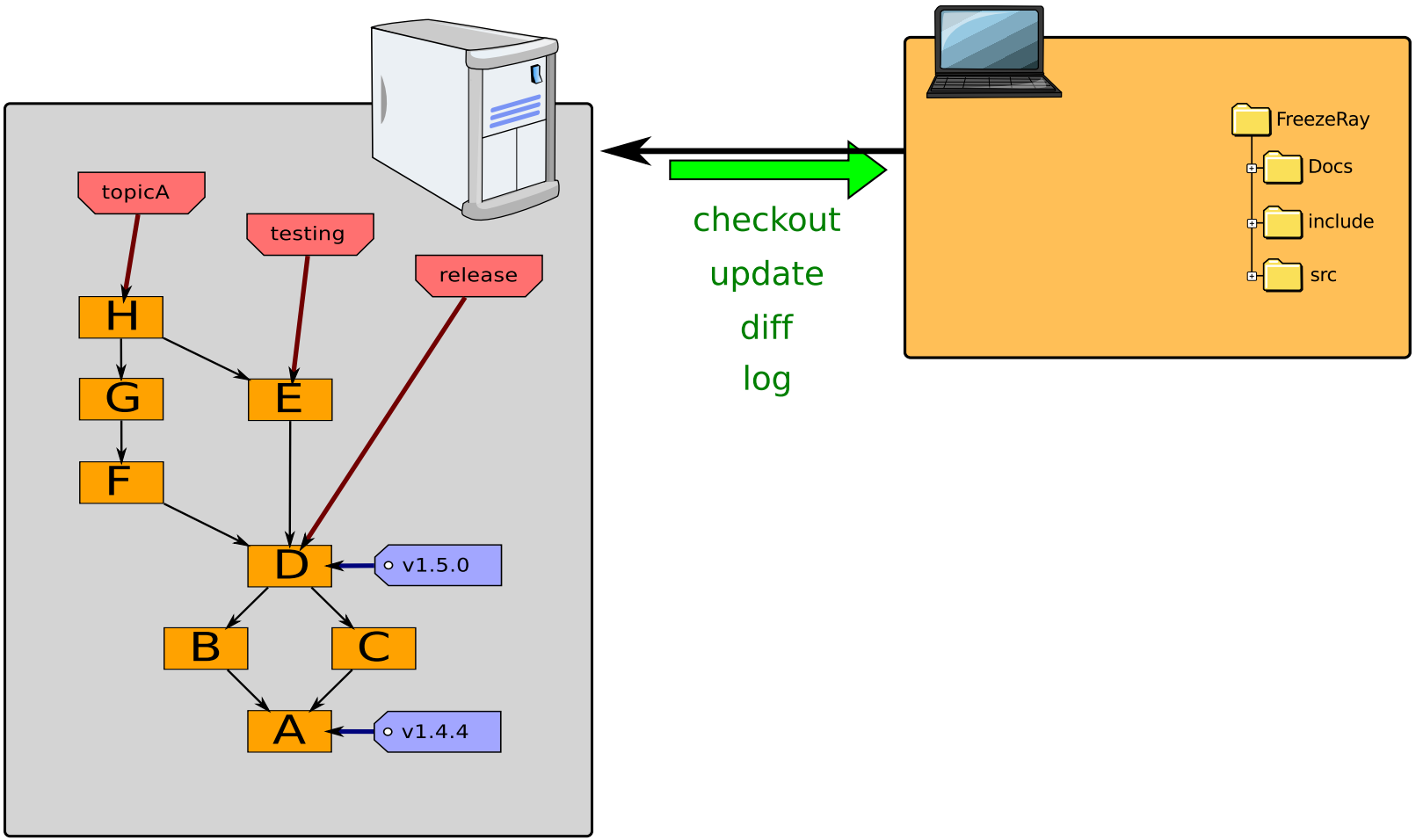


Centralized version control



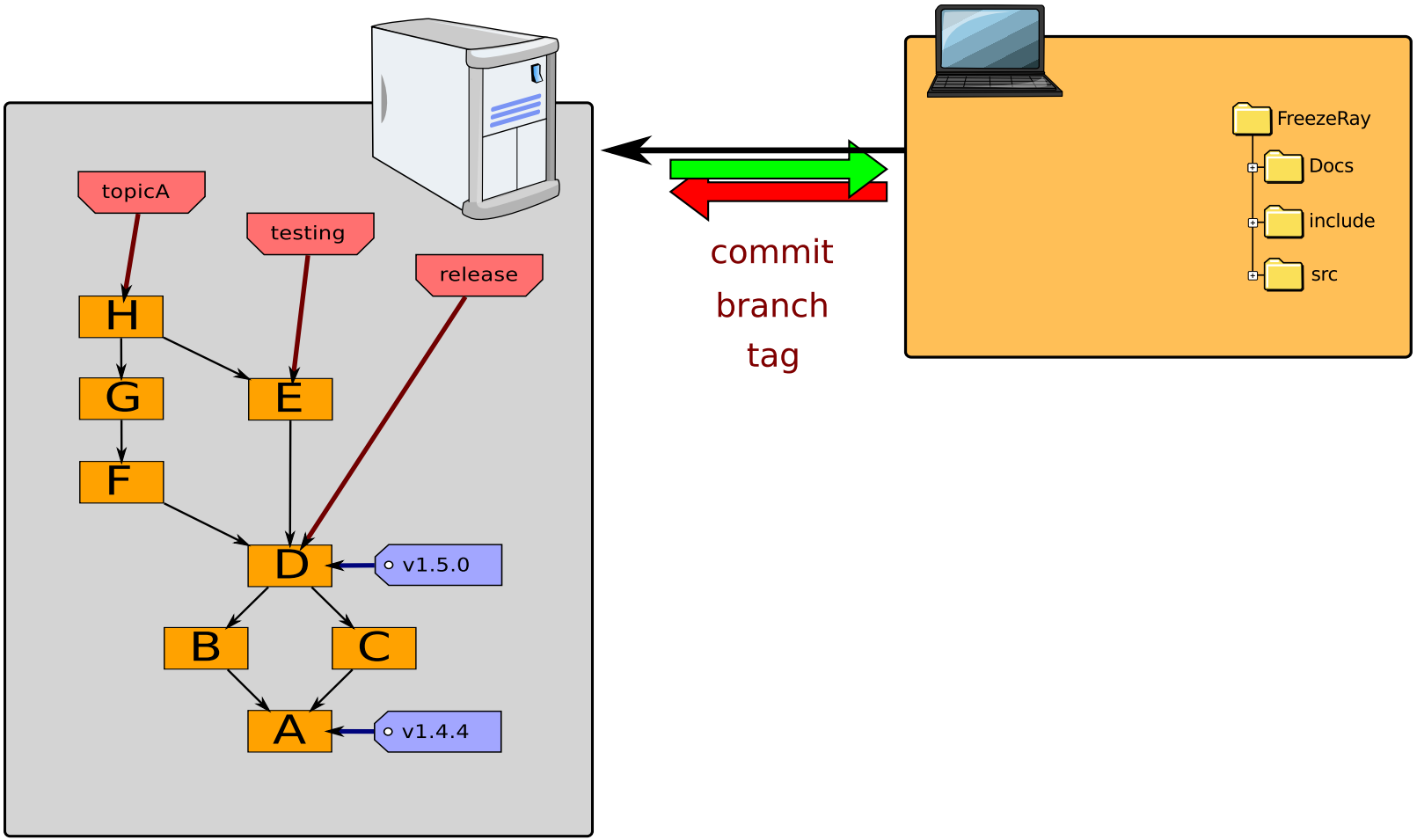
- Client has only working tree

Centralized version control



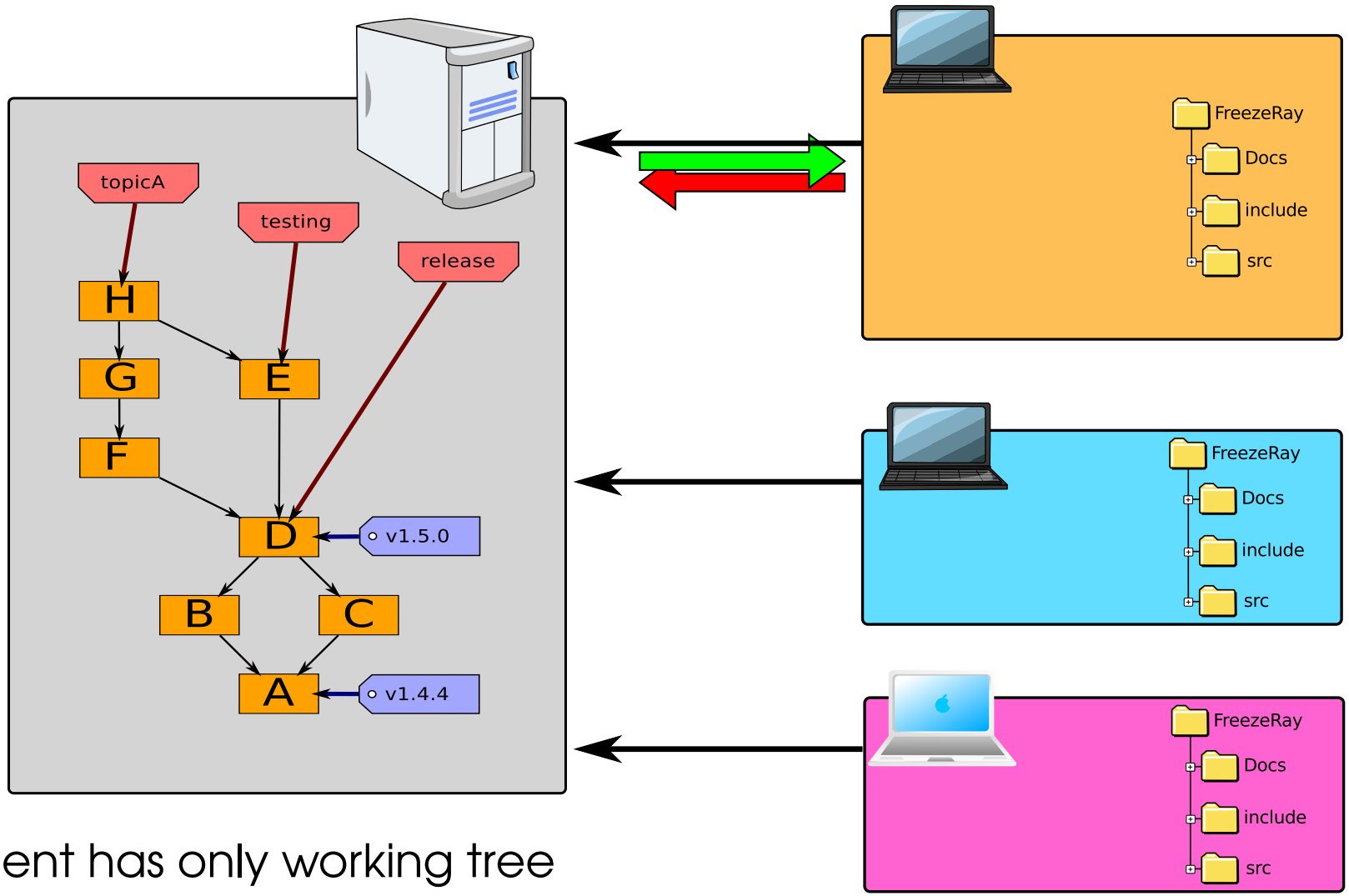
- Client has only working tree
- All operations need server

Centralized version control



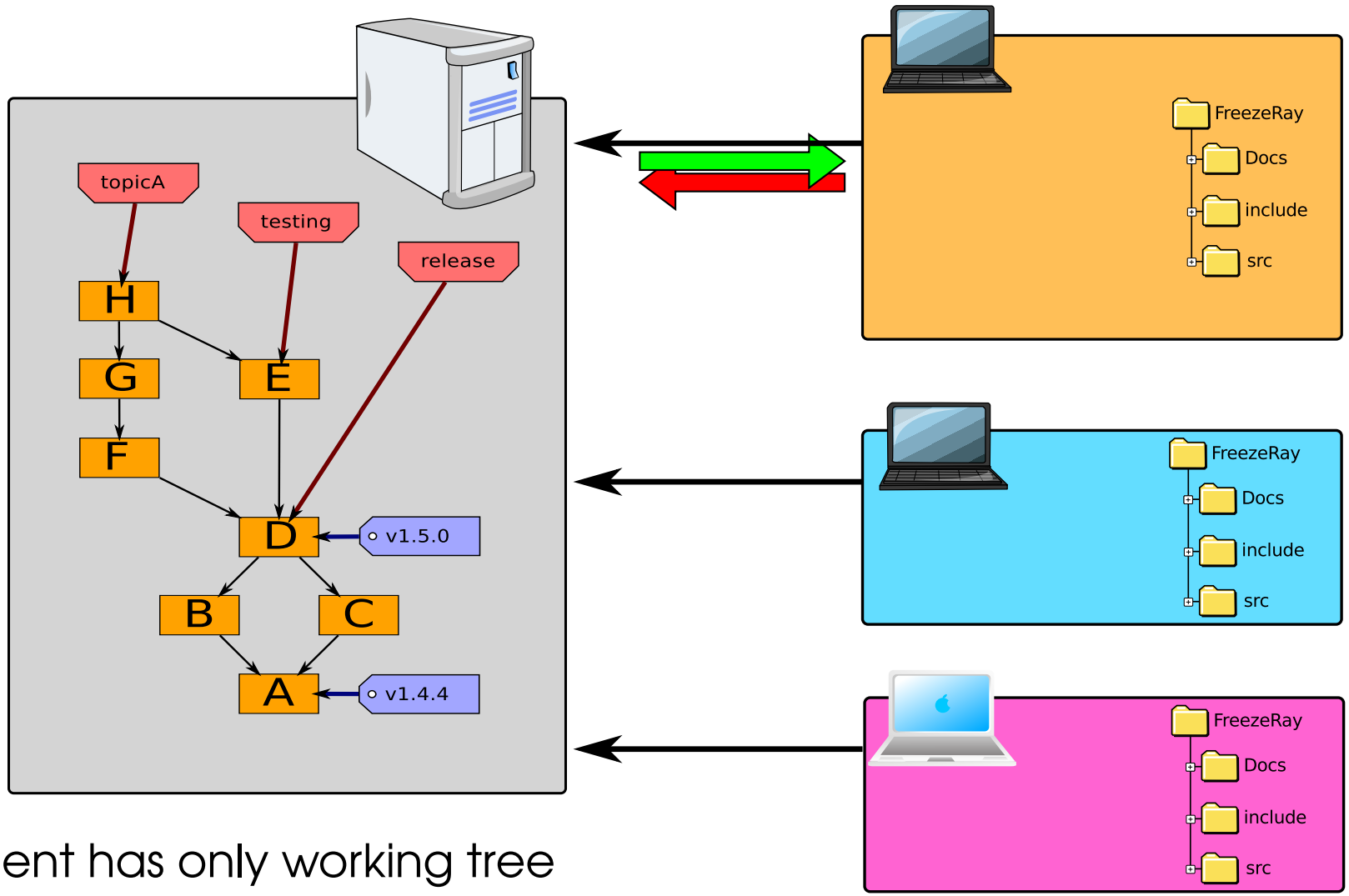
- Client has only working tree
- All operations need server

Centralized version control



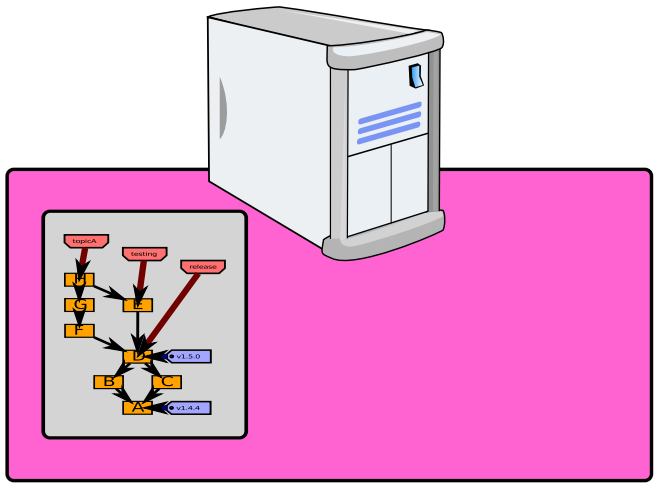
- Client has only working tree
- All operations need server

Centralized version control

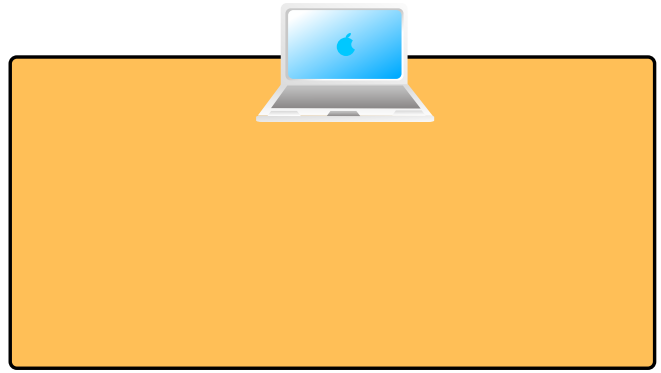


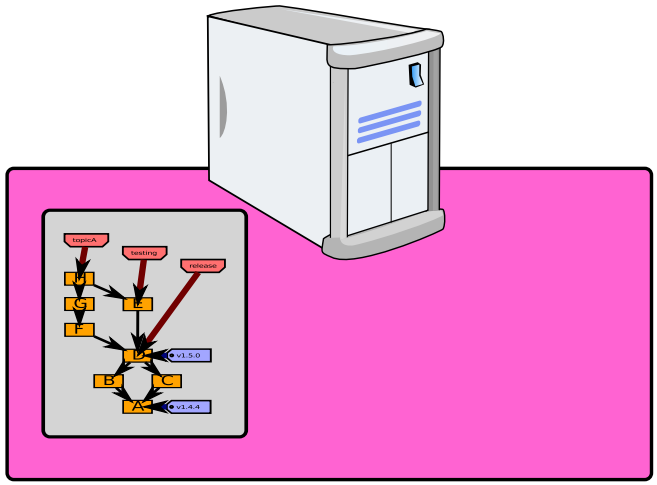
- Client has only working tree
- All operations need server
  - Single point of failure
  - Bottleneck

Decentralized version control

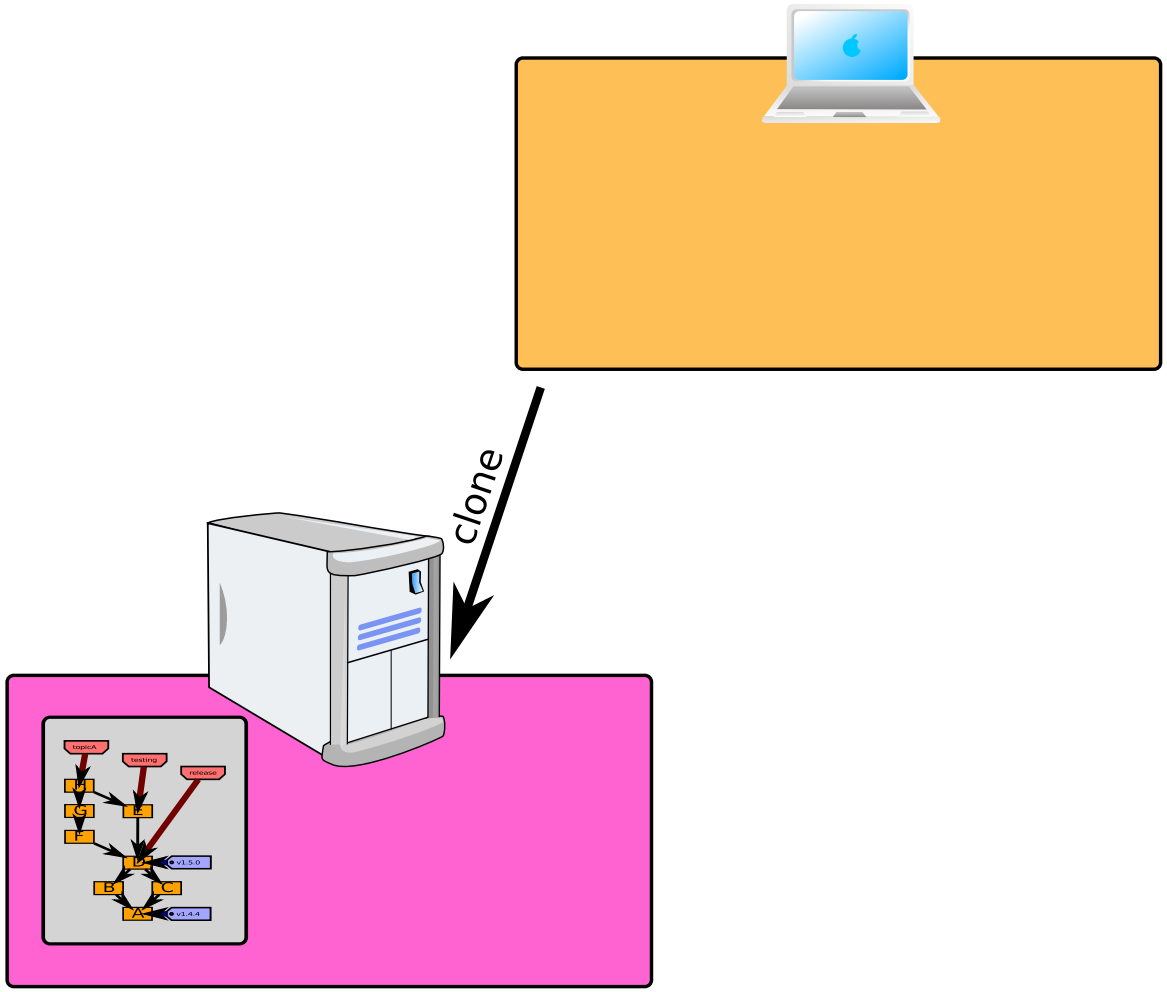


Decentralized version control

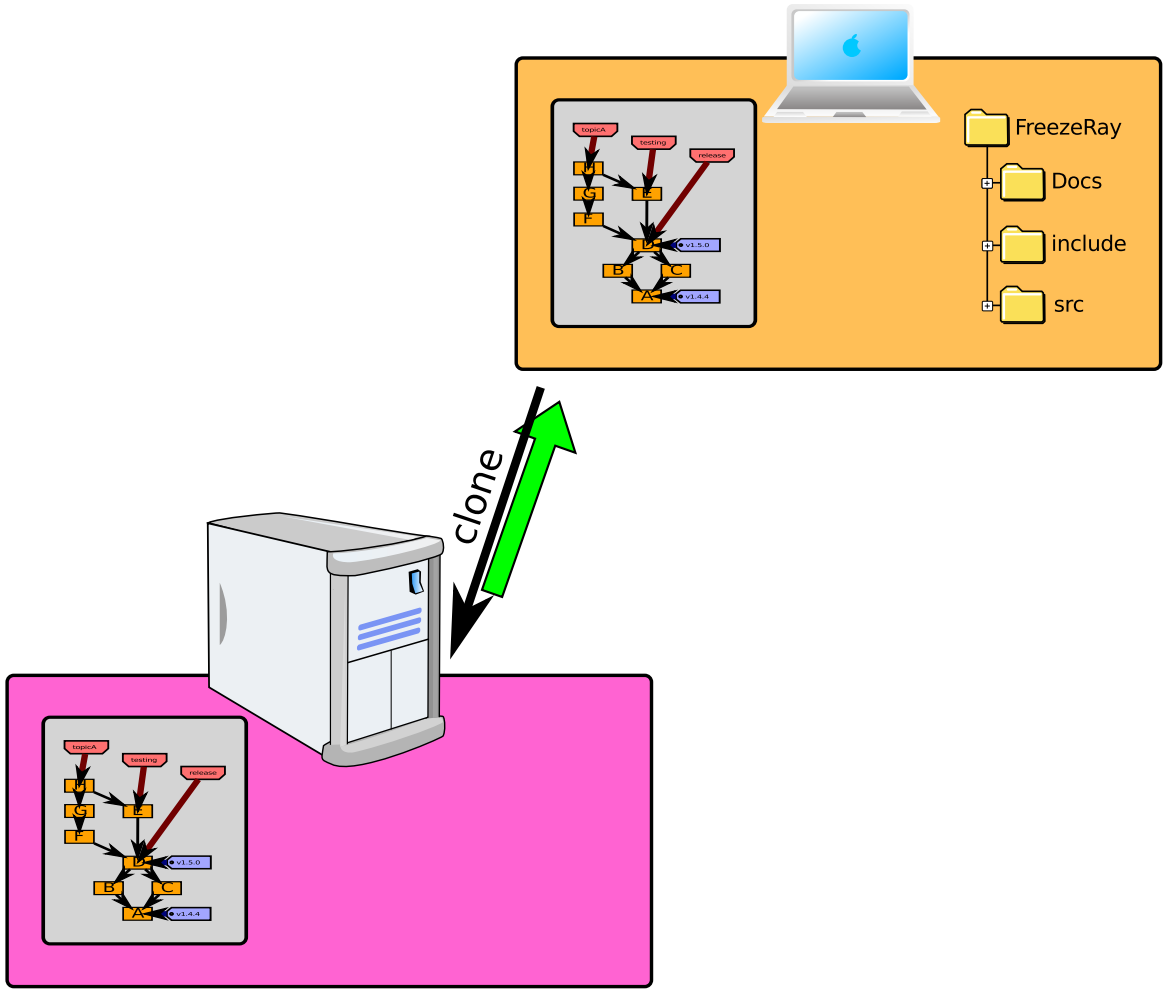




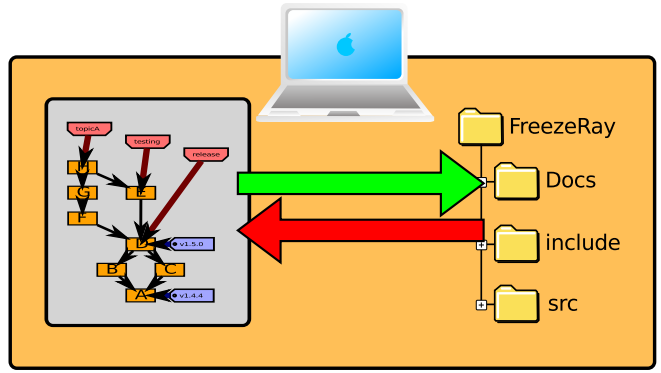
- Client has a clone of repository

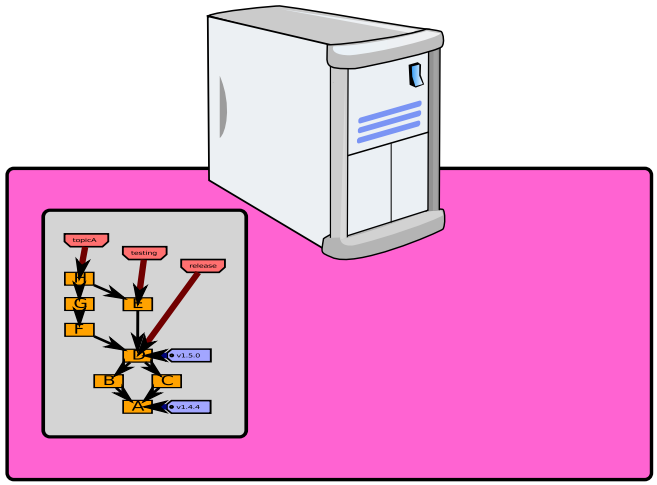


- Client has a clone of repository



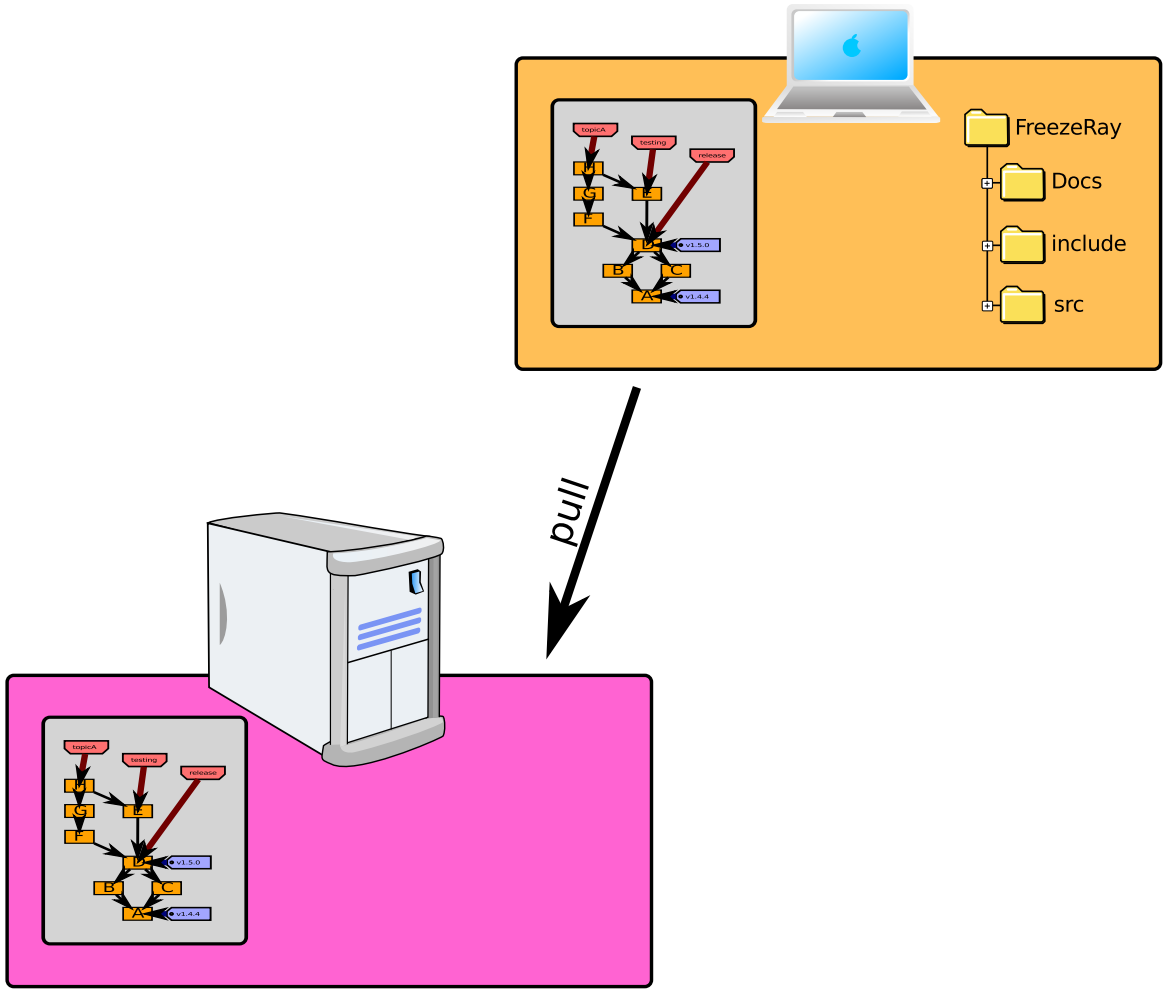
- Client has a clone of repository (operations are local)



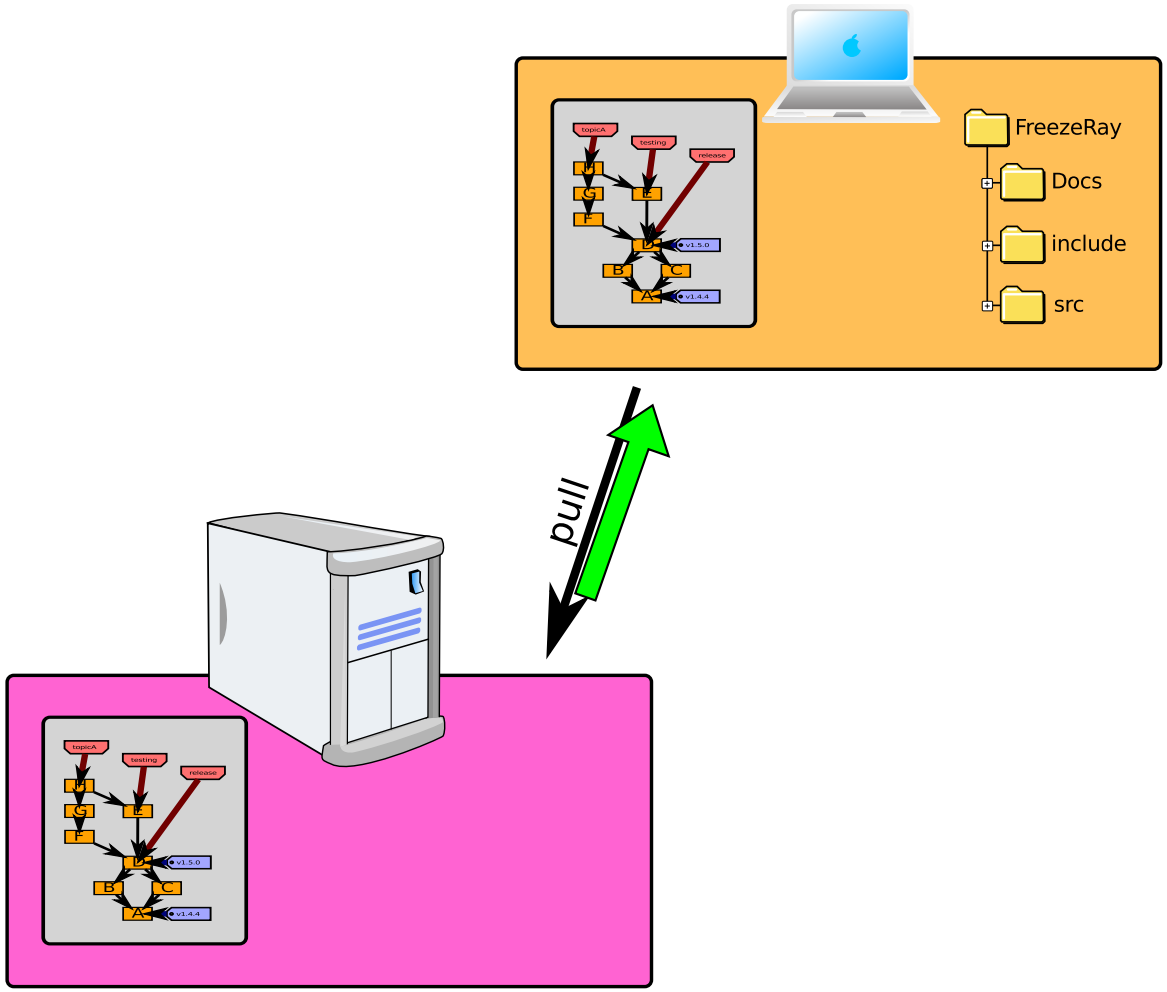


Decentralized version control

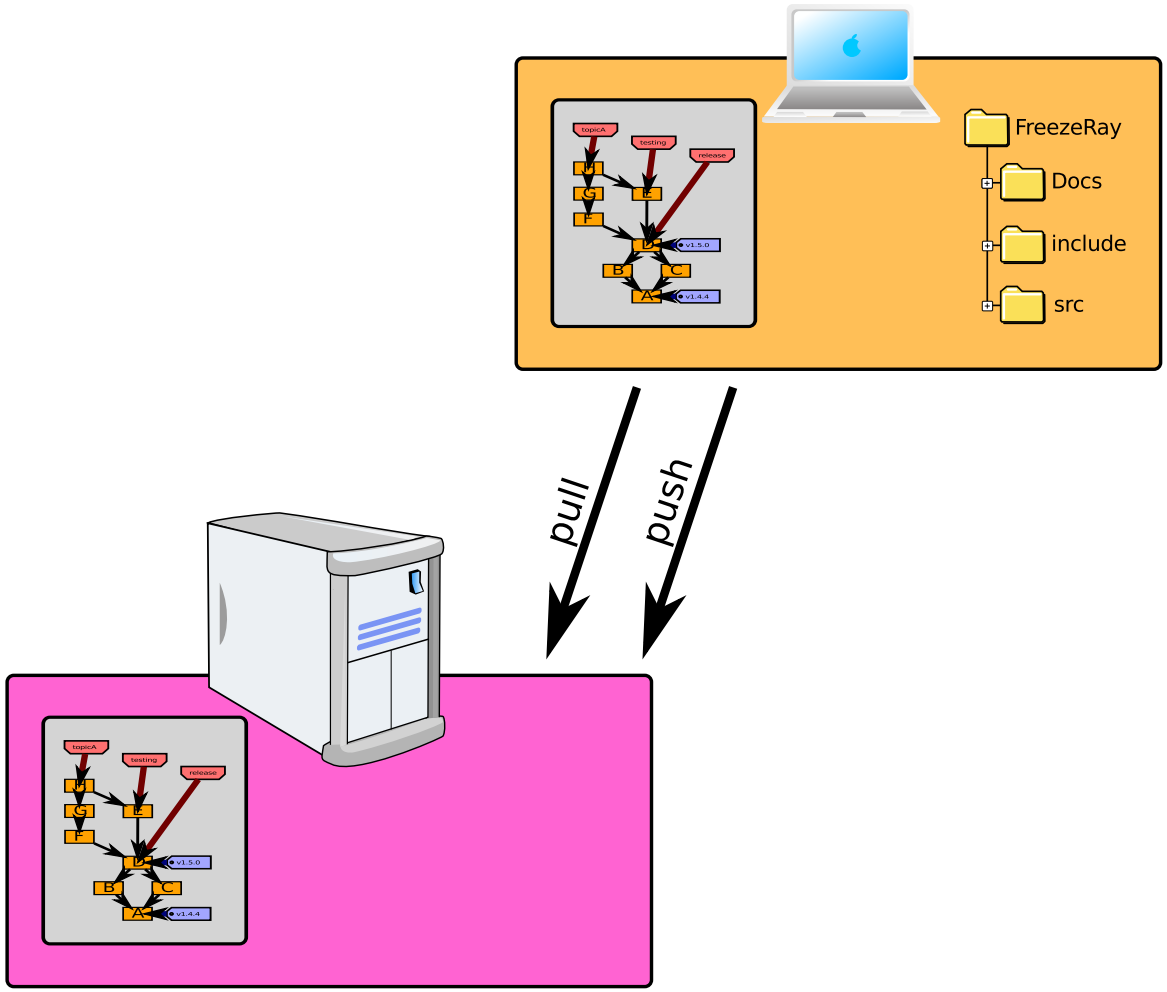
- Client has a clone of repository (operations are local)
- Operation pull (download)



- Client has a clone of repository (operations are local)
- Operation pull (download)

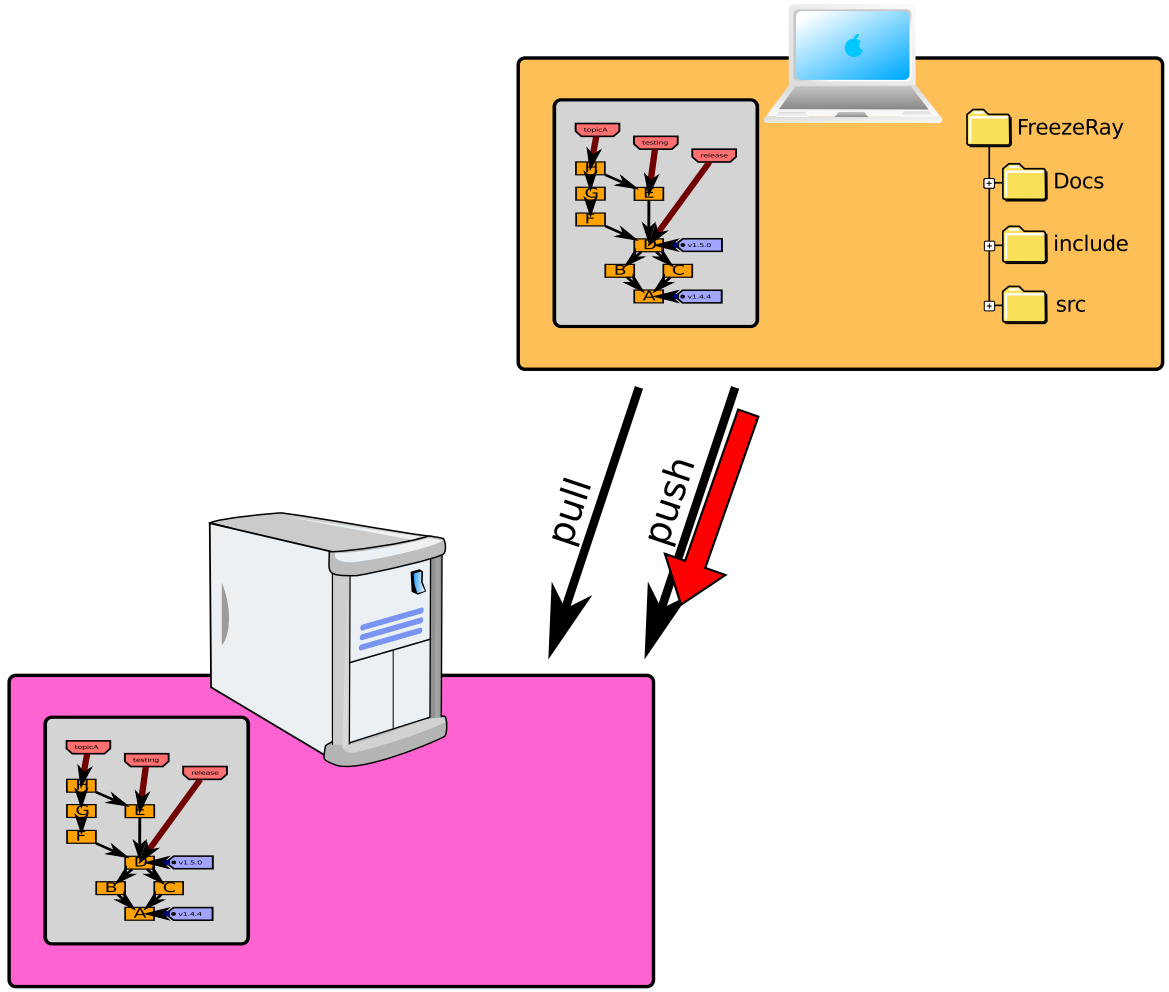


- Client has a clone of repository (operations are local)
- Operation pull (download) and push (upload to server)

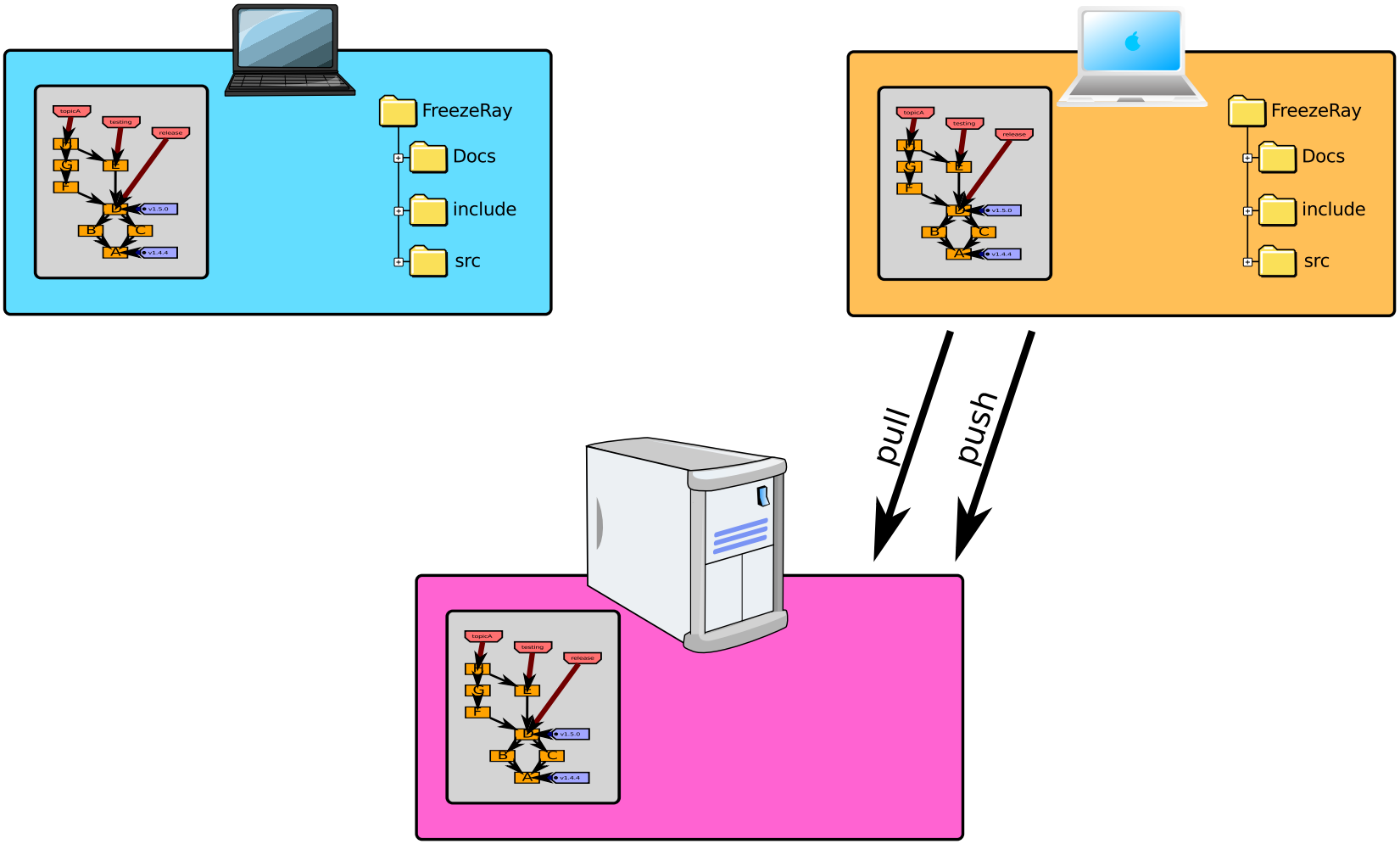


Decentralized version control

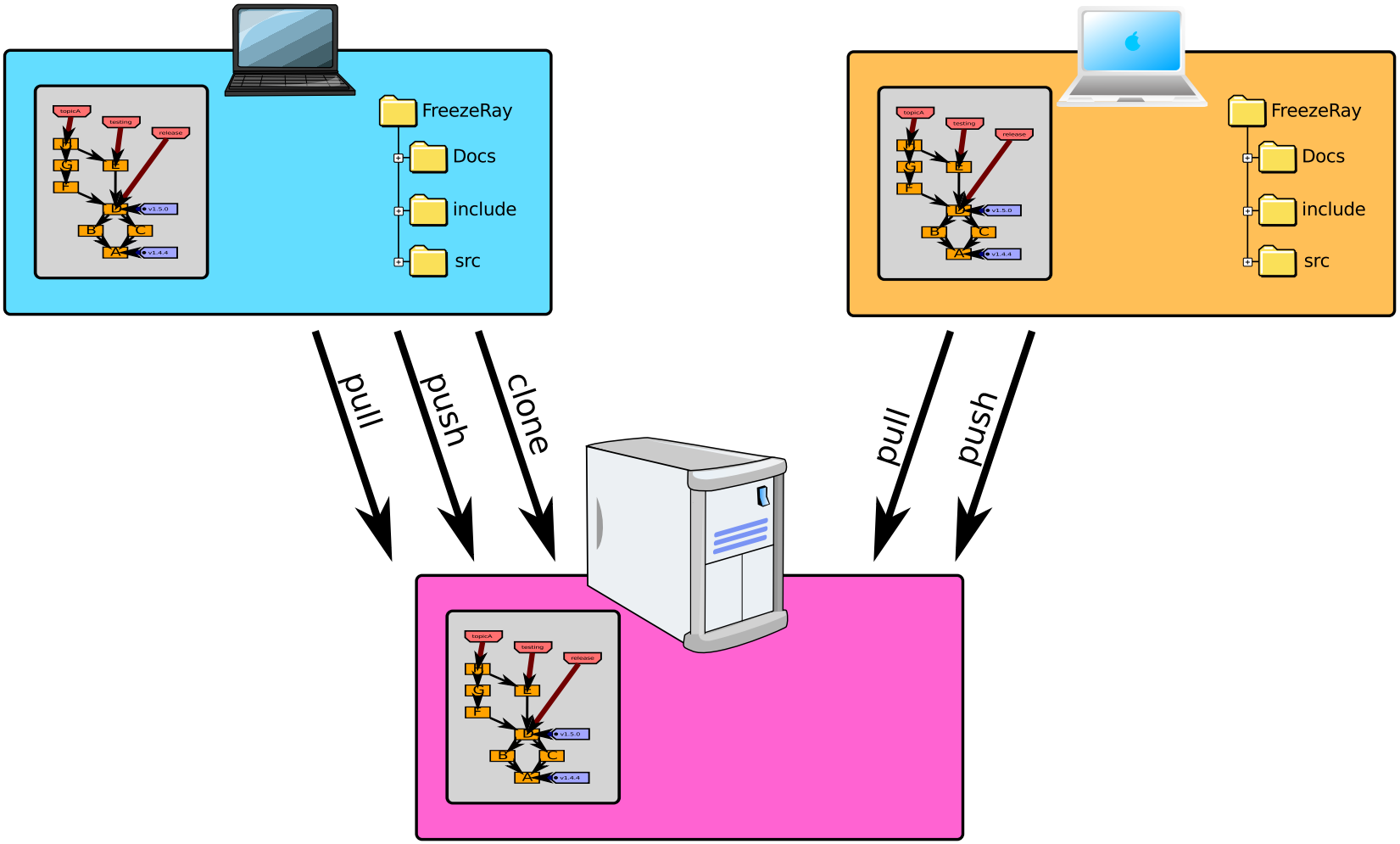
- Client has a clone of repository (operations are local)
- Operation pull (download) and push (upload to server)



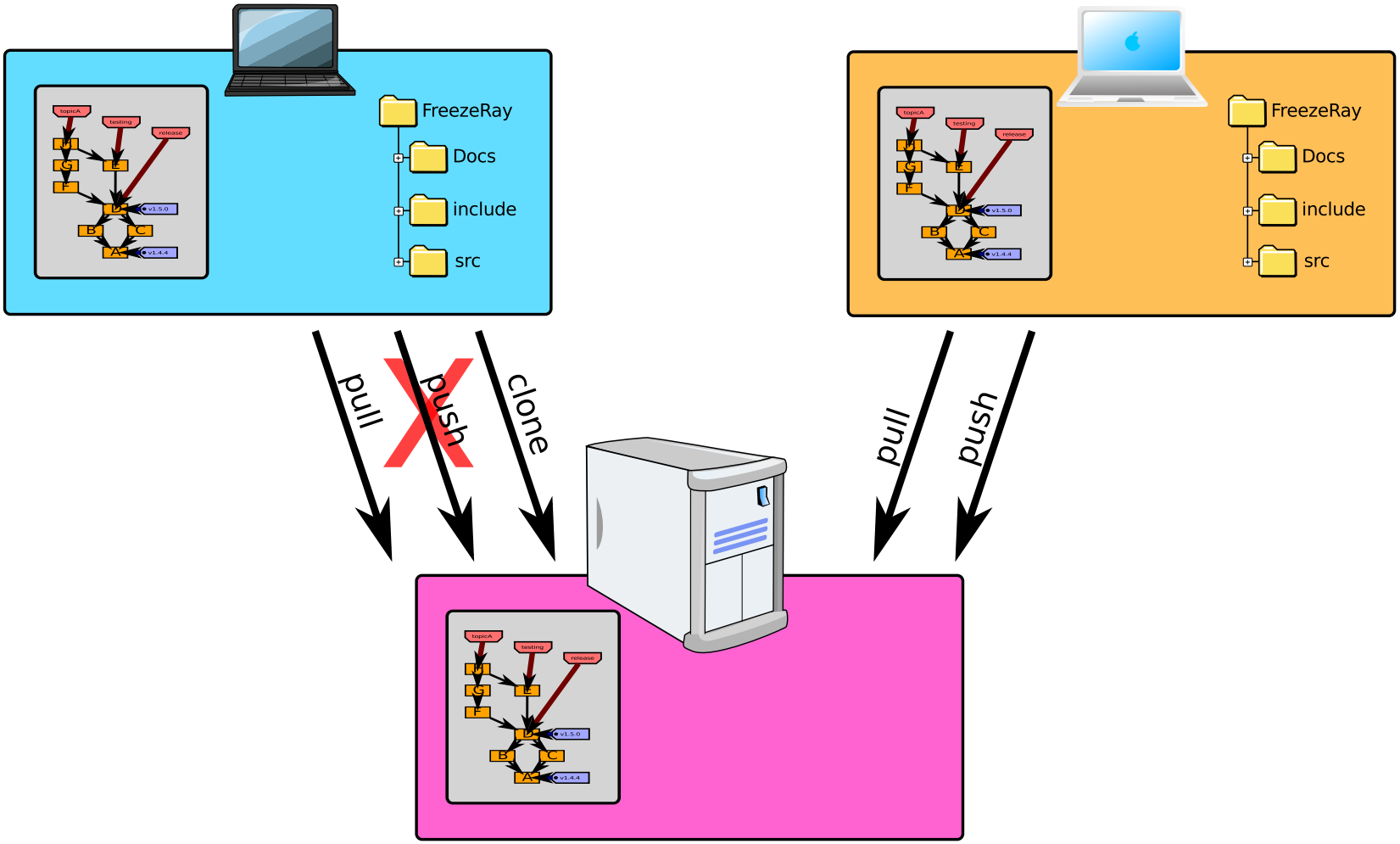
- Client has a clone of repository (operations are local)
- Operation pull (download) and push (upload to server)



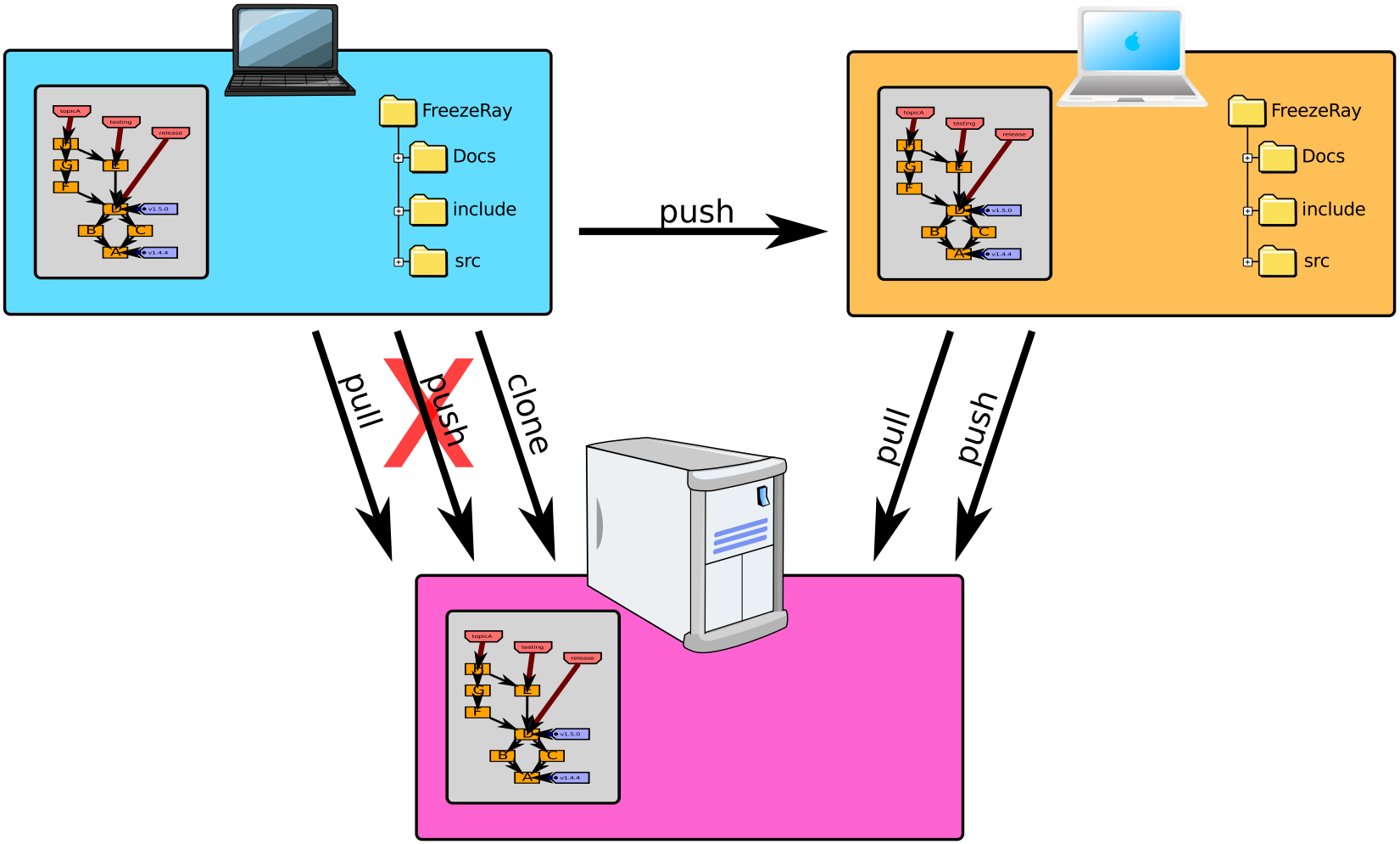
- Client has a clone of repository (operations are local)
- Operation pull (download) and push (upload to server)



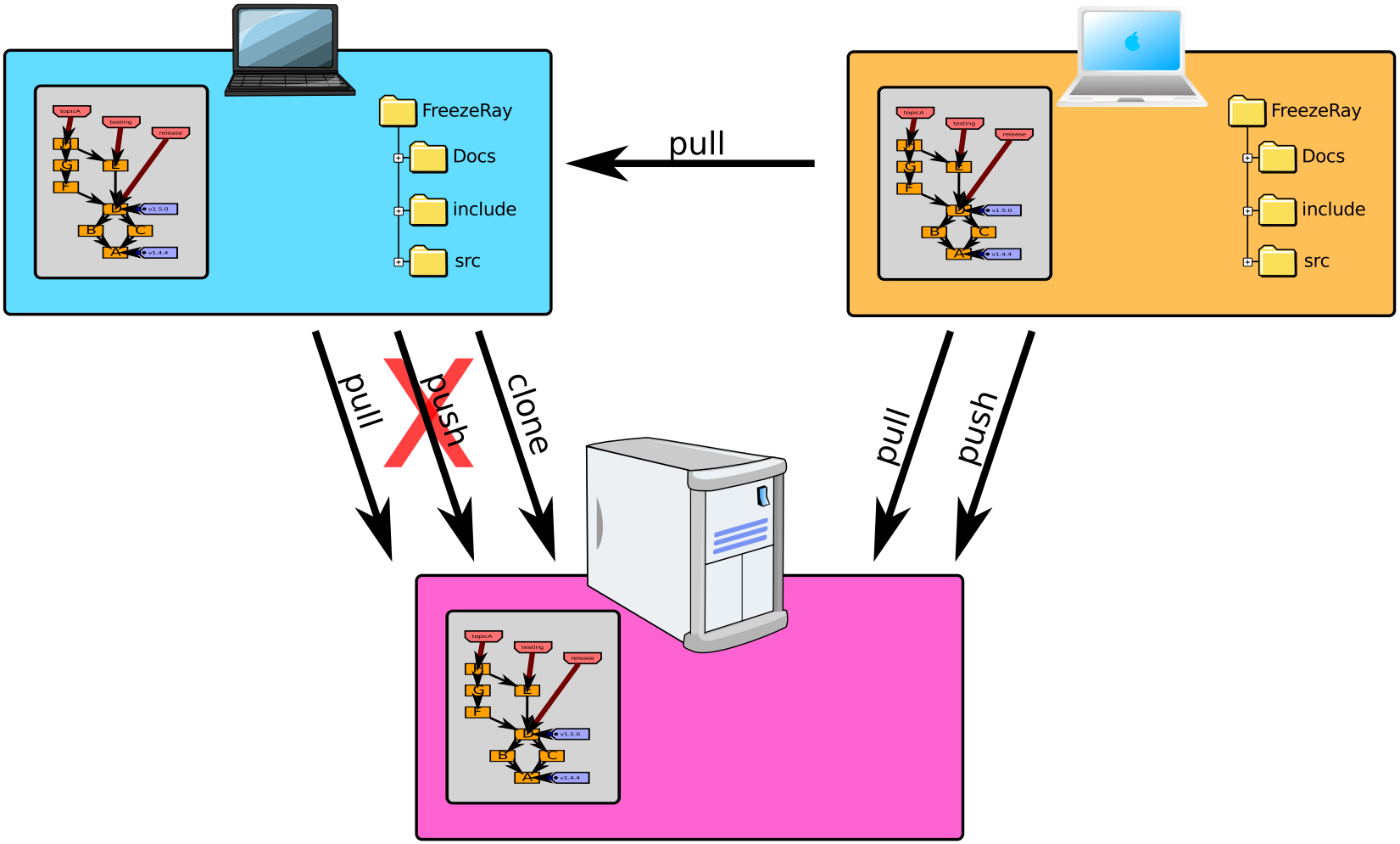
- Client has a clone of repository (operations are local)
- Operation pull (download) and push (upload to server)
- Anyone can be server (but may not grant write permission)



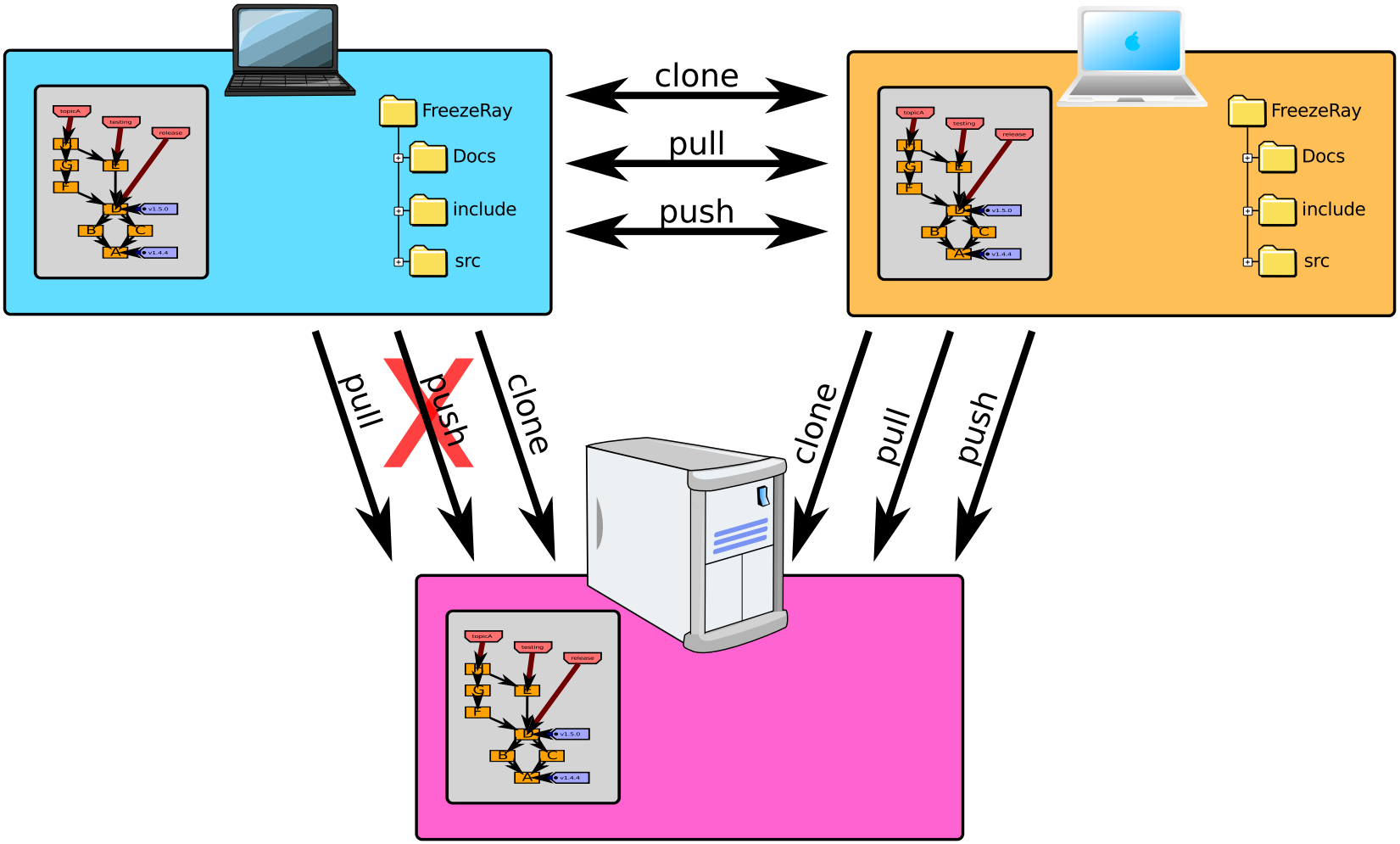
- Client has a clone of repository (operations are local)
- Operation pull (download) and push (upload to server)
- Anyone can be server (but may not grant write permission)



- Client has a clone of repository (operations are local)
- Operation pull (download) and push (upload to server)
- Anyone can be server (but may not grant write permission)



- Client has a clone of repository (operations are local)
- Operation pull (download) and push (upload to server)
- Anyone can be server (but may not grant write permission)



Advantages of distributed version control

- Speed (local operations)
- Non-invasive "micro" commits
  - Commit frequently
  - Commit small changes (code in our branch does not have to work)
  - Does not break build

Advantages of distributed version control

- Speed (local operations)
- Non-invasive "micro" commits
  - Commit frequently
  - Commit small changes (code in our branch does not have to work)
  - Does not break build
- Offline work
- Absence of single point of failure

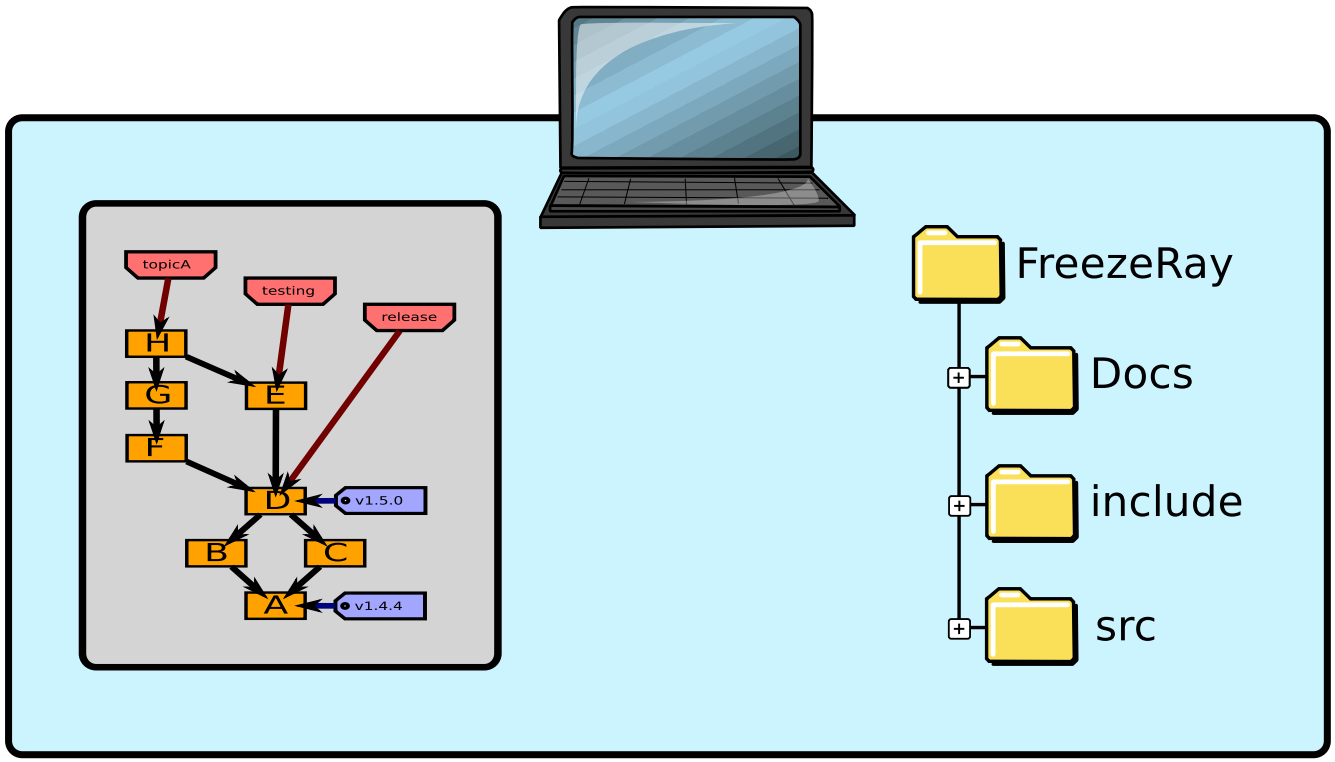
Advantages of distributed version control

- Speed (local operations)
- Non-invasive "micro" commits
  - Commit frequently
  - Commit small changes (code in our branch does not have to work)
  - Does not break build
- Offline work
- Absence of single point of failure
- Trivial backup
  - each developer is a backup

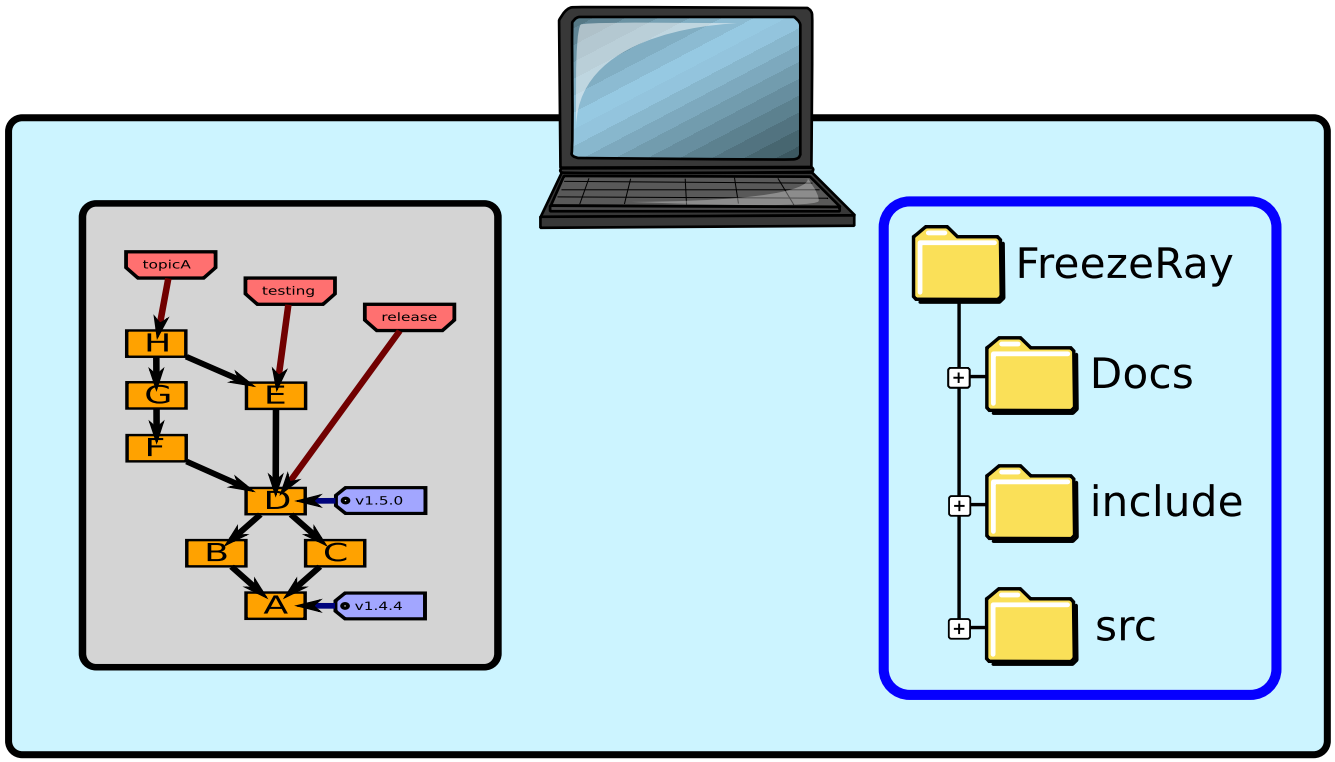
Topology of distributed VCS

- Single developer
- Centralized
- Integrator
- Dictator and lieutanants

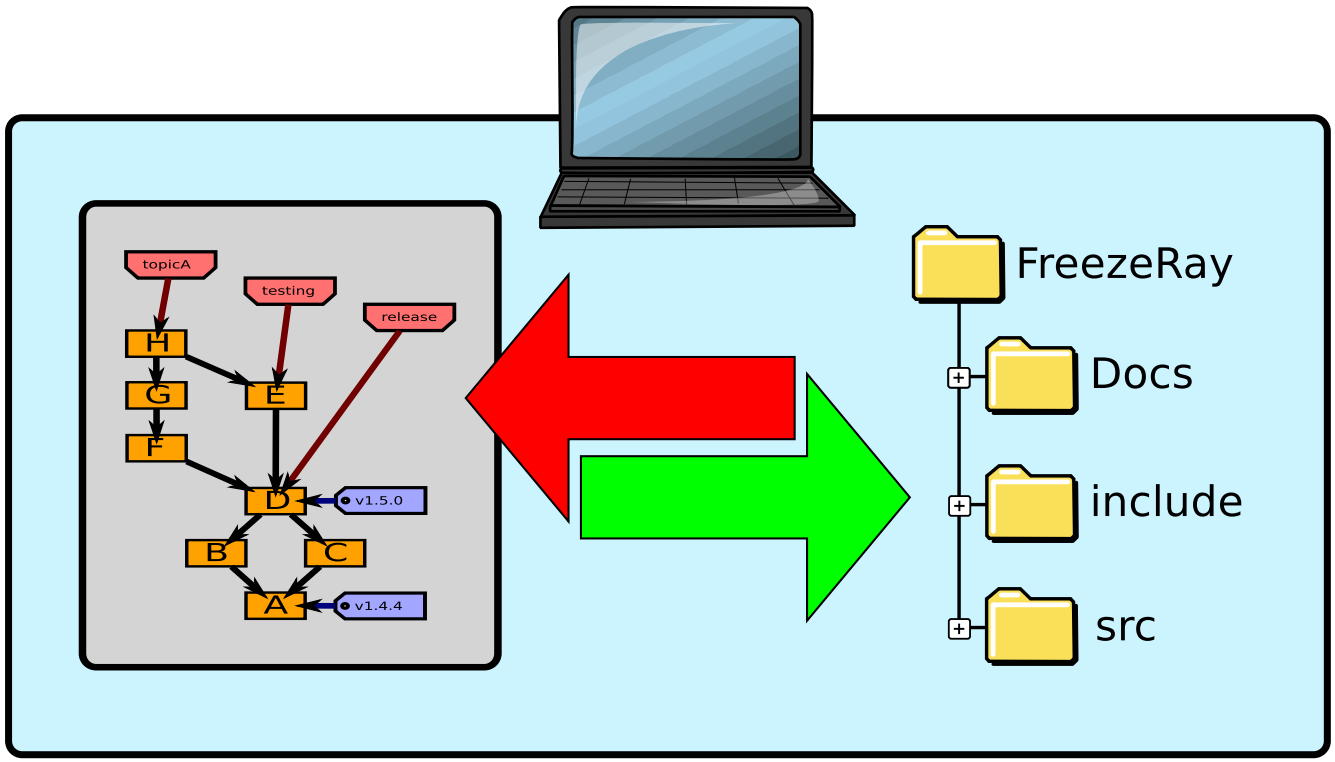
Developer creates their project locally



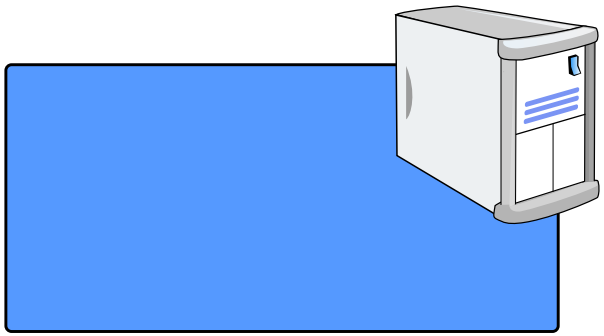
Developer creates their project locally

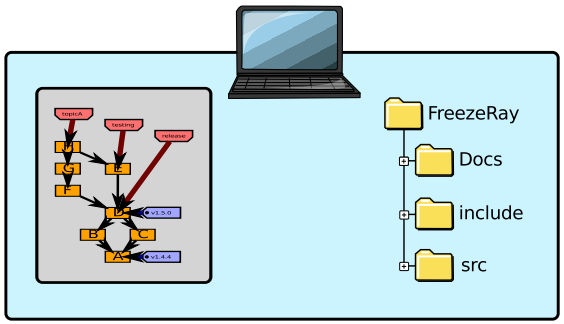


Developer creates their project locally

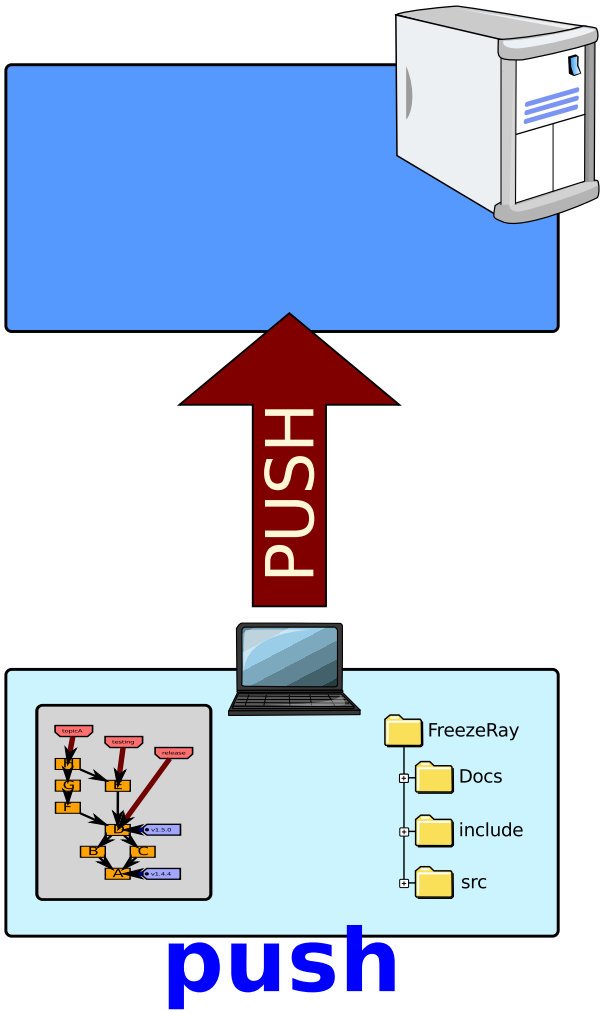


Code publishing

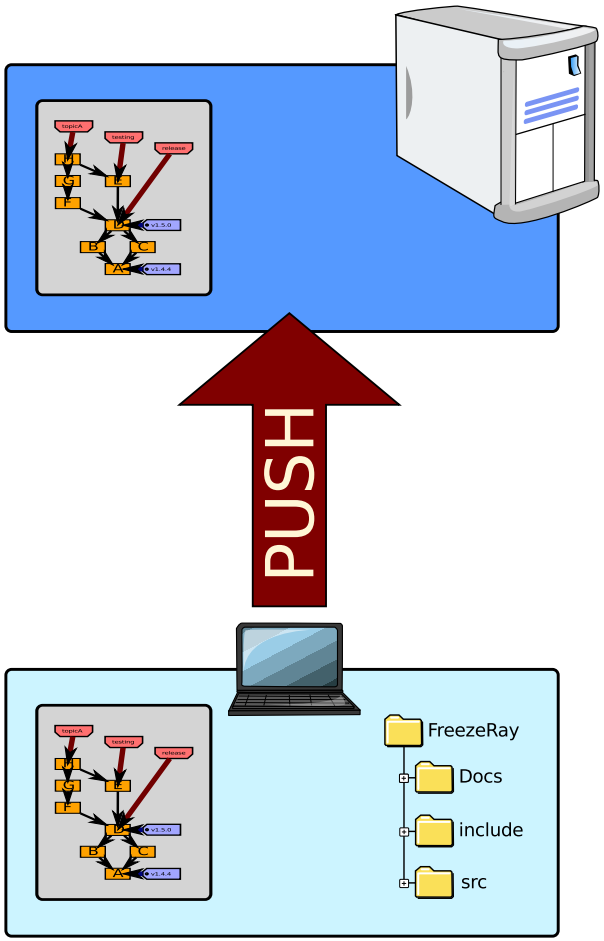




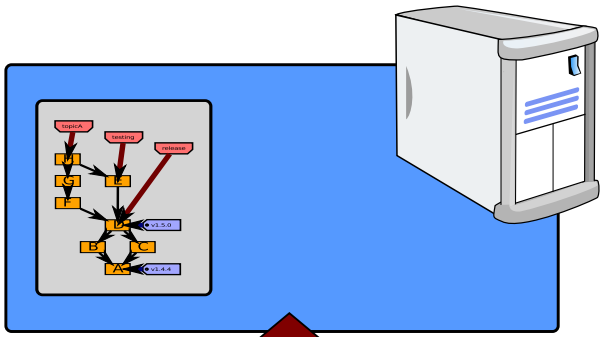
Code publishing

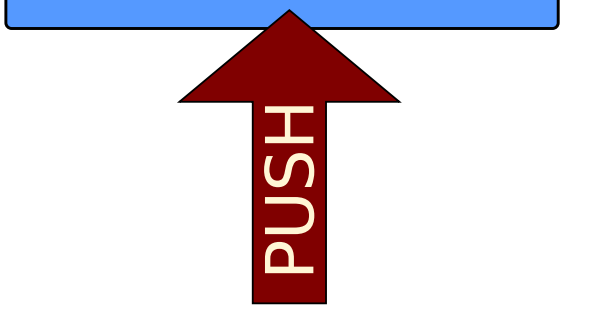


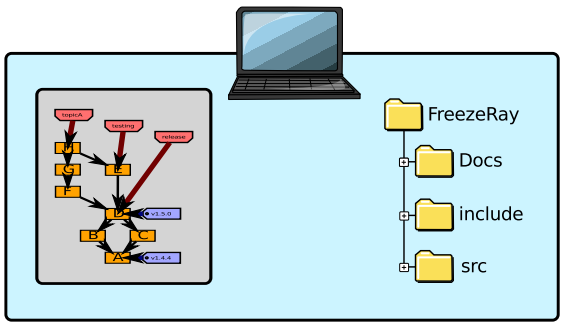
Code publishing

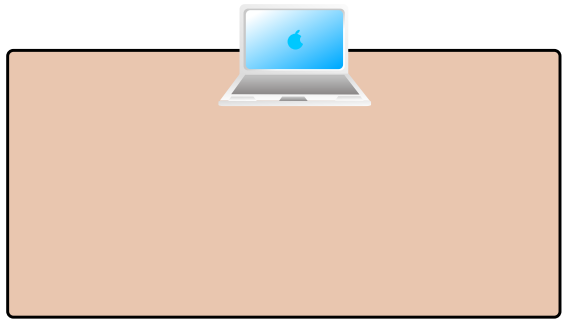


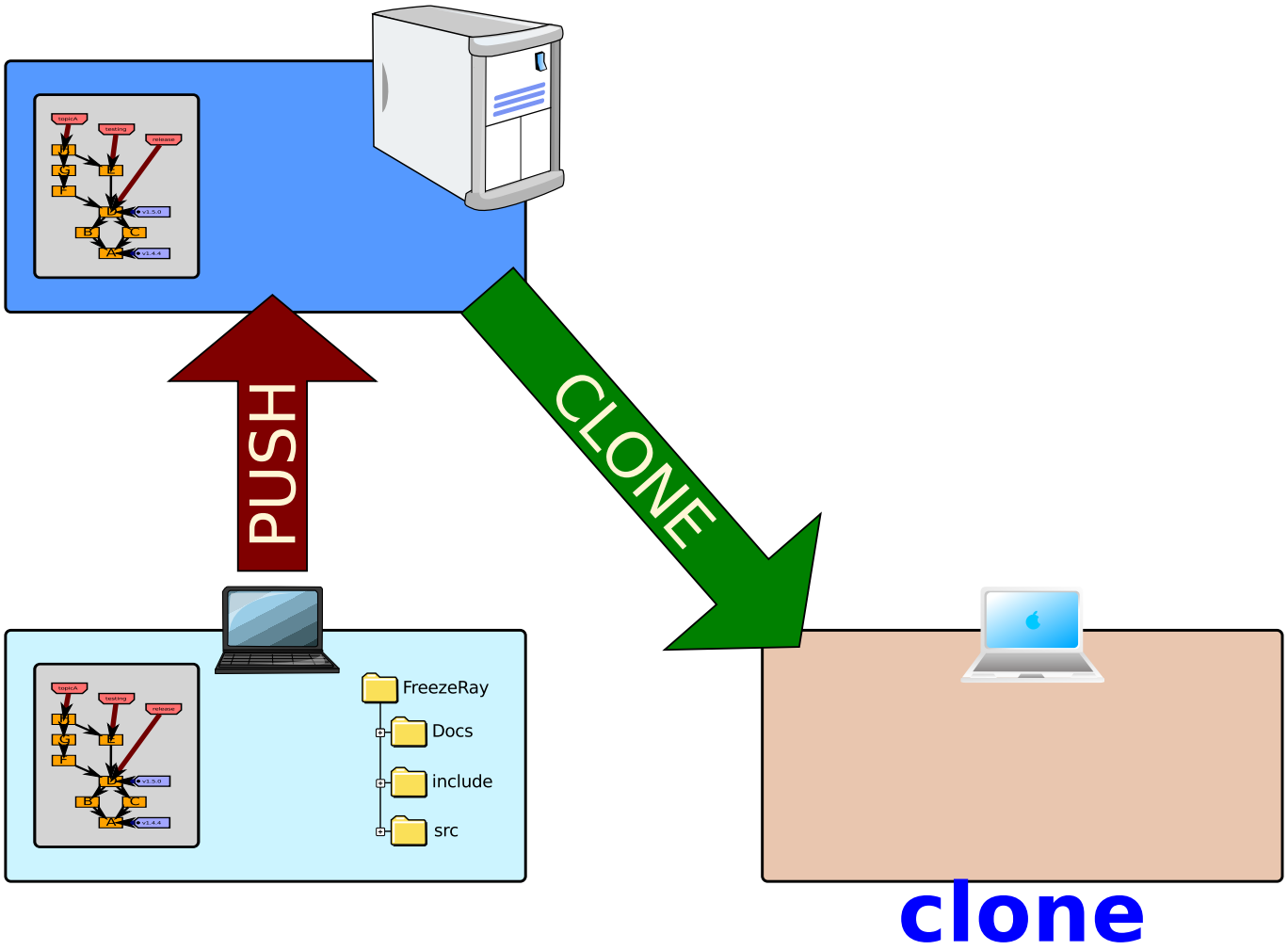
Code publishing



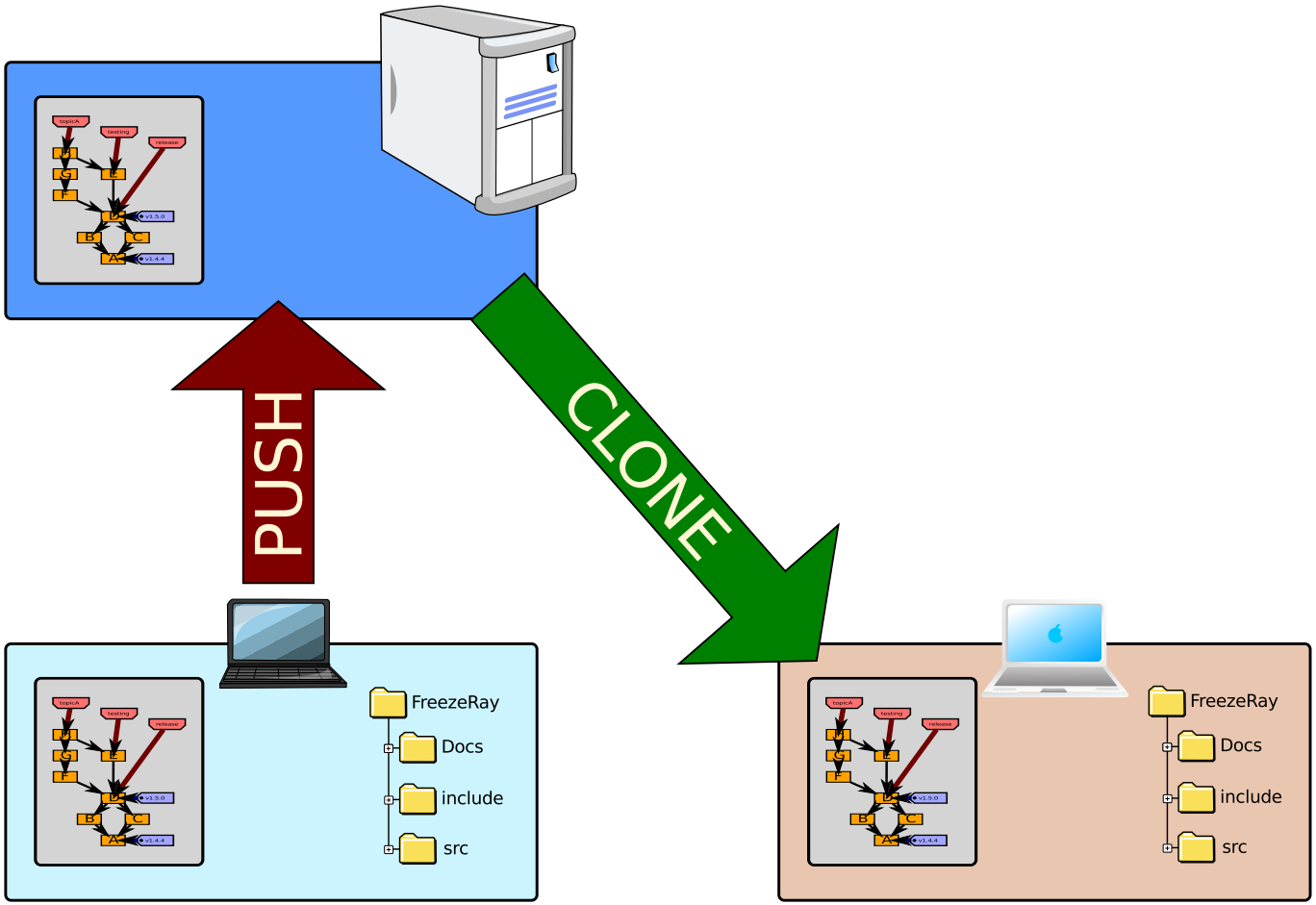


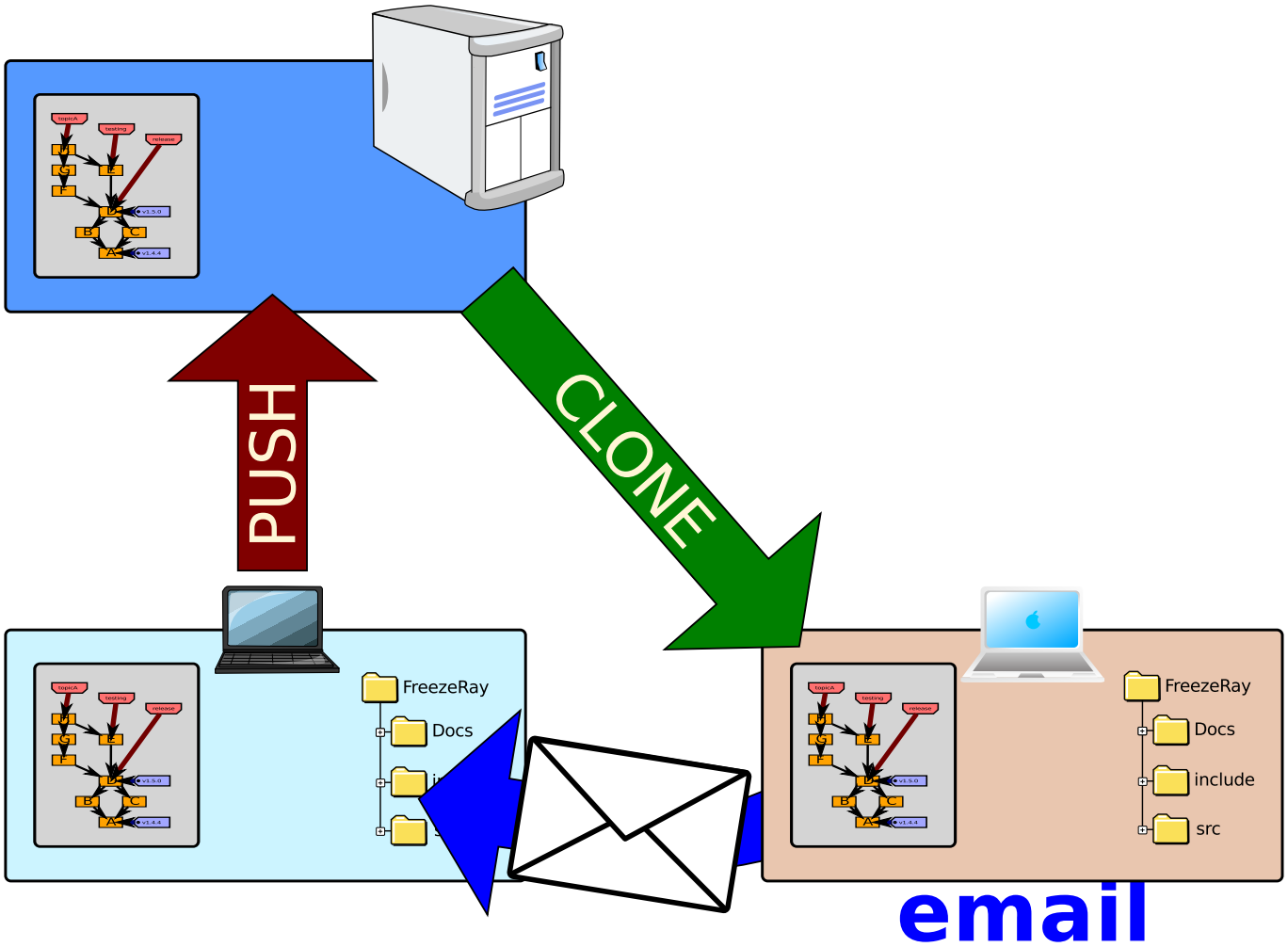





PUSH
CLONE
FreezeRay
Docs
include
src
clone

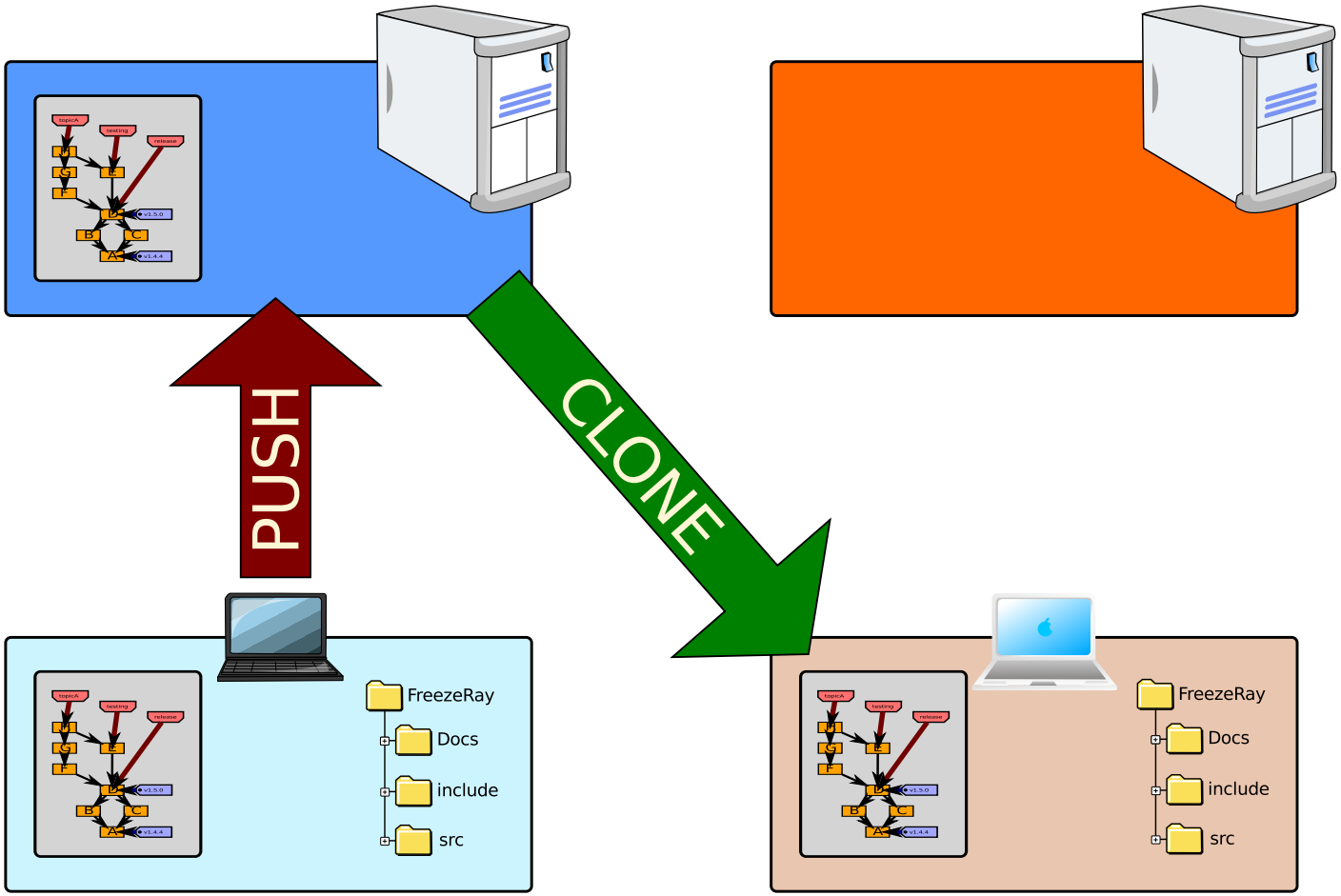
Code publishing

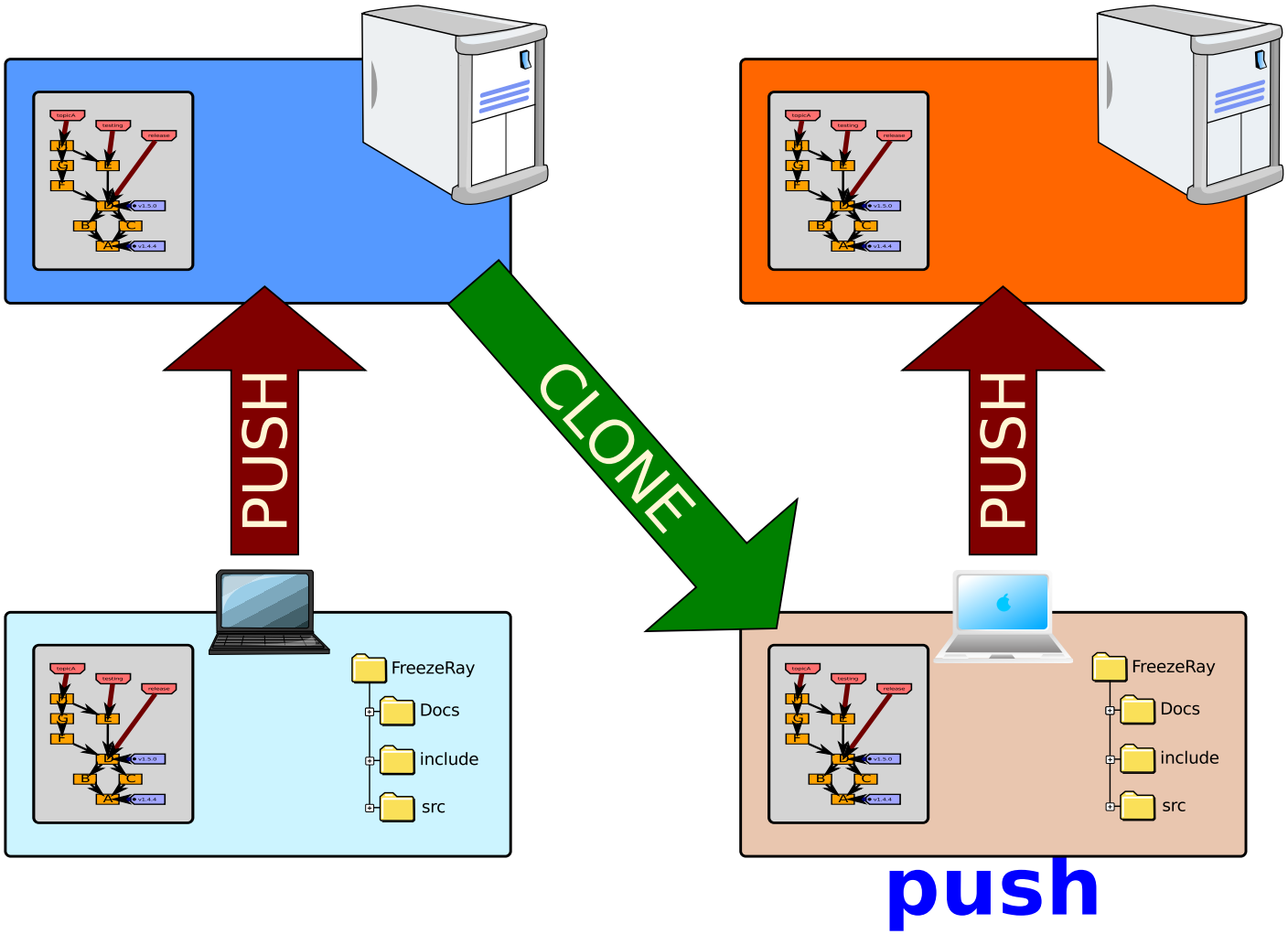




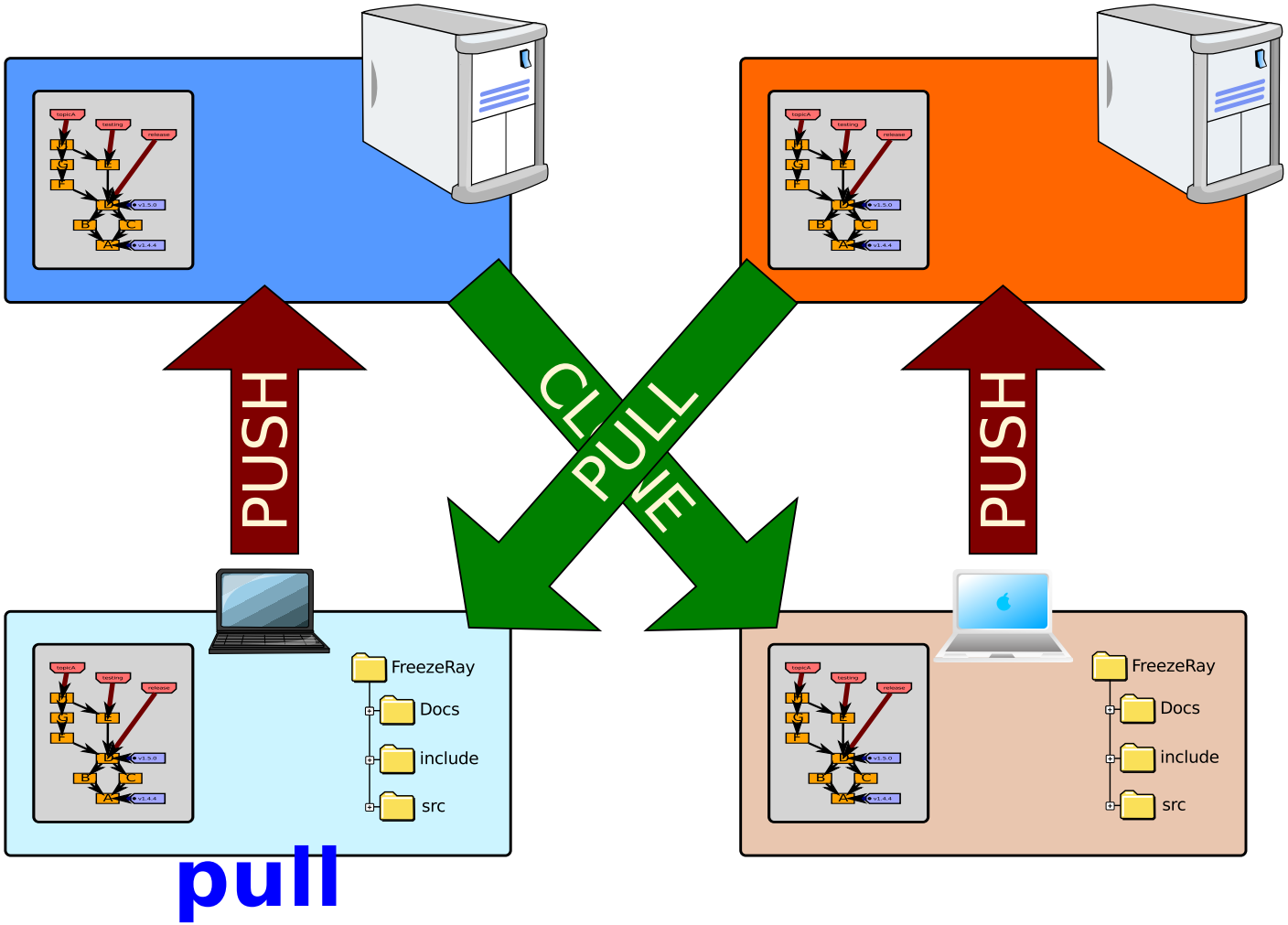
Patch sent via email

Code publishing



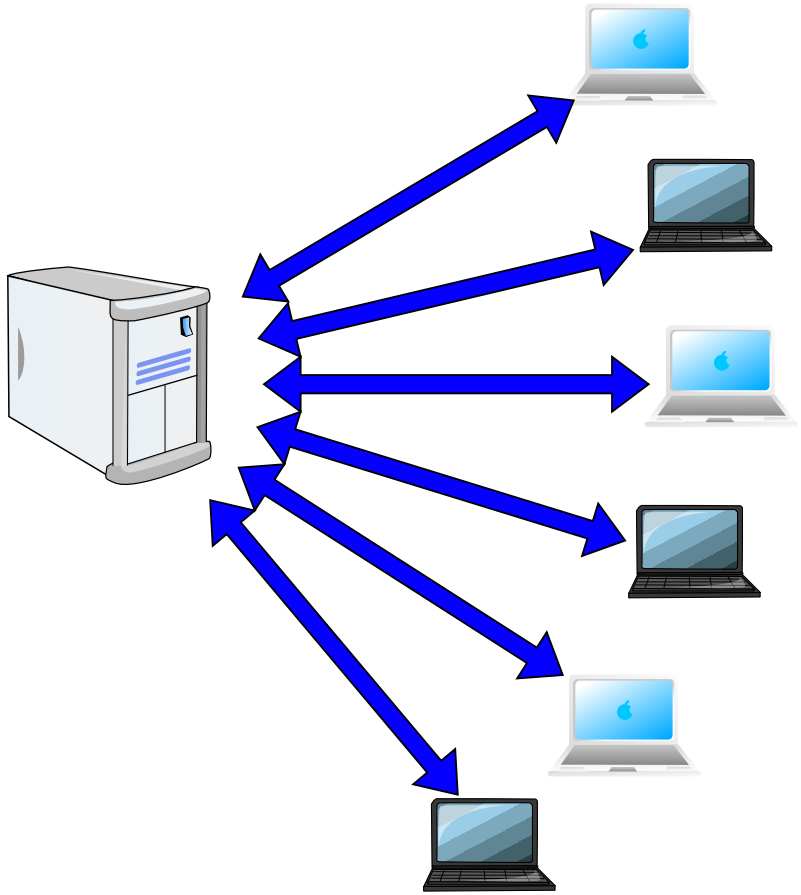


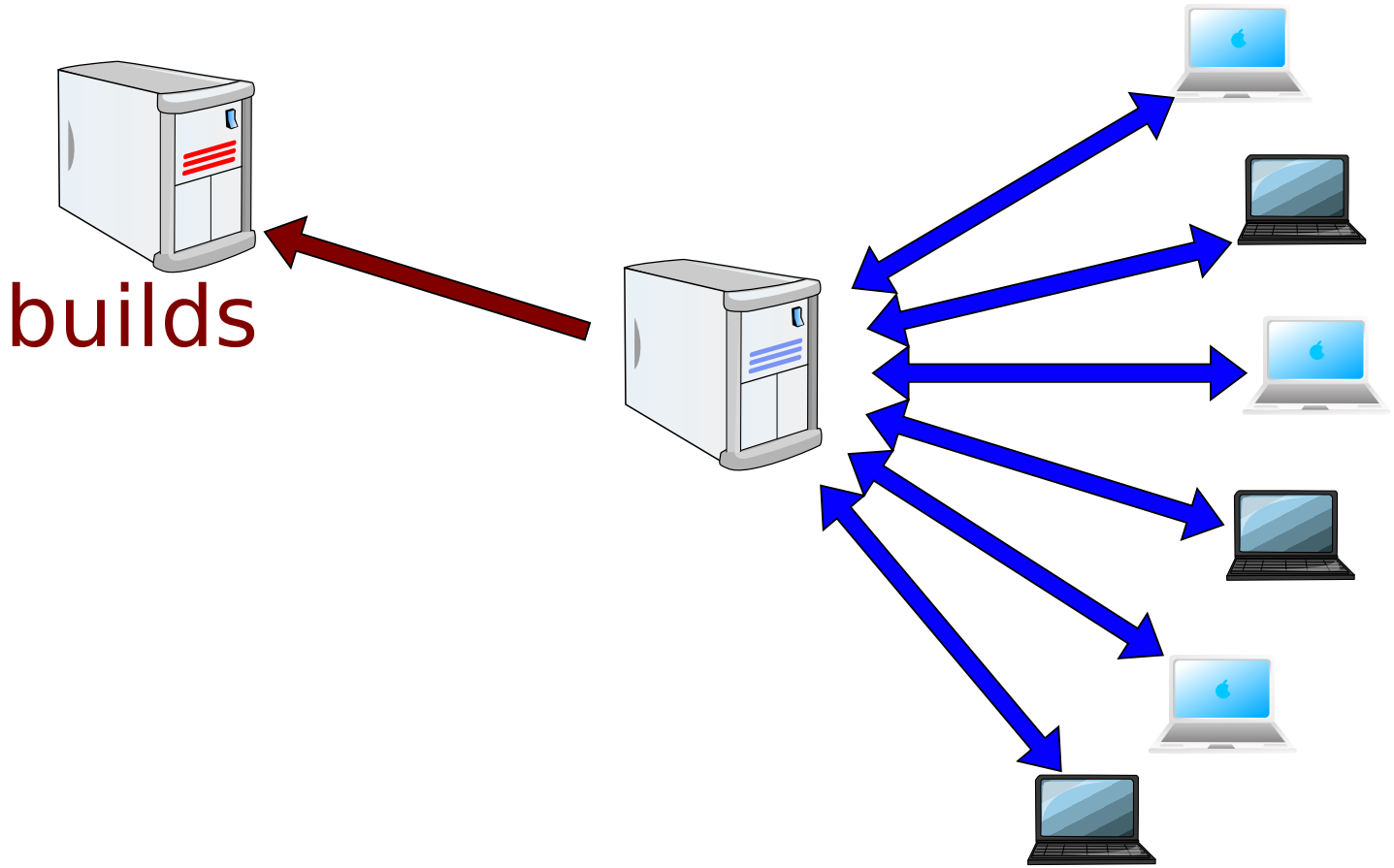
Publish to own repository and send Pull request



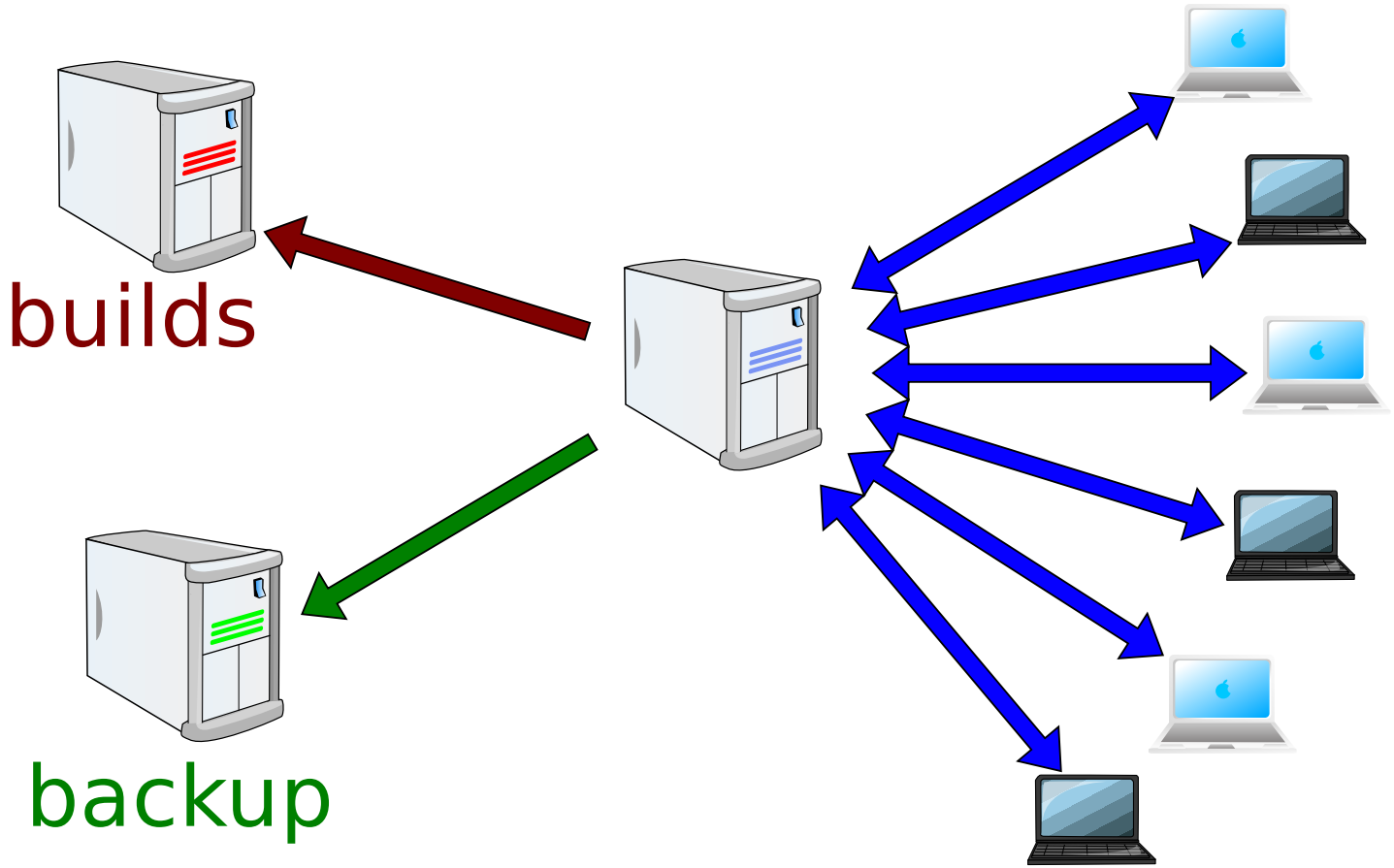
Publish to own repository and send Pull request

Centralized topology

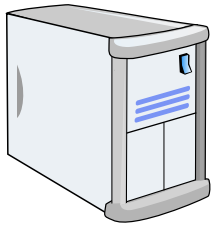


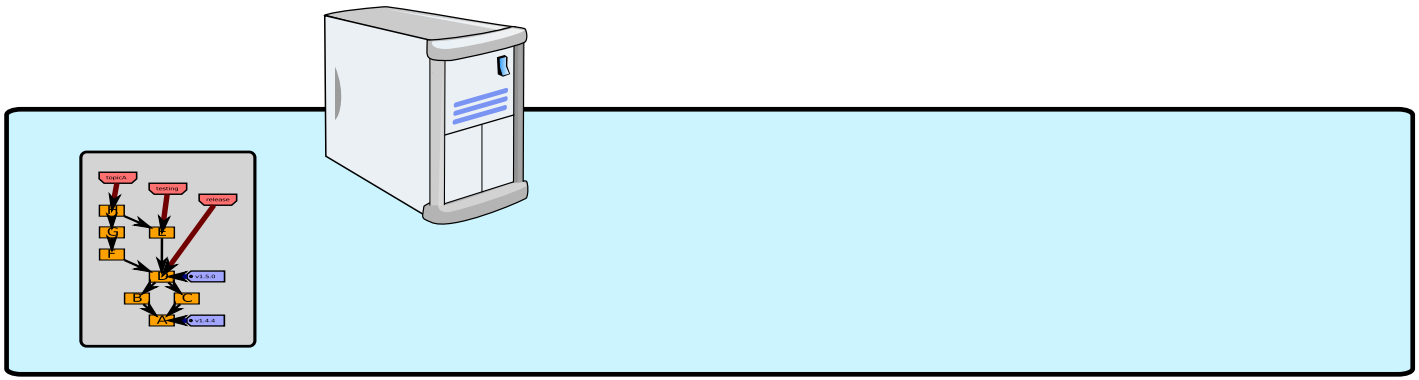


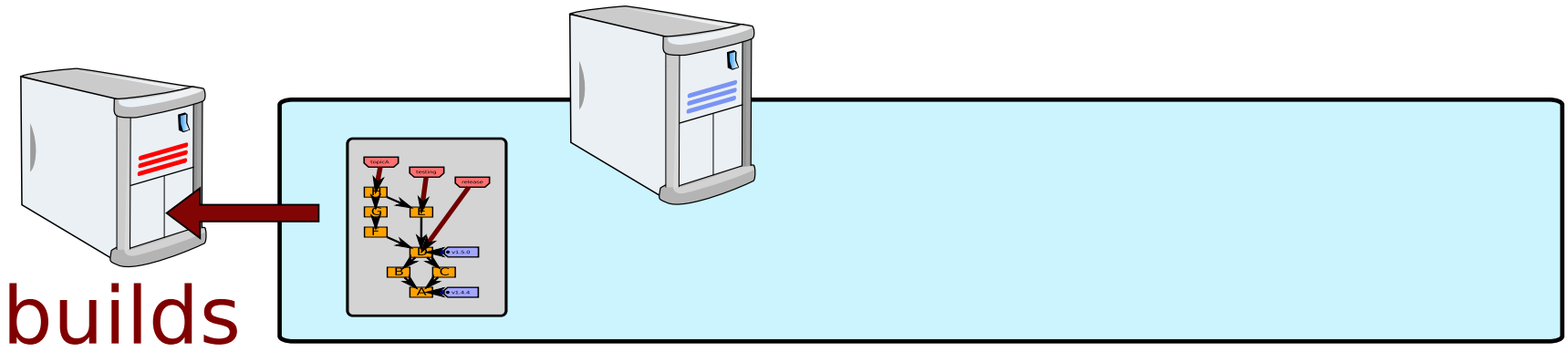
Specialized servers for building

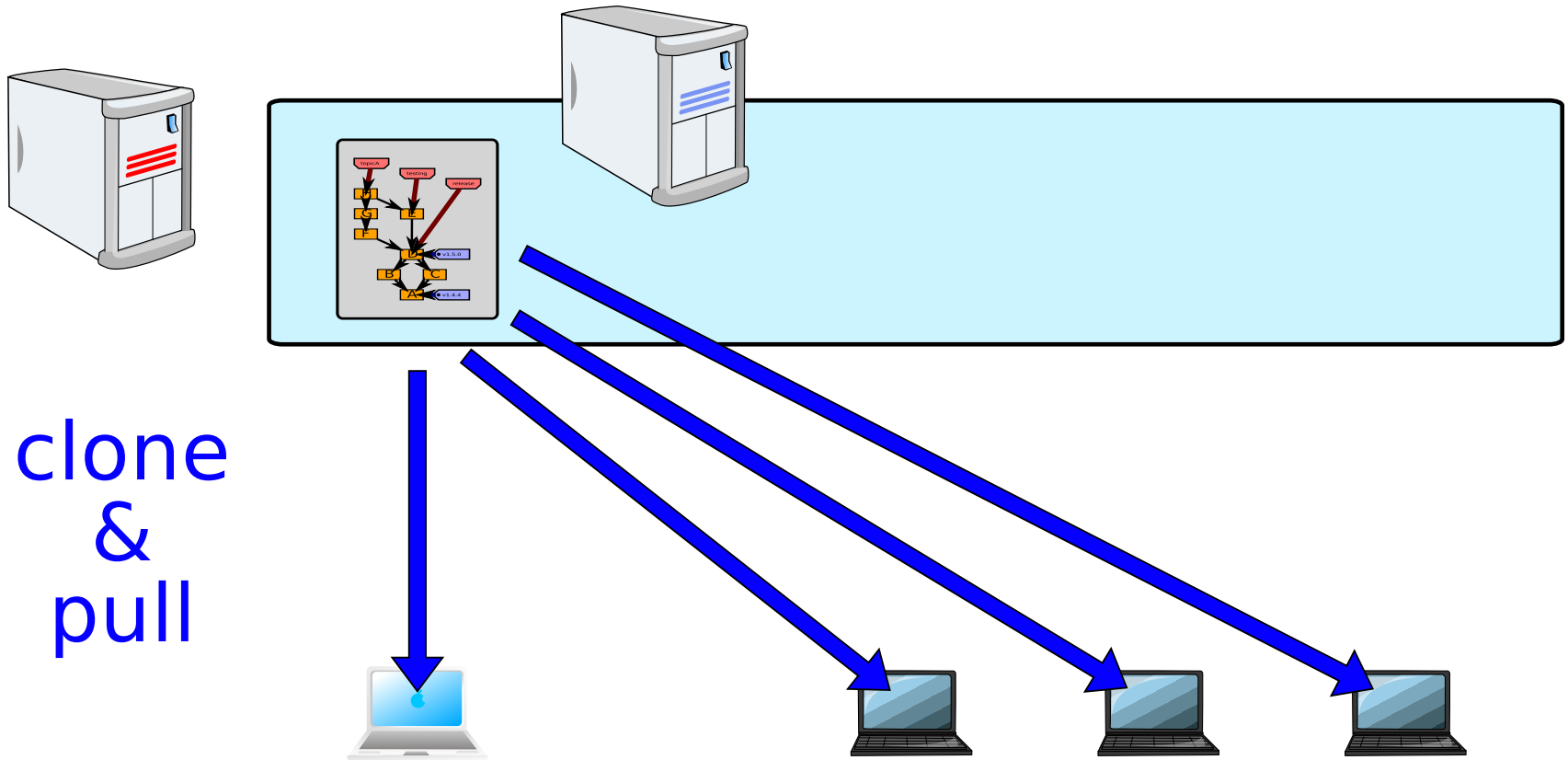


Specialized servers for building and backup

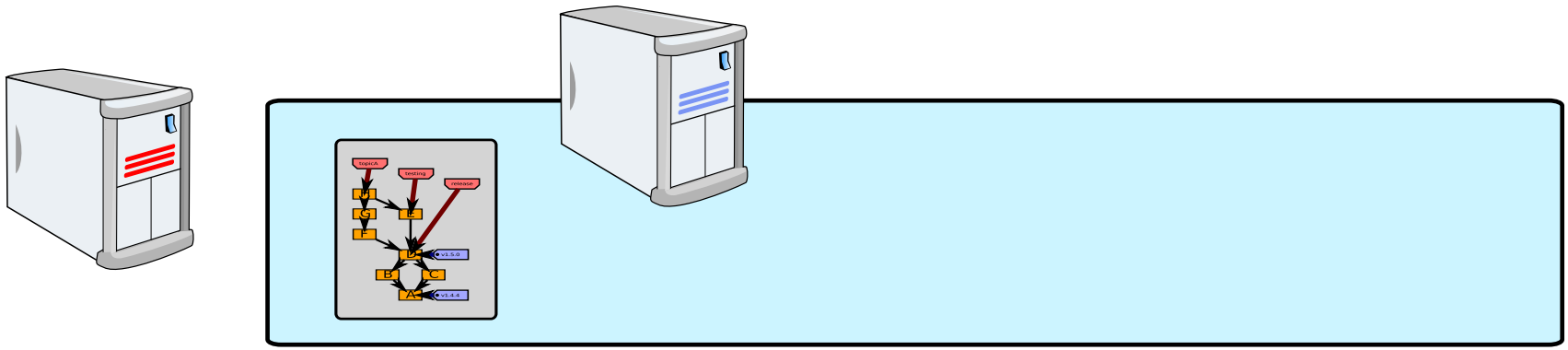



topics
testing
release
M
G
F
L
D
v1.5.0
B
C
A
v1.4.4


builds



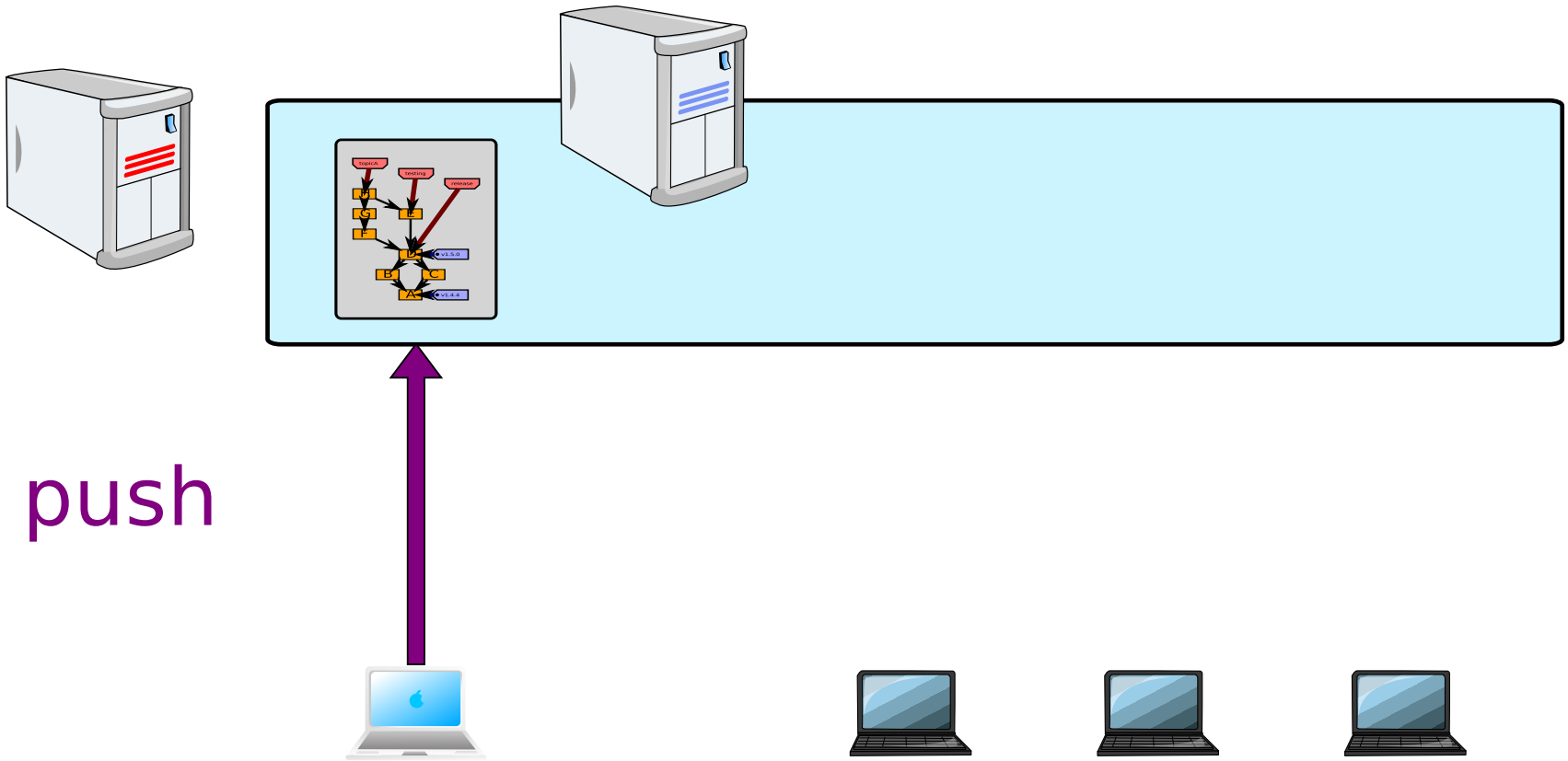
Anyone can read it



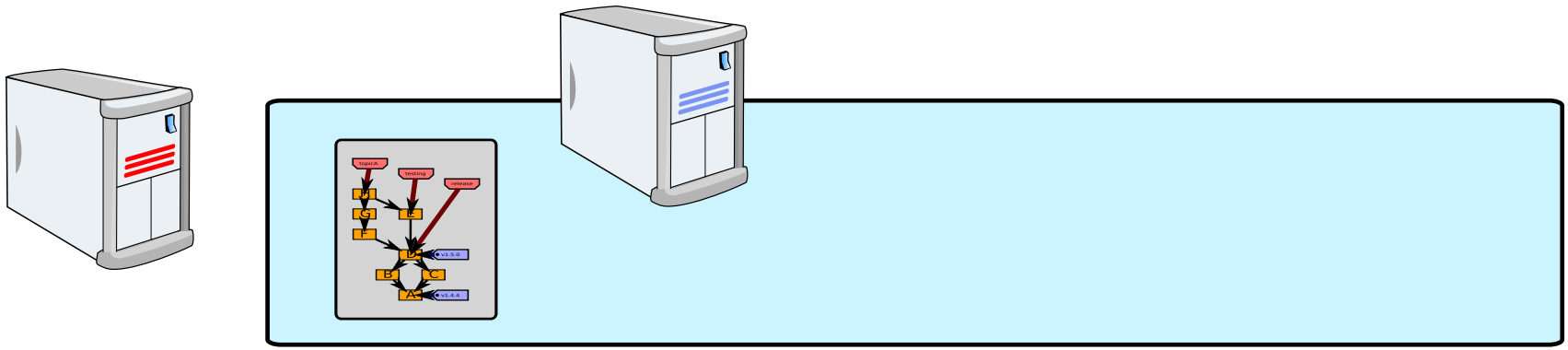
gate keeper



Only one developer can write

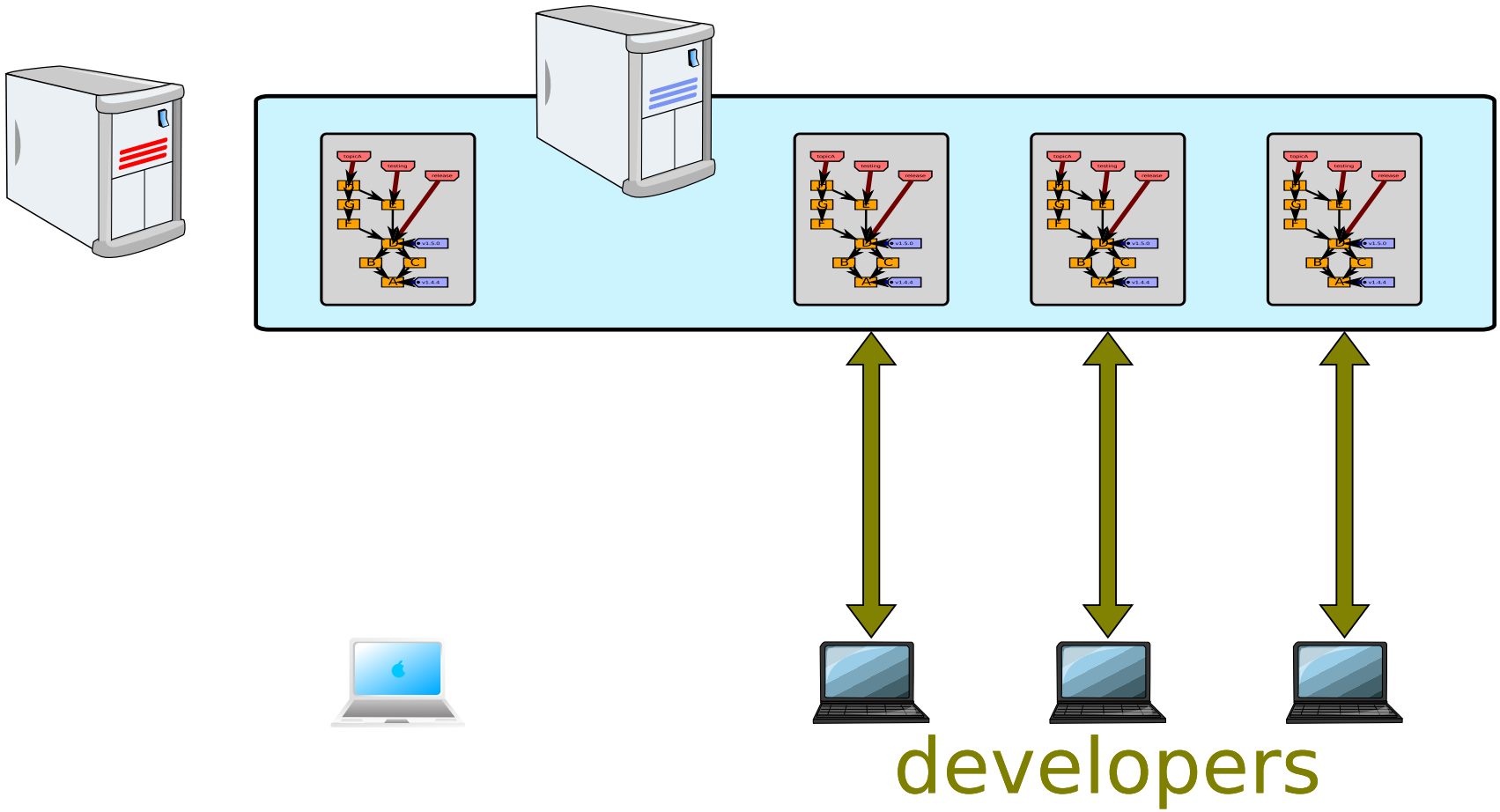


Only one developer can write

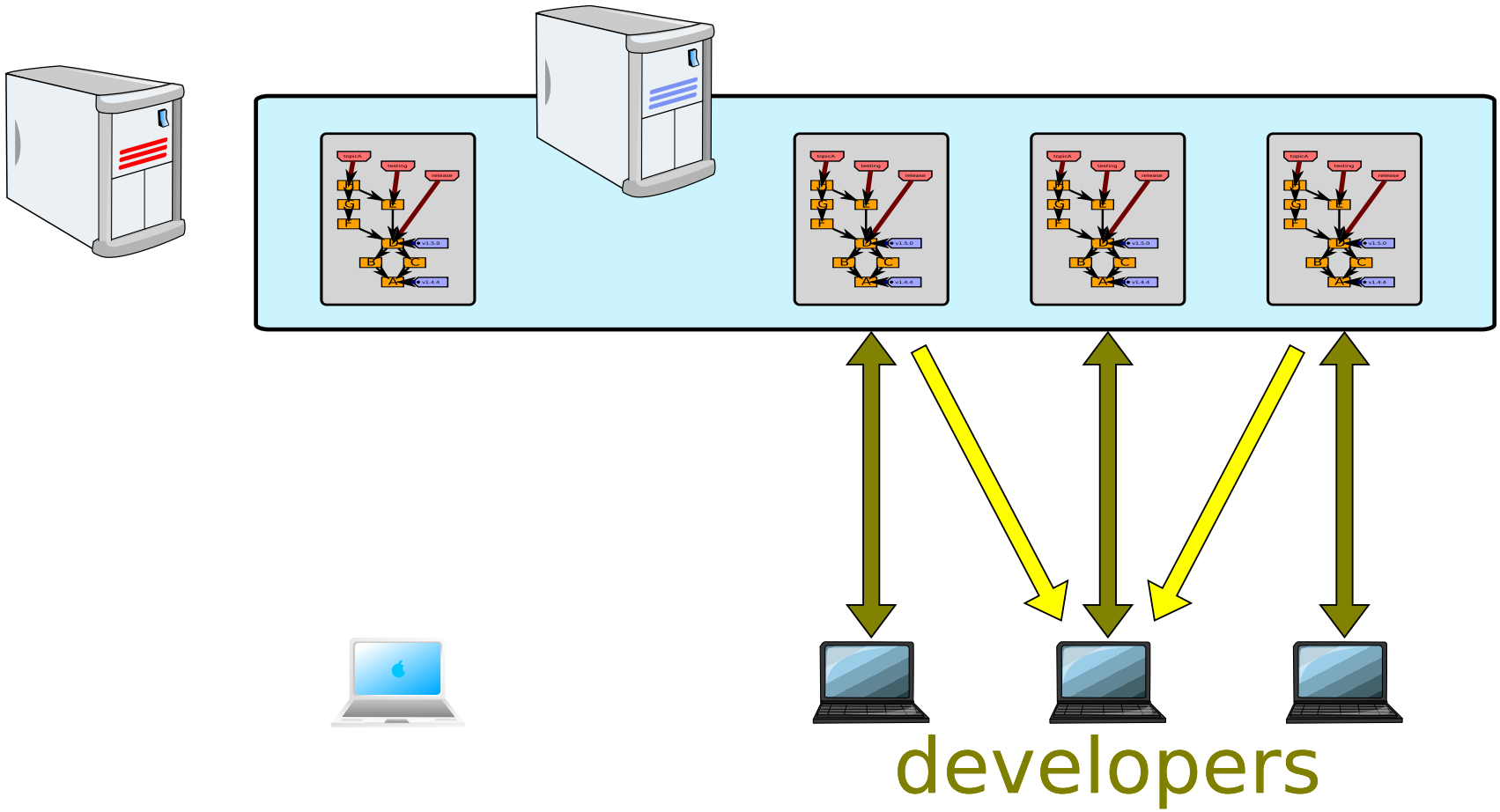


developers

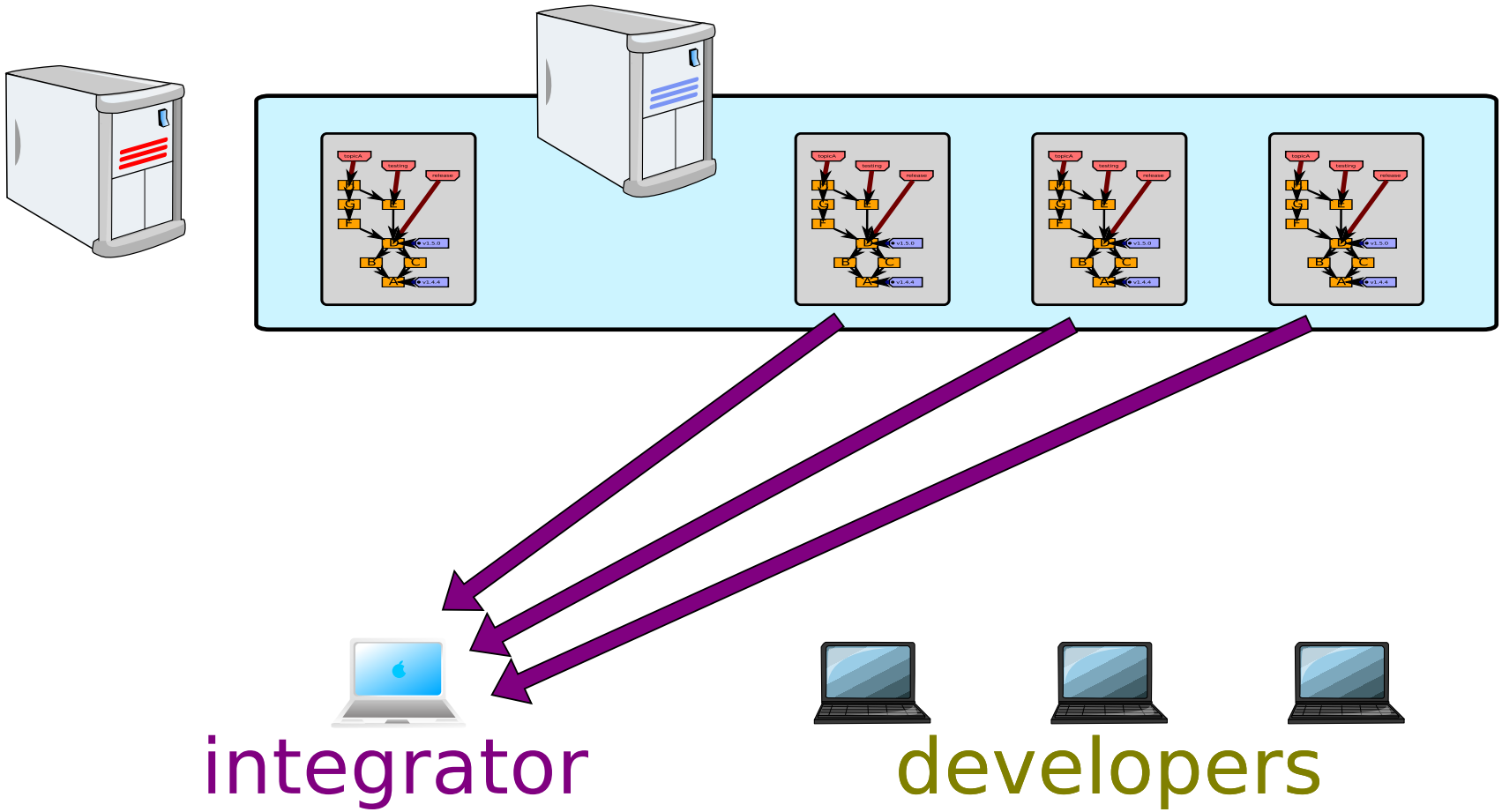
Other developers maintain their repository clones



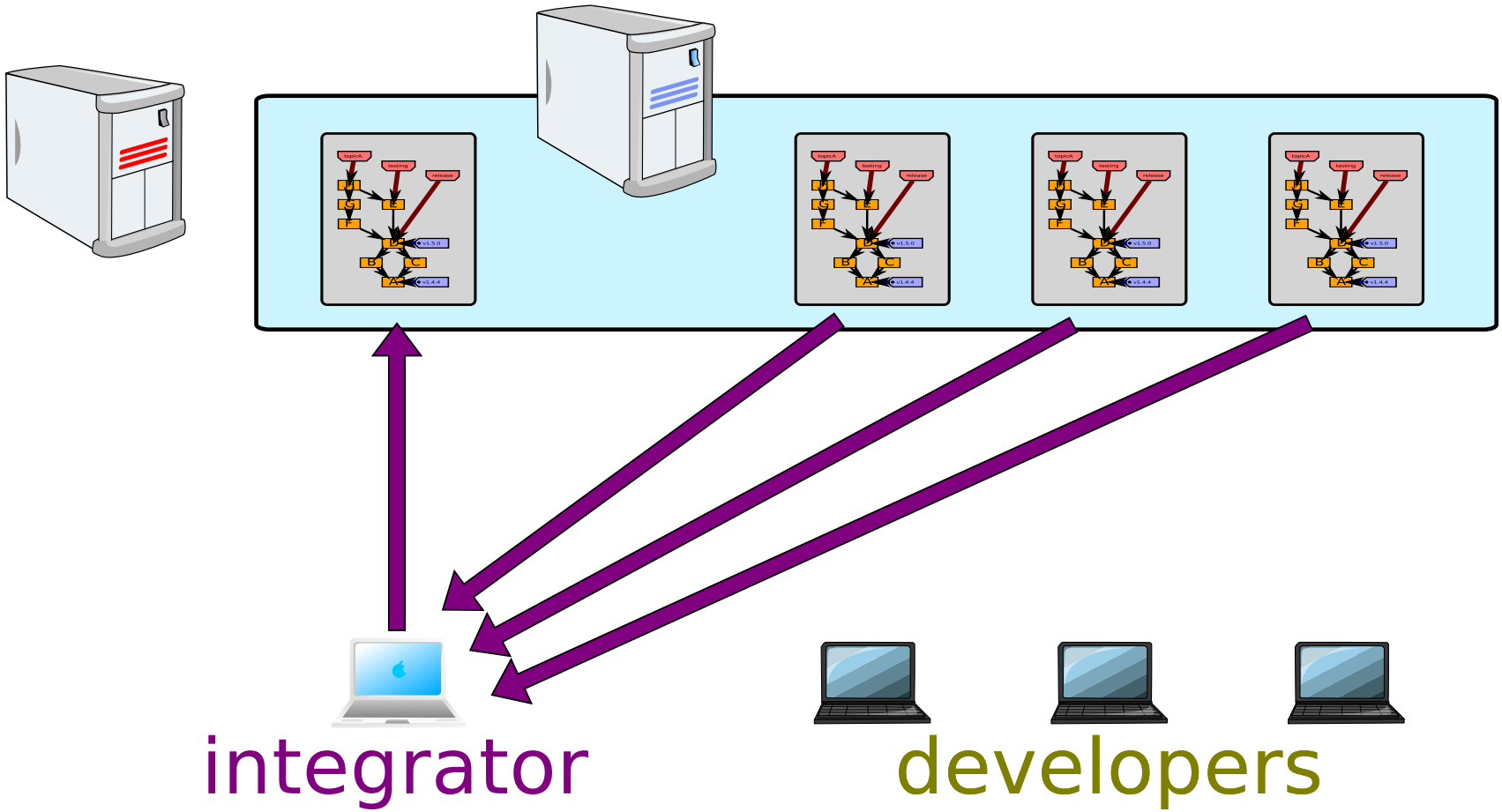
Other developers maintain their repository clones



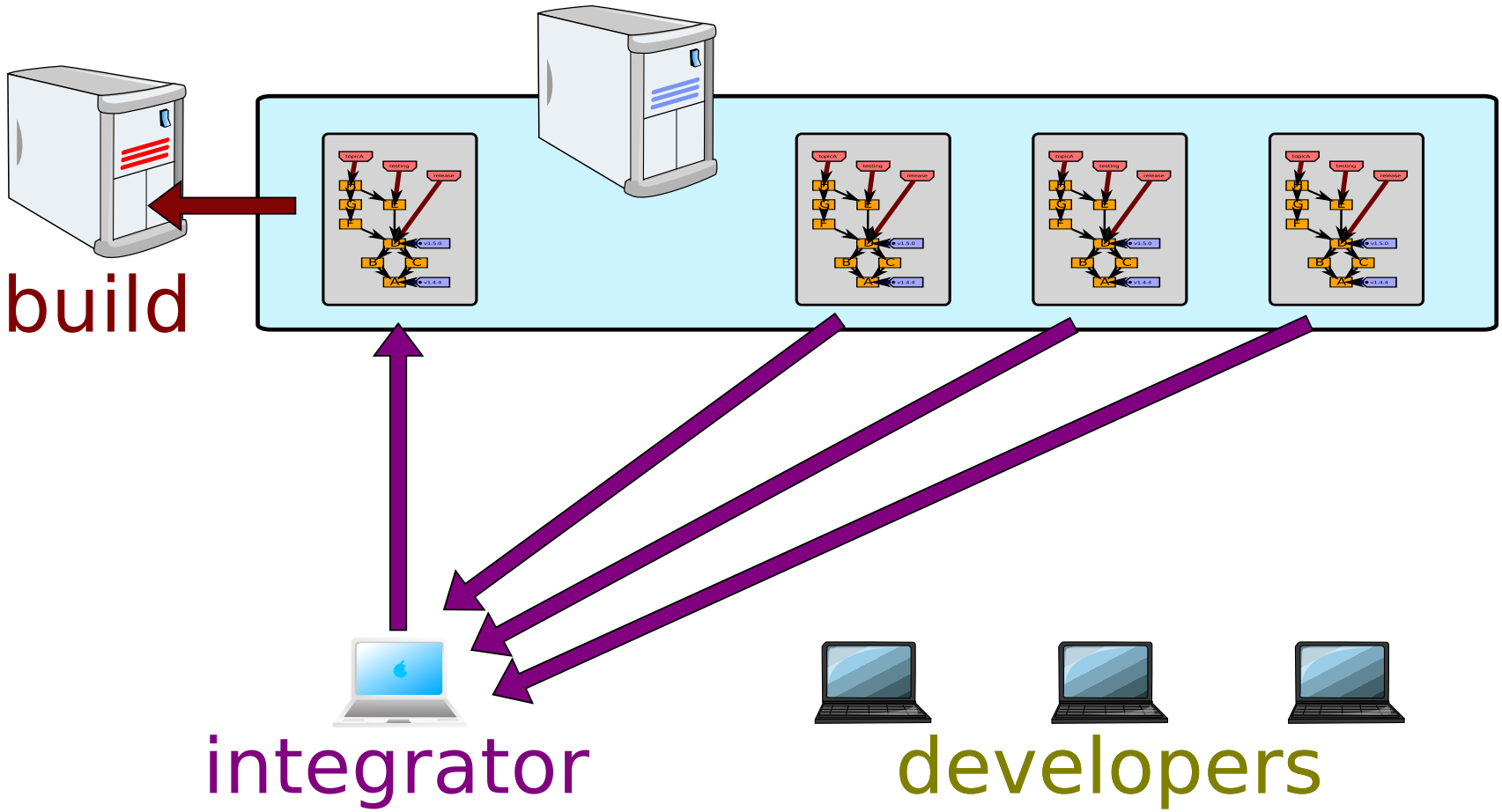
Work can be shared



Integrator downloads and integrates topic branches

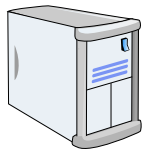


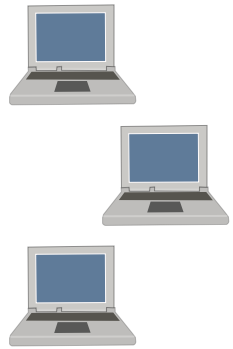
Integrator tests the result and uploads to central repository

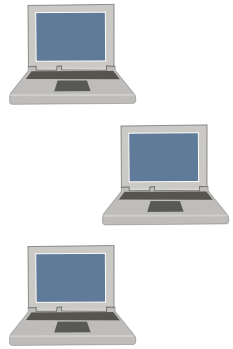


Repository content is automatically built

Dictator and lieutanants





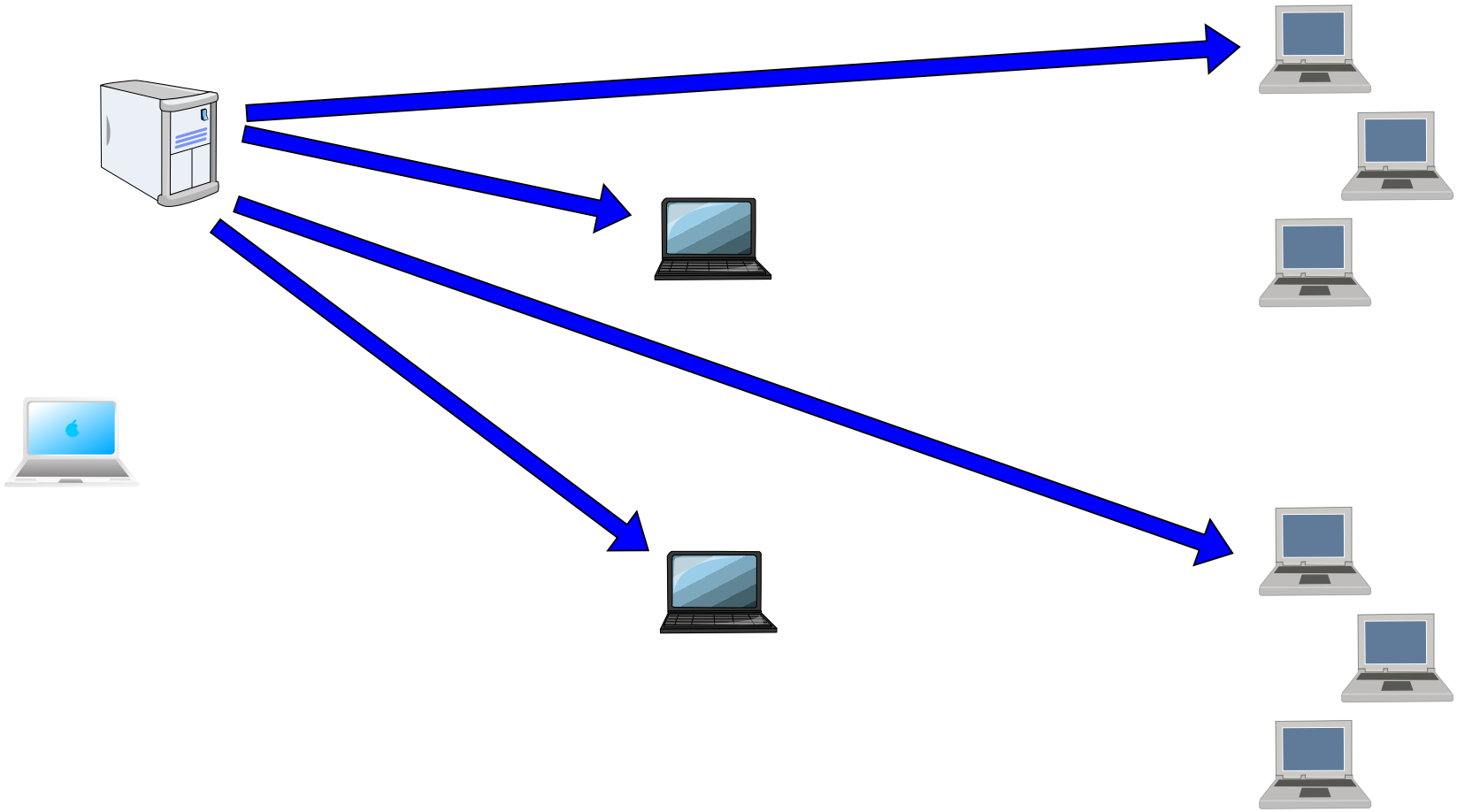



Used for example for development of Linux kernel

Dictator and lieutanants

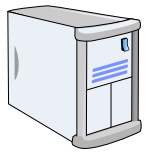
Linus Torvalds created first kernel version

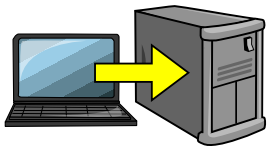
Dictator and lieutanants

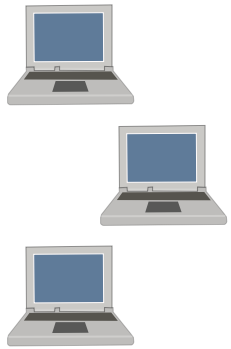


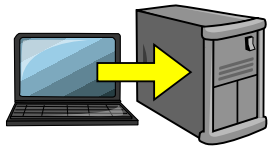
Everyone clones the repository

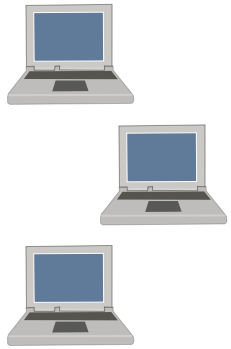
Dictator and lieutanants





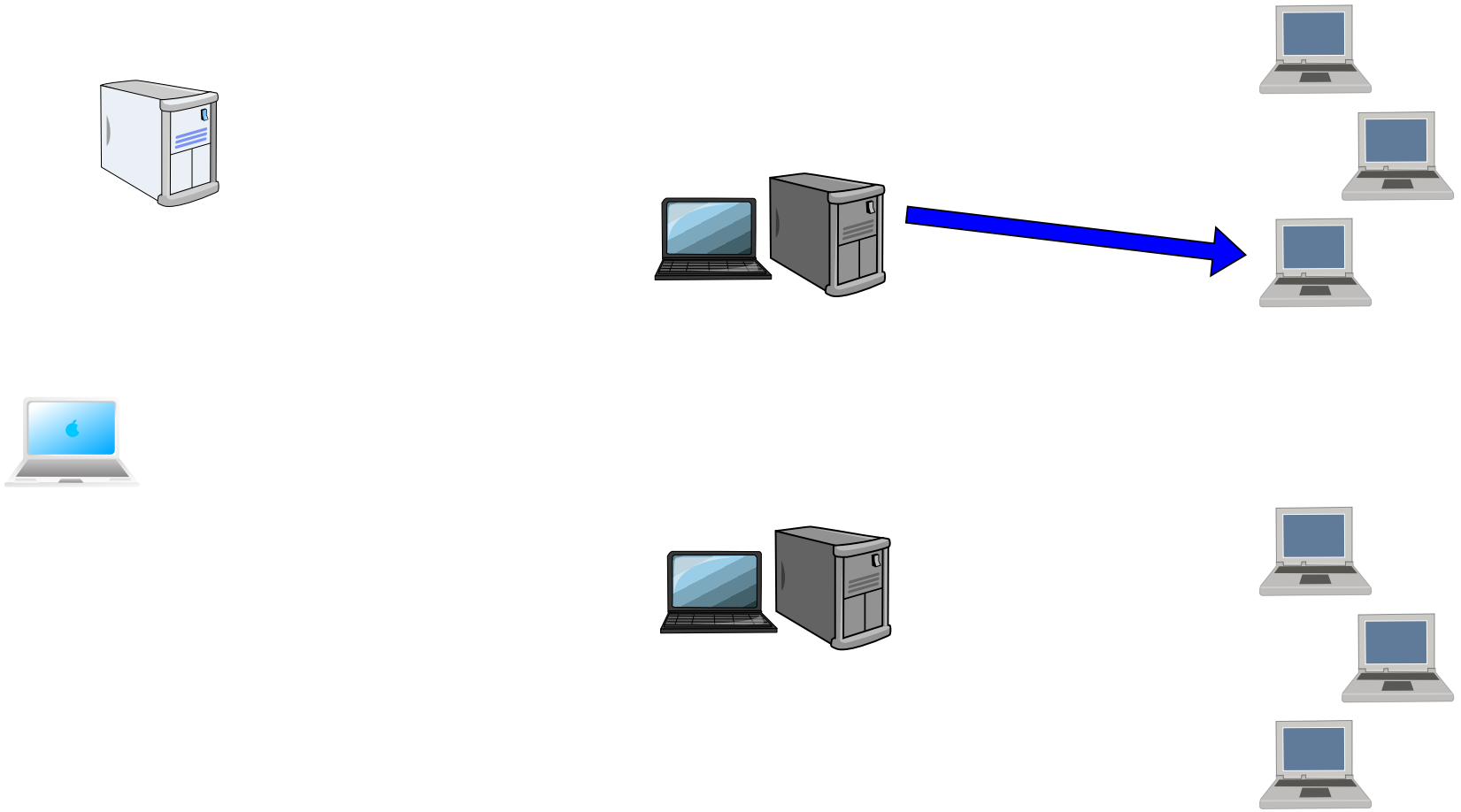






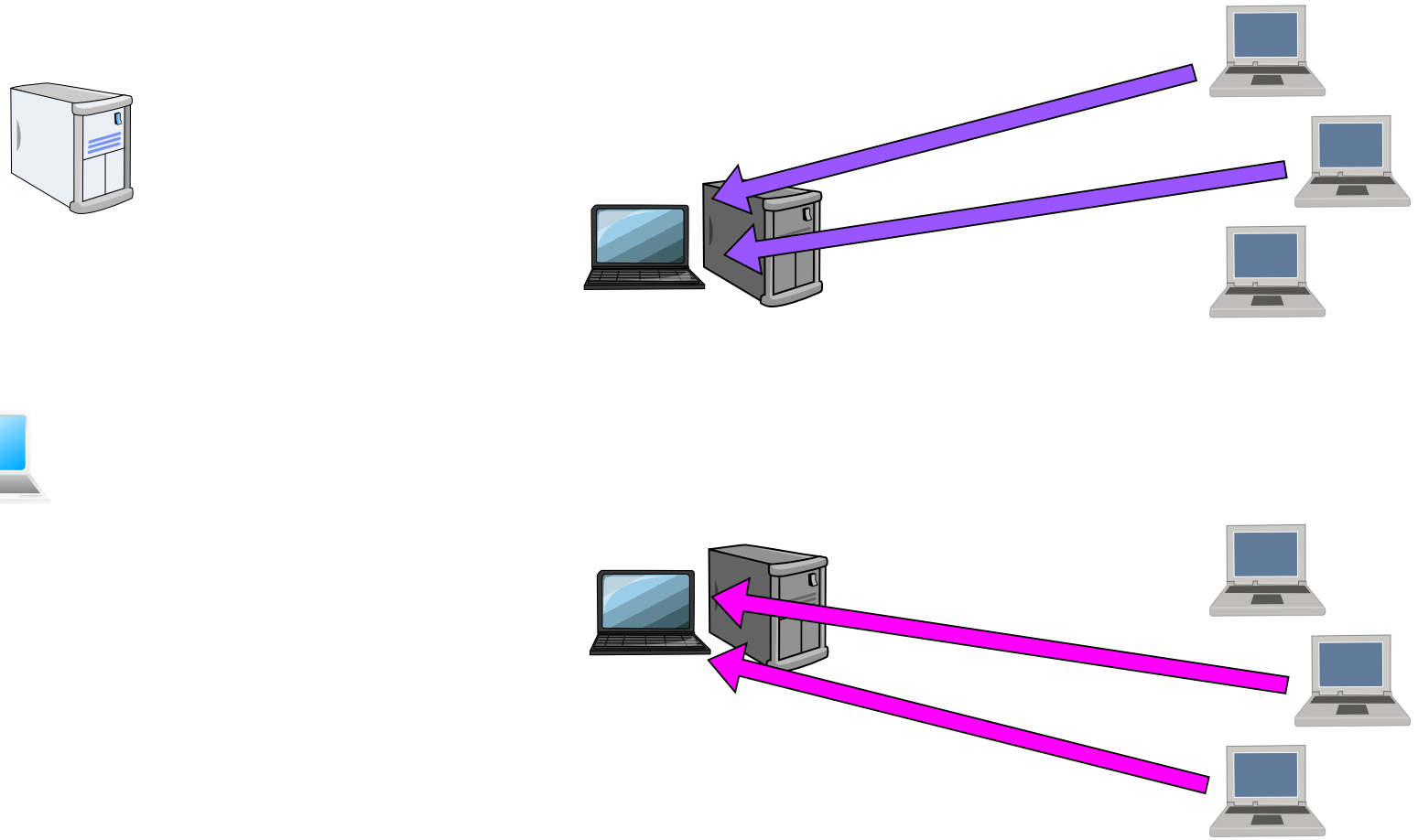
Lieutanants create repositories for subsystems

Dictator and lieutanants



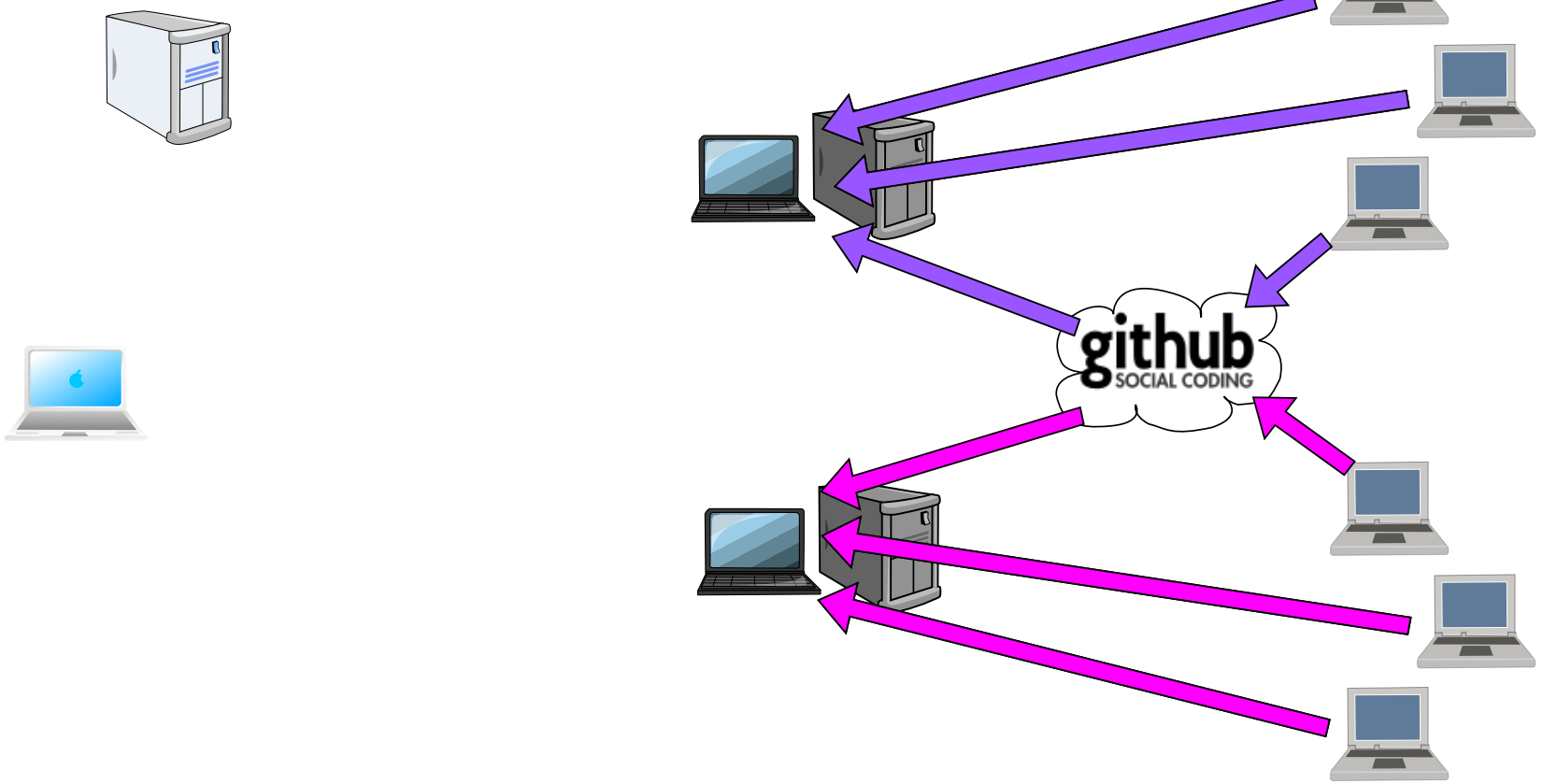
Lieutanants are responsible for individual subsystems

Dictator and lieutanants



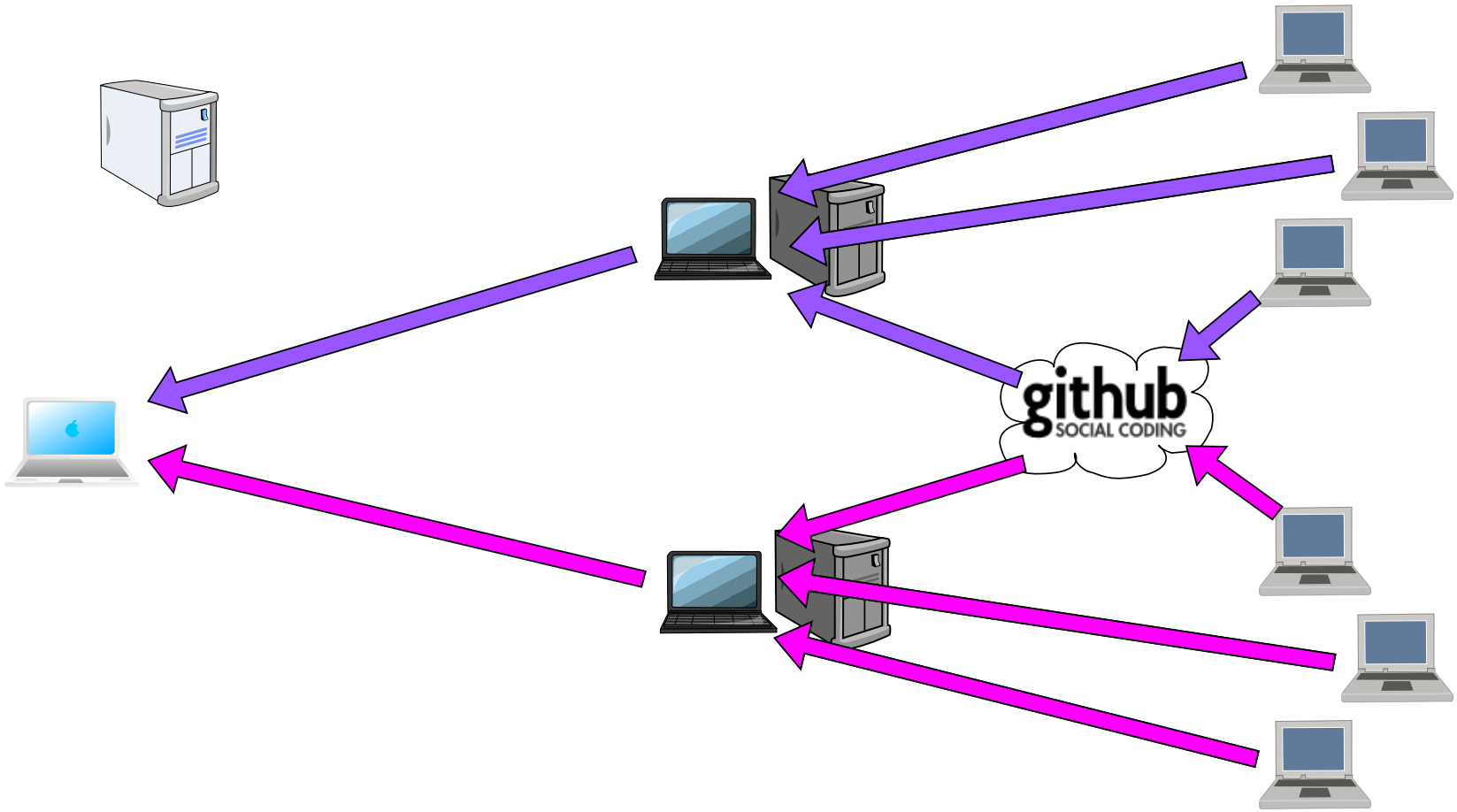
Lieutanants are responsible for individual subsystems

Dictator and lieutanants



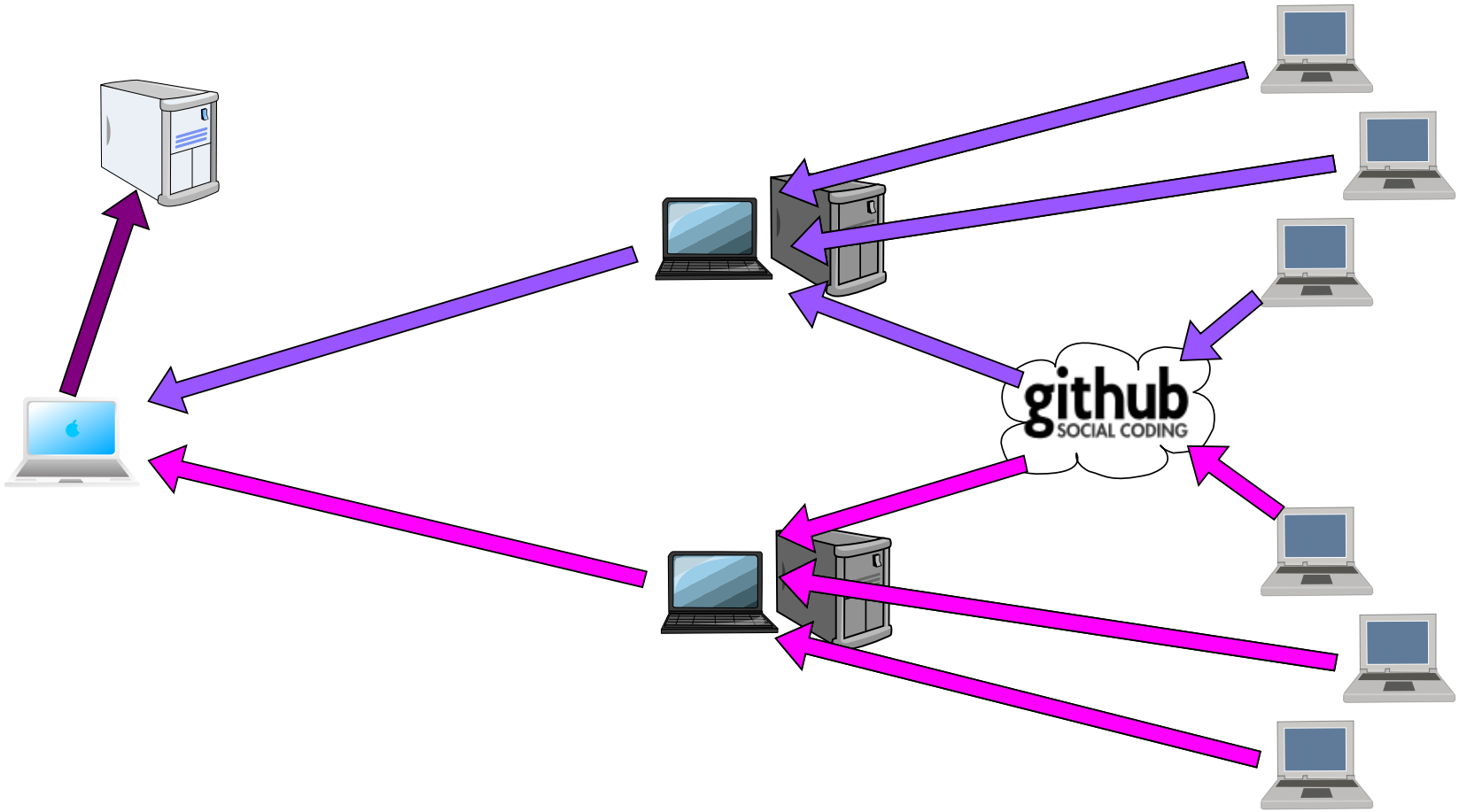
Lieutanants are responsible for individual subsystems

Dictator and lieutanants



Lieutanants send dictator pull requests

Dictator and lieutanants



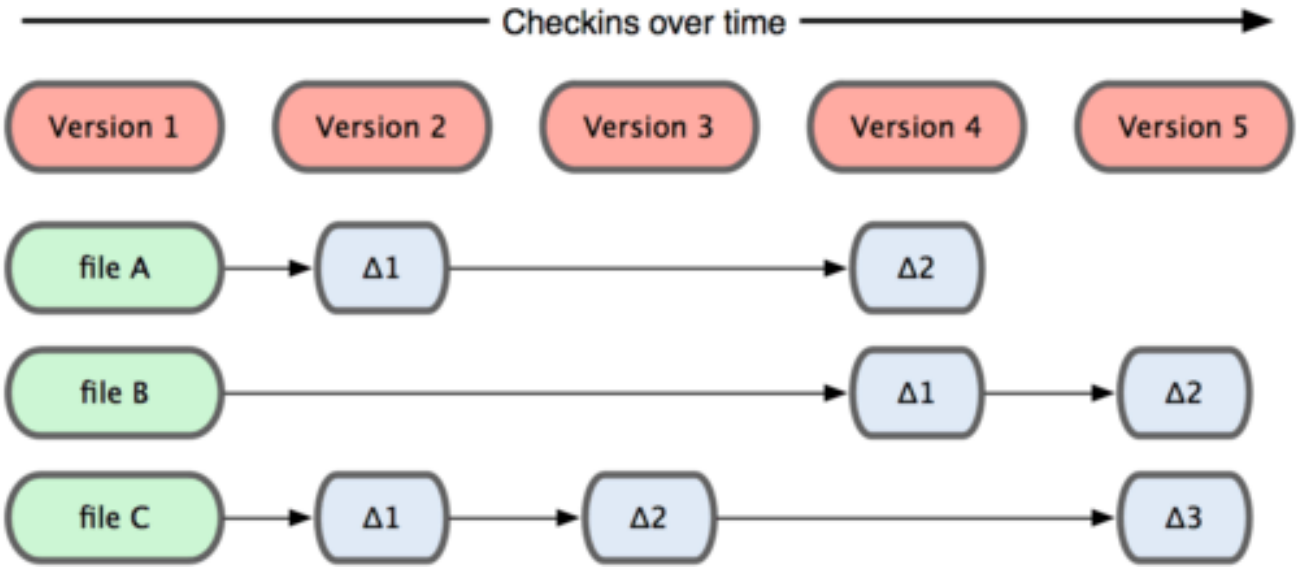
Dictator manages central repository

GIT

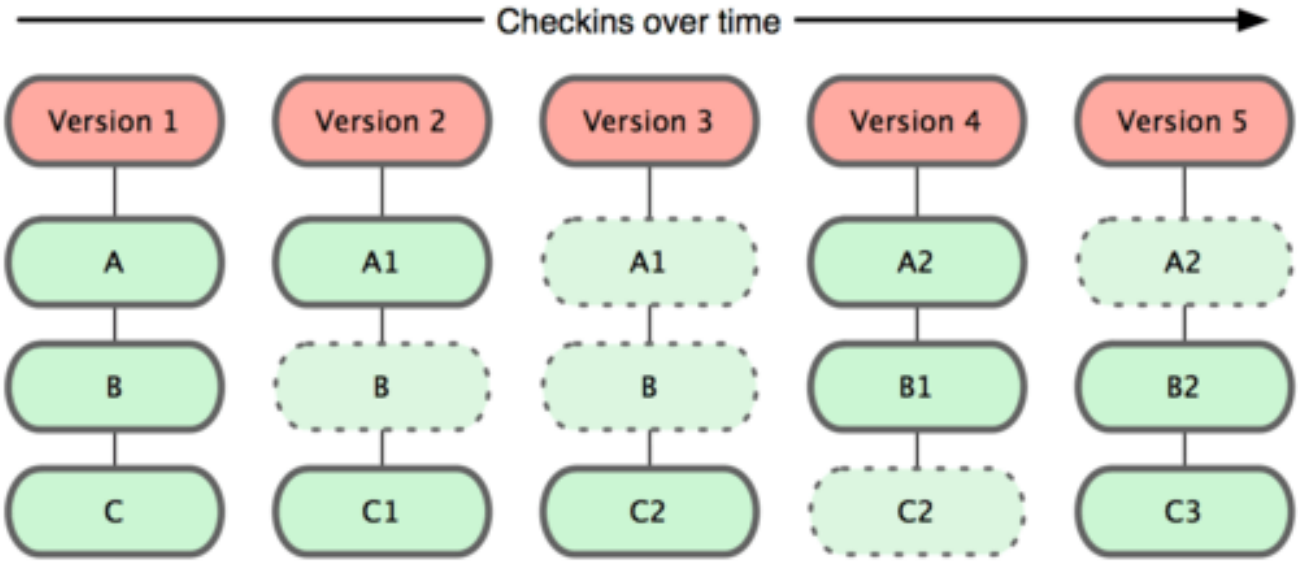
- Distributed version control system
- There is no central repository – each developer has their own copy (including backup)
- For large files, developers can use Git Large File Storage `https://git-lfs.github.com/`
- Ignores empty directories
- Uses snapshots instead of increments

Increments vs. Snapshots

- Increments:



- Snapshots:



- Local (single repository)

```
git config <key> <value>
```

- Global (in user's home directory for all repositories)

```
git config --global <key> <value>
```

- Important settings:

```
git config --global user.name "John Doe"
git config --global user.email "john.doe@example.com"
git config --global color.ui auto
```

Creating a repository

- New repository

  `git init`

  - creates a local repository with working tree in the current directory
  - metadata in directory `.git`
- New shared repository on a server

  `git init --bare`

  - creates a repository without a working tree
  - metadata directly in the repository directory
- Clone existing repository

  `git clone <...>`

  - clones remote repository

git status

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file: my-file-added-using-git-add
        deleted: my-file-deleted-using-git-rm

Changed but not updated:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
  working directory)
        modified: my-modified-file
        deleted: my-file-deleted-by-rm-command

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

git status

```
On branch master
Changes to be committed:
   (use "git reset HEAD <file>..." to unstage)
         new file: my-file-added-using-git-add
         deleted: my-file-deleted-using-git-rm

Changed but not updated:
   (use "git add <file>..." to update what will be committed)
   (use "git checkout -- <file>..." to discard changes in
   working directory)
         modified: my-modified-file
         deleted: my-file-deleted-by-rm-command

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

git status

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file: my-file-added-using-git-add
        deleted: my-file-deleted-using-git-rm

Changed but not updated:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout - <file>..." to discard changes in
  working directory)
        modified: my-modified-file
        deleted: my-file-deleted-by-rm-command

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

git status

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file: my-file-added-using-git-add
        deleted: my-file-deleted-using-git-rm

Changed but not updated:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
  working directory)
        modified: my-modified-file
        deleted: my-file-deleted-by-rm-command

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

Working tree status

`git diff`

- unstaged changes, to be committed
- does not contain untracked files

`git diff --cached`

- staged changes vs last commit

git log

```
commit 1d33b1b1b7530a1168d7c3adc44063bfe32592eb
Author: Random <jrh@example.net>
Date: Sat Sep 12 15:52:20 2009 -0400

    third commit

commit 895740c342fdbc03ce26b6417051ab6f2188a6d0
Author: Random <jrh@example.net>
Date: Sat Sep 12 14:53:20 2009 -0400

    second commit

commit db35c00c63a7e98c4a89e180d317c9fb08e40f4b
Author: Random <jrh@example.net>
Date: Sat Sep 12 10:08:20 2009 -0400

    first commit
```

Creating revisions

- Commit is a single revision
- Identified by a hash (SHA-1)
  - **bf2ca58**1680417f466fbedf35f1a4aa1c4c6c5e9
- Contains a message, author, date and time
- Usual convention for messages:
  - first line = brief summary
    - seen in all history logs
  - second line is empty
  - other lines = more detailed information, for example:
    - reason for these changes
    - solution details
    - side effects

Creating revisions

`git commit [-m message]`

- If no message is specified, git runs a text editor
  - $VISUAL, $EDITOR, vi

`git commit --amend [-m message]`

- When fixing previous commit → merge changes

`git commit -a [-m message]`

- all files, even the unstaged ones
- use carefully – avoid committing of changes of settings for local testing etc.

`git commit <file> [-m message]`

- selected file only

git gui

- only the basic functionality

gitk

- browse history only

gitg:

- not many functions, but clear history and commits

git cola

- clear, automatically refreshes status, advanced search (search commits by changed file etc.)

TortoiseGIT

- integrated to Windows Explorer

GitKraken

- free for open source projects only, advanced functionality is commercial

1. Set up git
   - mainly name and email

     ```
     git config --global user.name "John Doe"
     git config --global user.email "jd@example.com"
     ```
2. Create local repository
   - `mkdir myprogram && cd myprogram`
   - **`git init`**
3. Add all meaningful untracked files – if you have no one, you can create README and `.gitignore`
   - **`git add`** `<file>`
4. Create first revision (commit)
   - **`git commit`** `-m "Initial commit"`
5. Write your program
   - `vim main.c / emacs main.c / subl main.c`
   - **`git add / git rm / git mv`**
6. Create another revision (commit)
   - **`git commit`** `-m "add main.c"`

What should a repository contain

- tool configuration
  - static analysis (linter)
  - testing framework
  - compiler (for example Makefile)
  - ...
- README
- .editorconfig
  - line ending and indentation settings
- .gitignore
  - a list of ignored directories and files

What should a repository not contain

Not everything should be versioned, for example:

- everything, that can be generated
  - binaries
  - generated documentation
  - PDF generated from LaTeX source
- local configuration (specific to one developer)
  - IDE settings
  - access to testing database
- temporary files
- logs

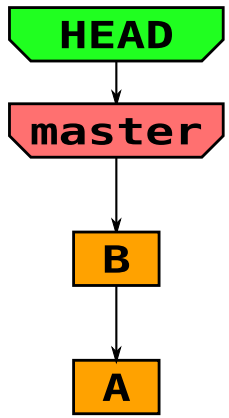
- a list of directories and files to be ignored by Git
- https://github.com/github/gitignore

```
# ignore .a files
*.a

# with the exception of lib.a
!lib.a

# ignore TODO file in this directory, but not in its subdirectories
/TODO

# ignore everything in build/ directory
build/

# ignore doc/notes.txt, except for doc/server/arch.txt
doc/*.txt

# ignore all pdf files in doc/ directory, subdirectories included
doc/**/*.pdf
```

- Project title
- Authors
- Brief functionality description
- License
- Installation instructions
- How to contribute (primarily open source)
- Anything else that makes sense ...

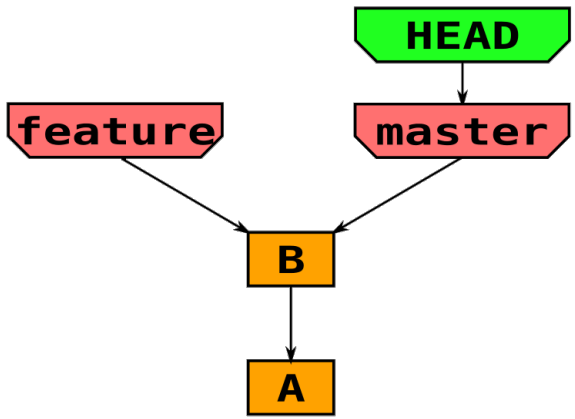
Branching in GIT

Current repository



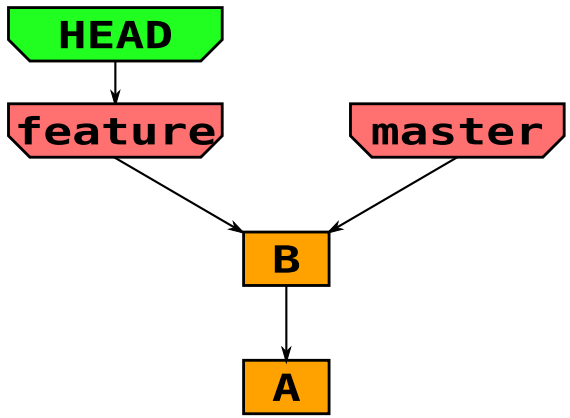
Create branch for feature:

```
git branch feature
```

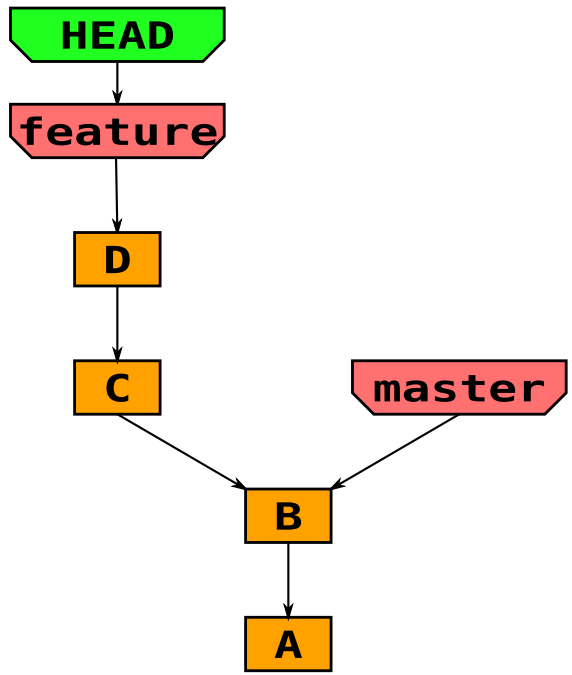


Checkout to feature branch:

```
git checkout feature
```

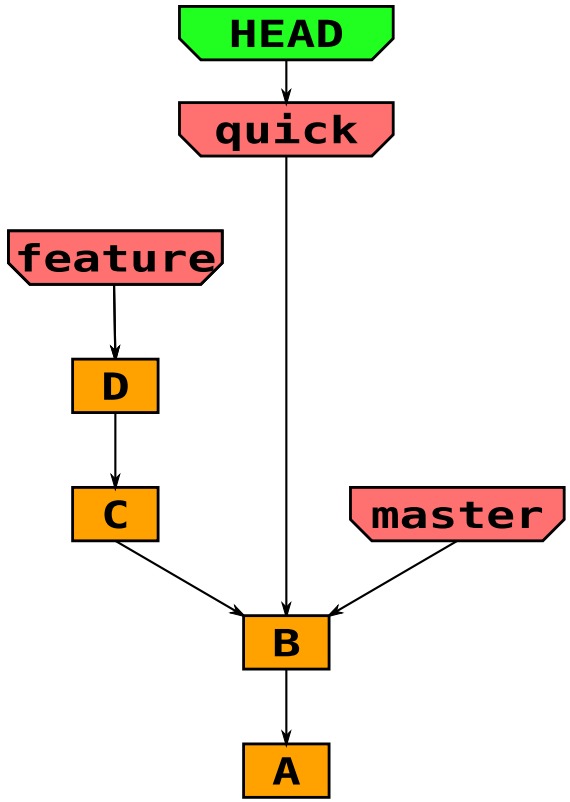


Develop feature.

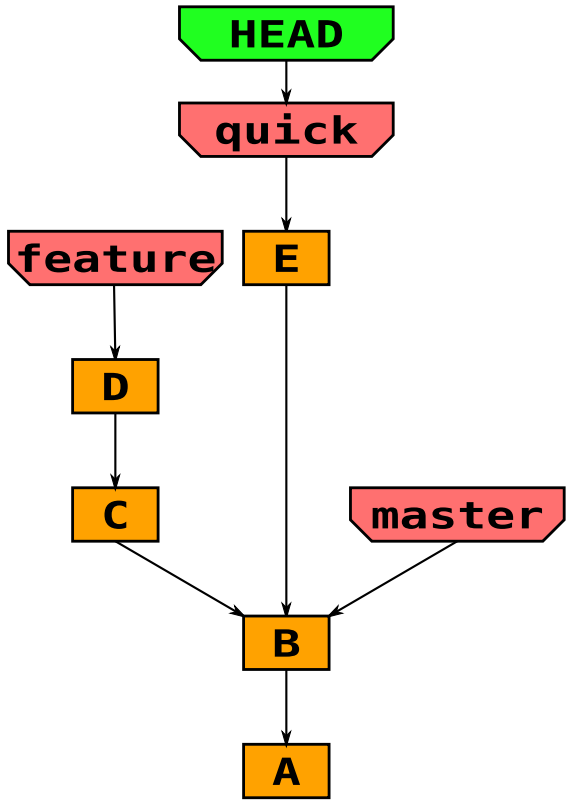


We've identified a bug,
that hinders the development,
and so we fix it now.
We will create a branch:

```
git checkout master
git checkout -b quick
```

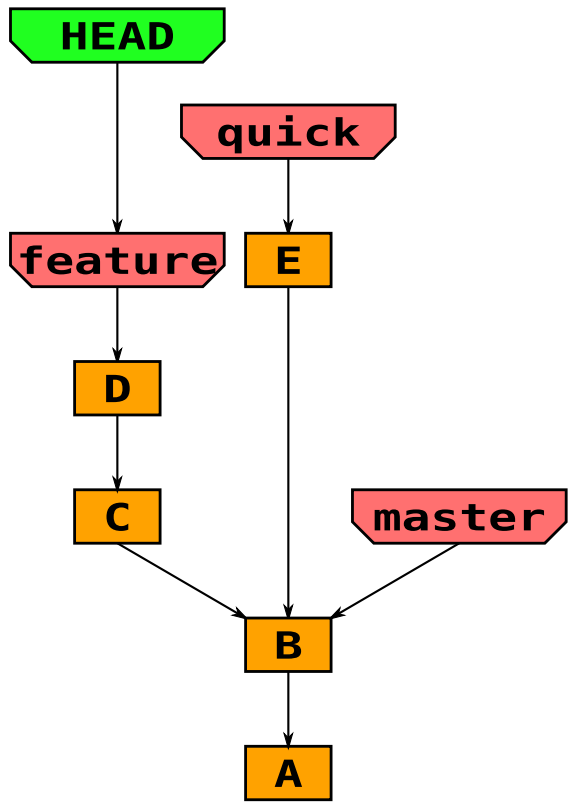


Quick bugfix.



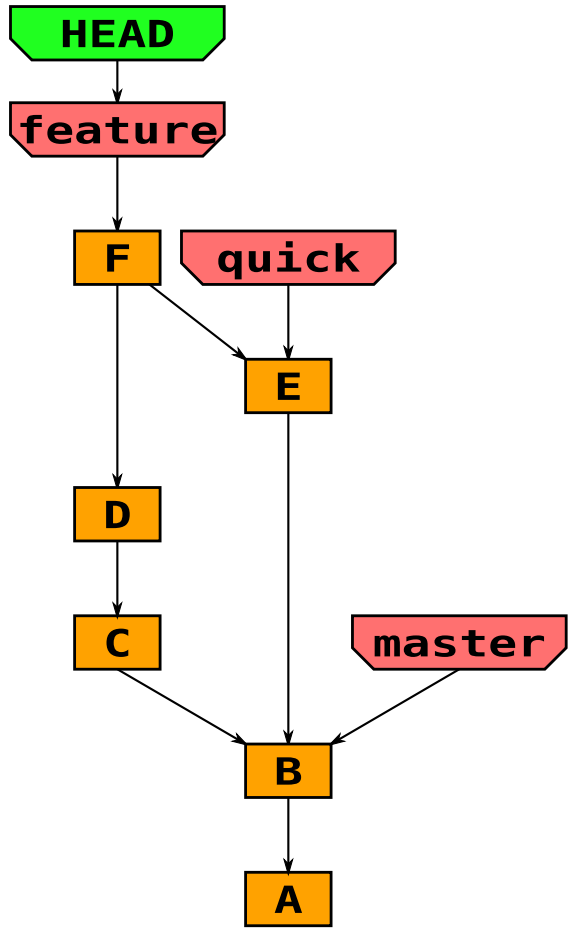
Back to developing feature:

```
git checkout feature
```



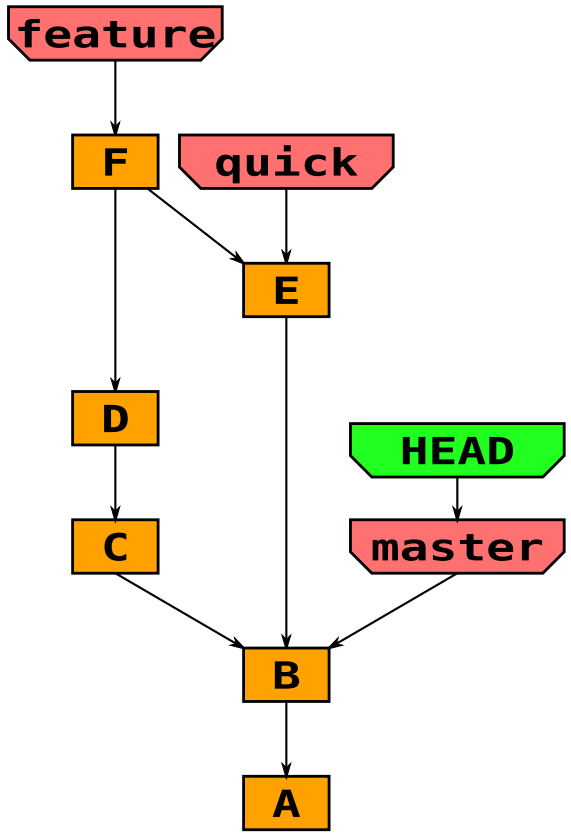
Merge bugfix:

```
git merge quick
```



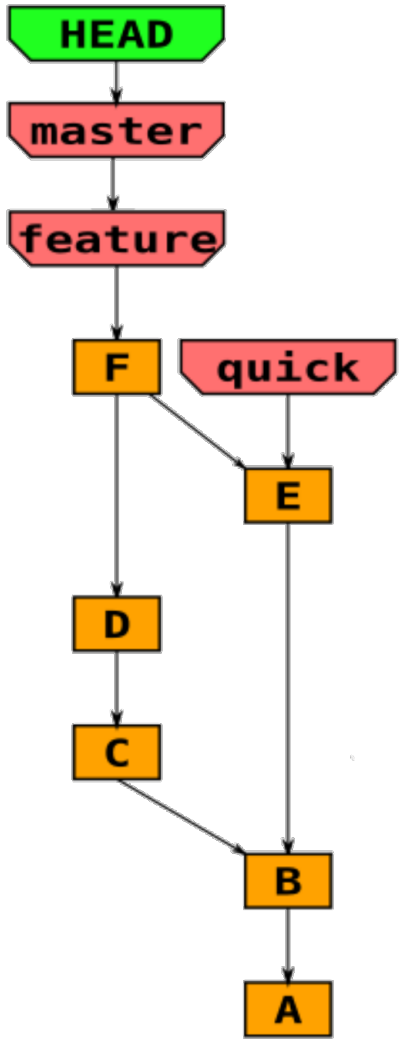
Feature is finished
and tested.
Now we will integrate it.
We will switch to master:

```
git checkout master
```



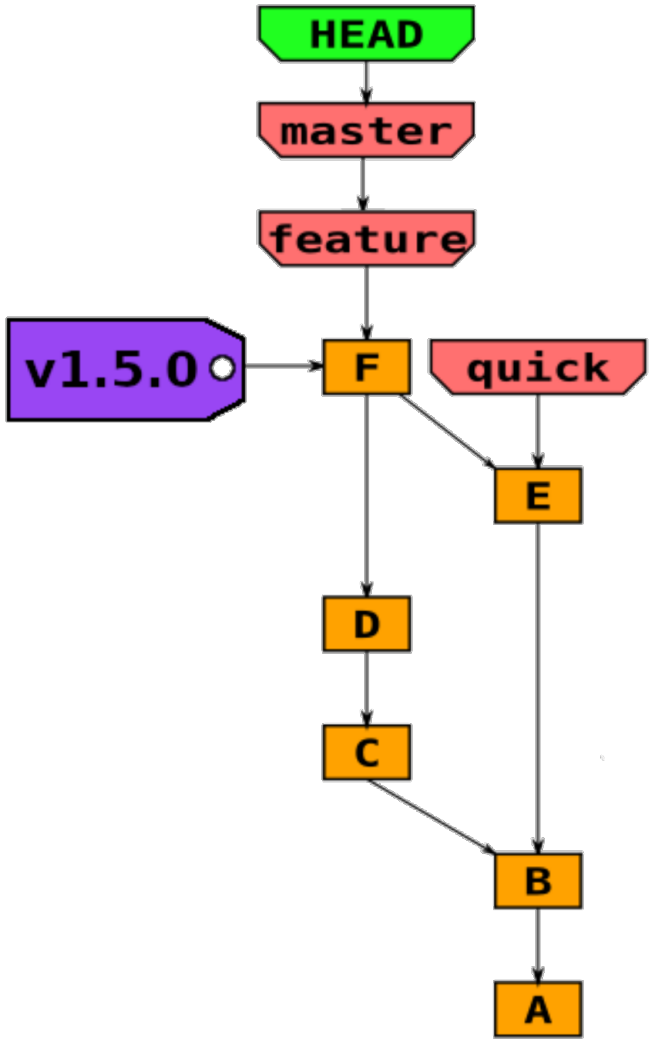
And now we can merge our feature:

```
git merge feature
```



We can also create
a new release for
our customer.
We will mark it with tag:

```
git tag v1.5.0
```



Two merge approaches

Fast-forward

- merged branches are direct successors of the HEAD
- simple change of pointer

Merge commit

- merged branches are not direct successors of HEAD
- it is necessary to identify common predecessor
- can create conflicts

`git checkout <destination>`

Can switch branch to practically anywhere

- `git checkout` **`branch`**
- `git checkout` **`a8d621`** (creates a *detached head* state)
- `git checkout` **`origin/master`** (*detached head* as well)

It can also revert changes

- `git checkout` **`main.c`**
- `git checkout` **`subdir`**
- `git checkout` **`.`** **(dangerous)**

Revert state

`git reset --soft <revision>`

- reverts active branch state to specified revision
- revision differences will be staged

`git reset [--mixed] <revision>`

- --soft + flush staged changes (but changes are preserved)

`git reset --hard <revision>`

- --mixed + nullifies all changes in **working tree**
- **dangerous**, difficult to revert

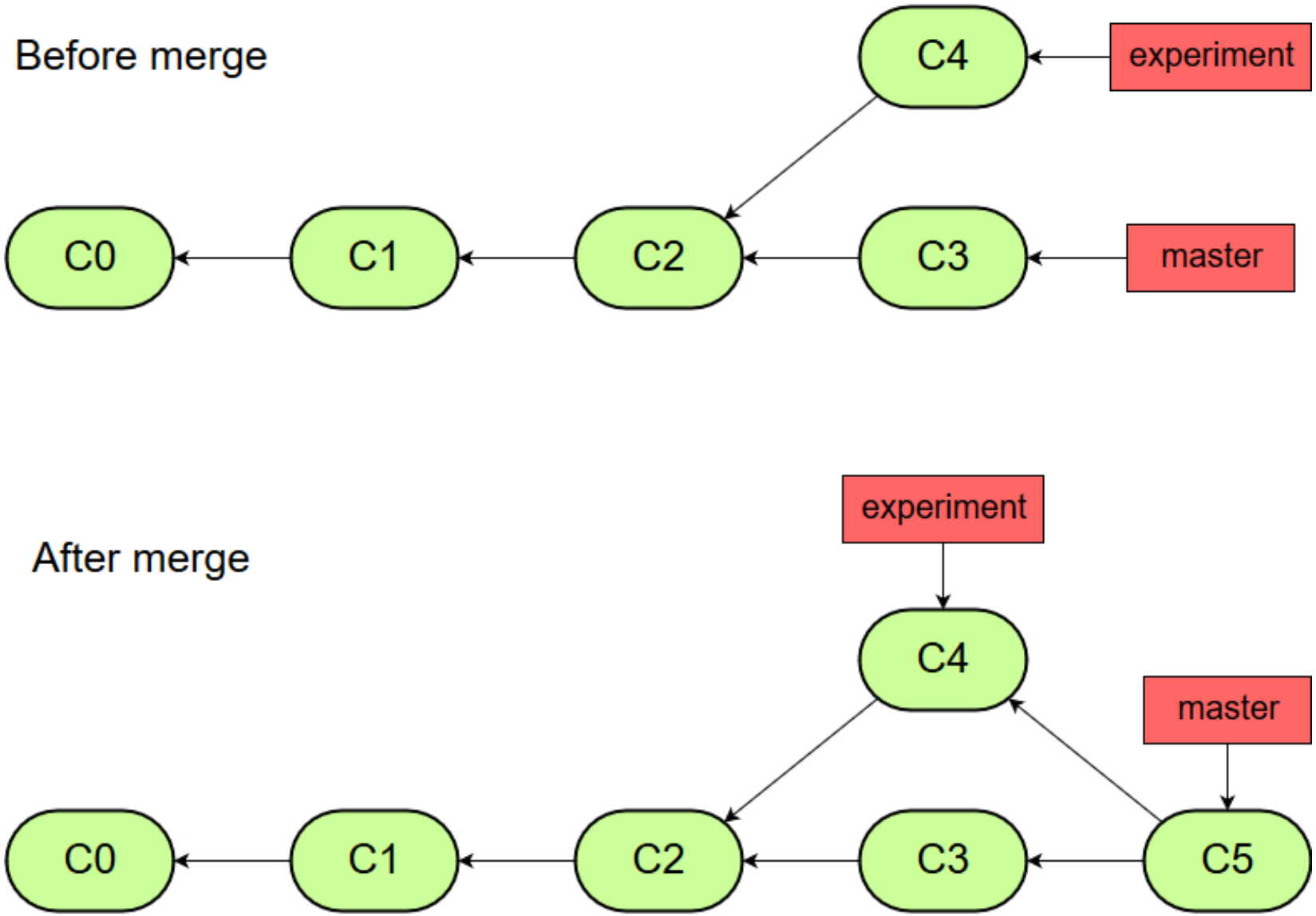
Revert changes

`git revert <commit>`

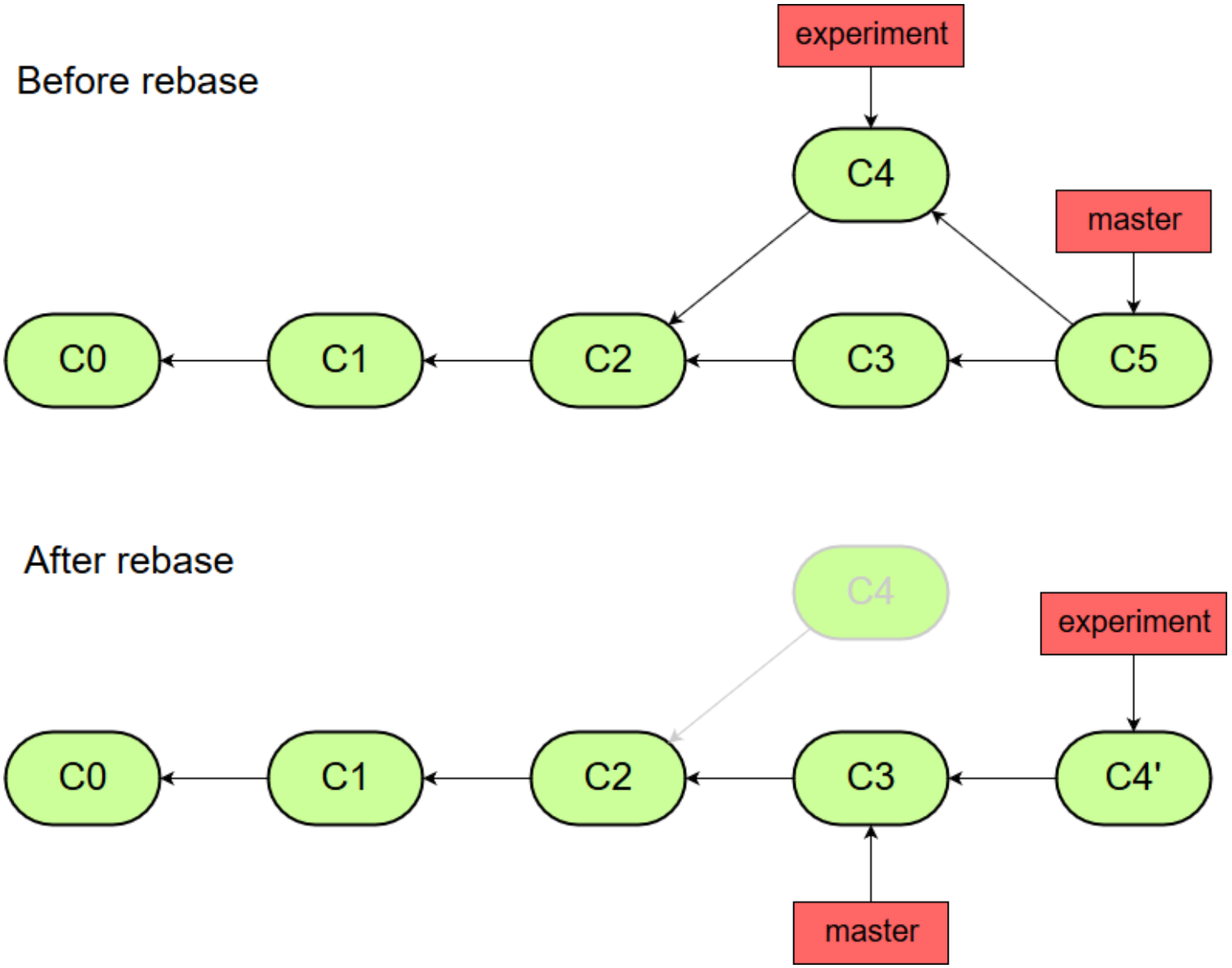
- Creates a commit that reverts changes made by another commit.
- It is performed with a clean working tree.
- With the `--no-commit` parameter, it does not create a commit, but only applies the changes to the working tree.

- **merge** command combines the changes of two branches.
- **rebase** command takes the changes introduced by the commits of one branch and applies them on the HEAD of the other branch.
- It creates a cleaner history, but it is dangerous.
  - There are automatically created commits based on comparing of branches.
  - It can break the result in some circumstances.

Merge example

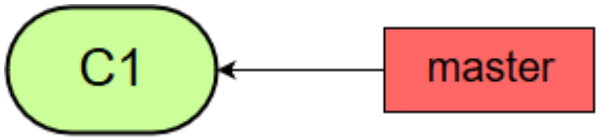


Rebase example



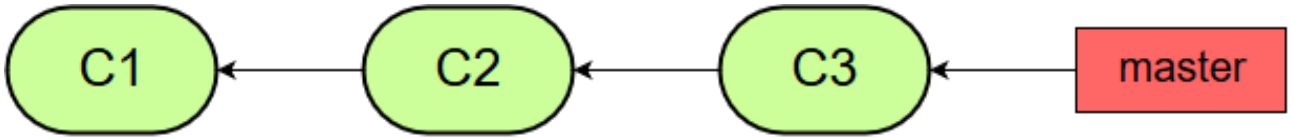
- Do not perform a rebase for commits, that were pushed to a remote repository.
  - It only makes the repository history look like a mess and your colleagues will not be happy about it.
- You should use rebase primarily to clean up local history before pushing to a remote repository.

Developer A clones the repository and starts working

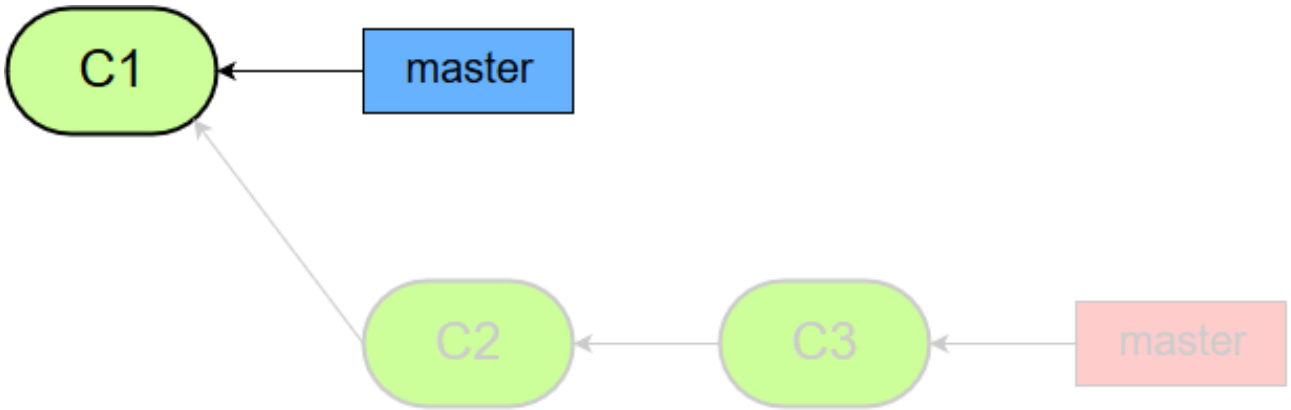


Developer A creates some commits . . .

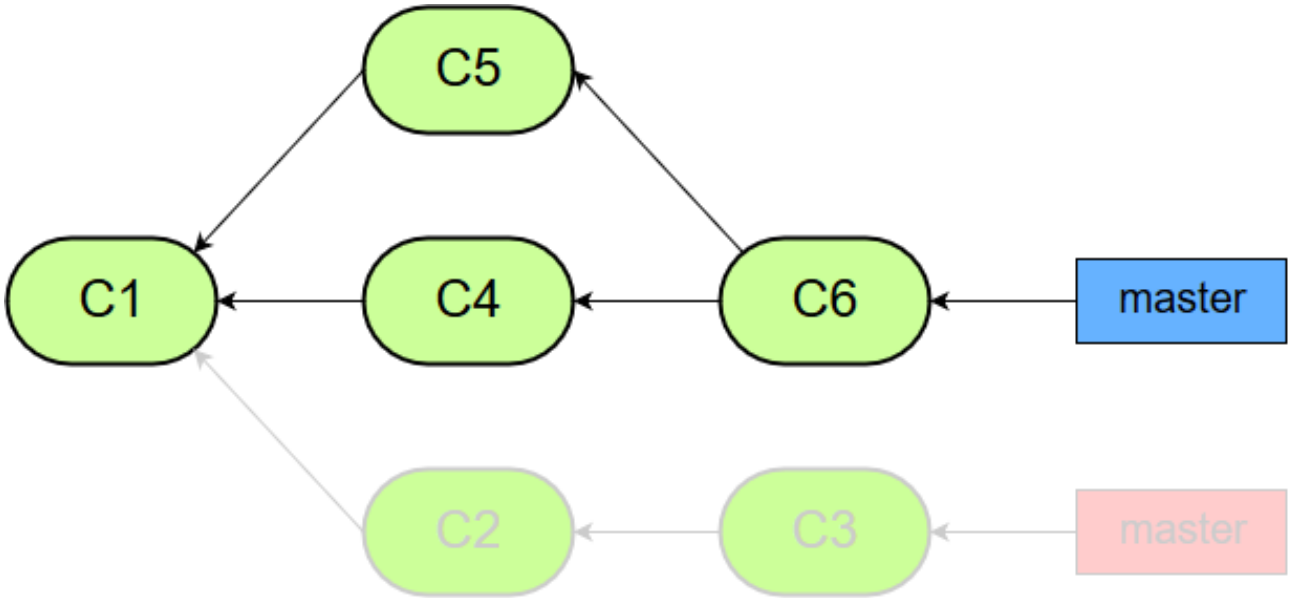
Developer A creates some commits



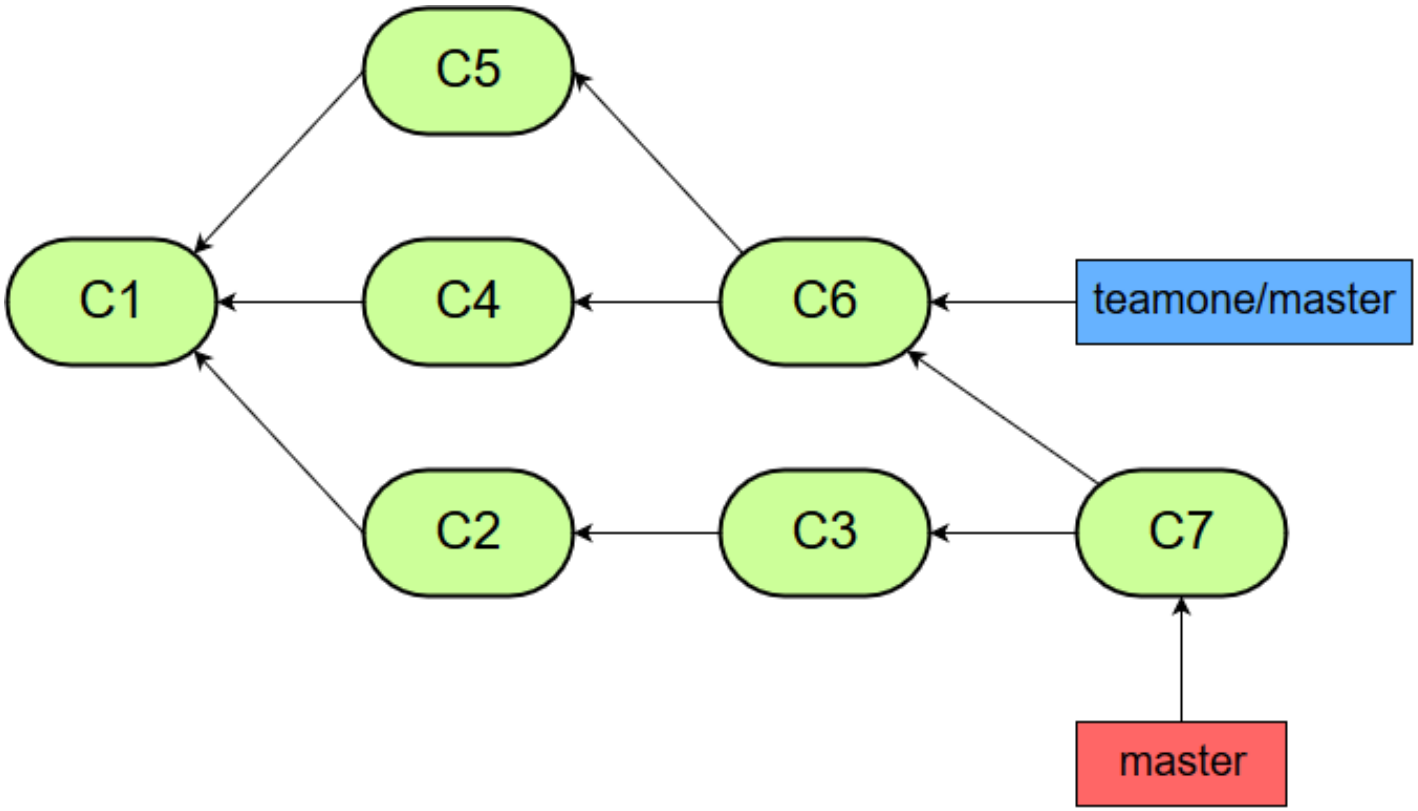
Developer B clones the repository and starts working . . .



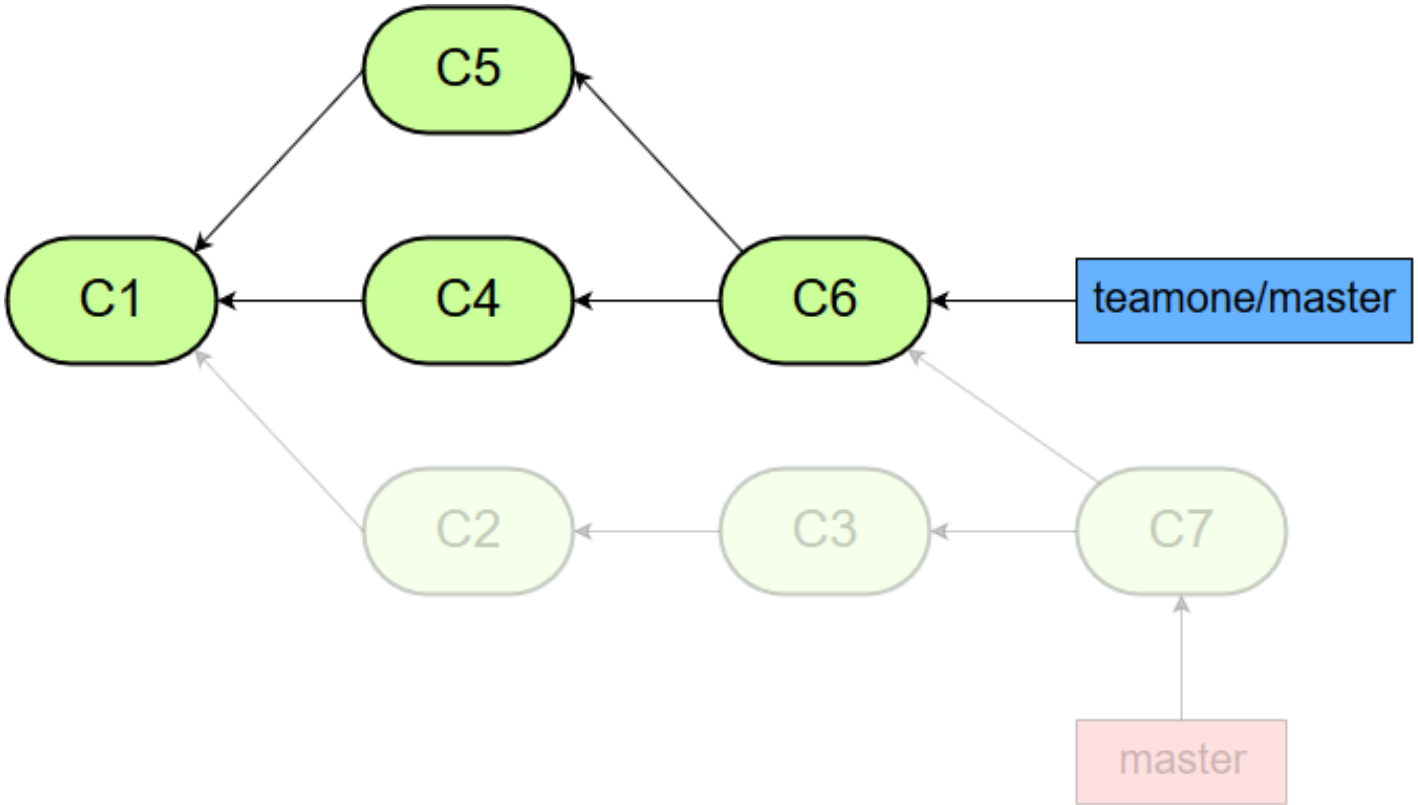
Developer B works with branches and pushes to the master



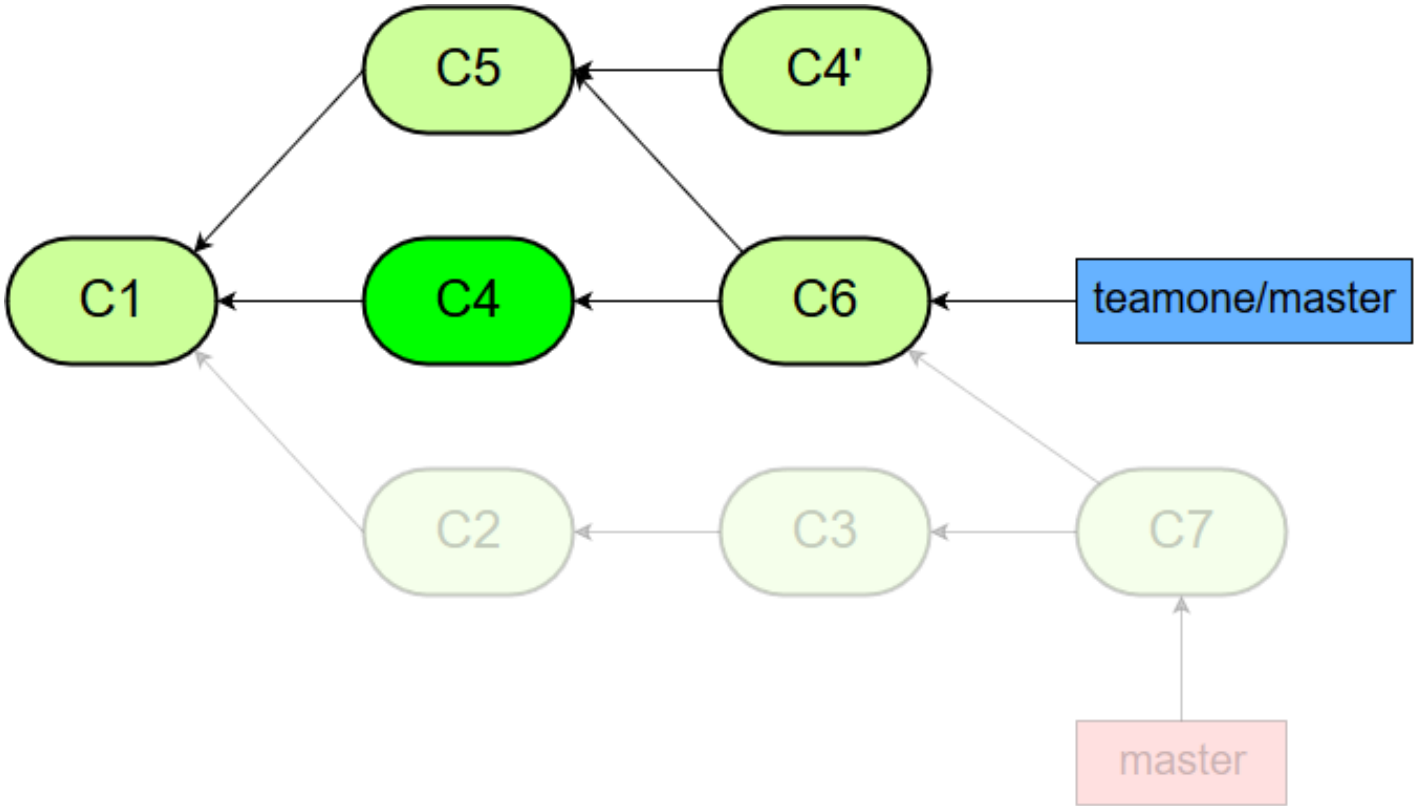
Developer A will pull the master



Back to view of B

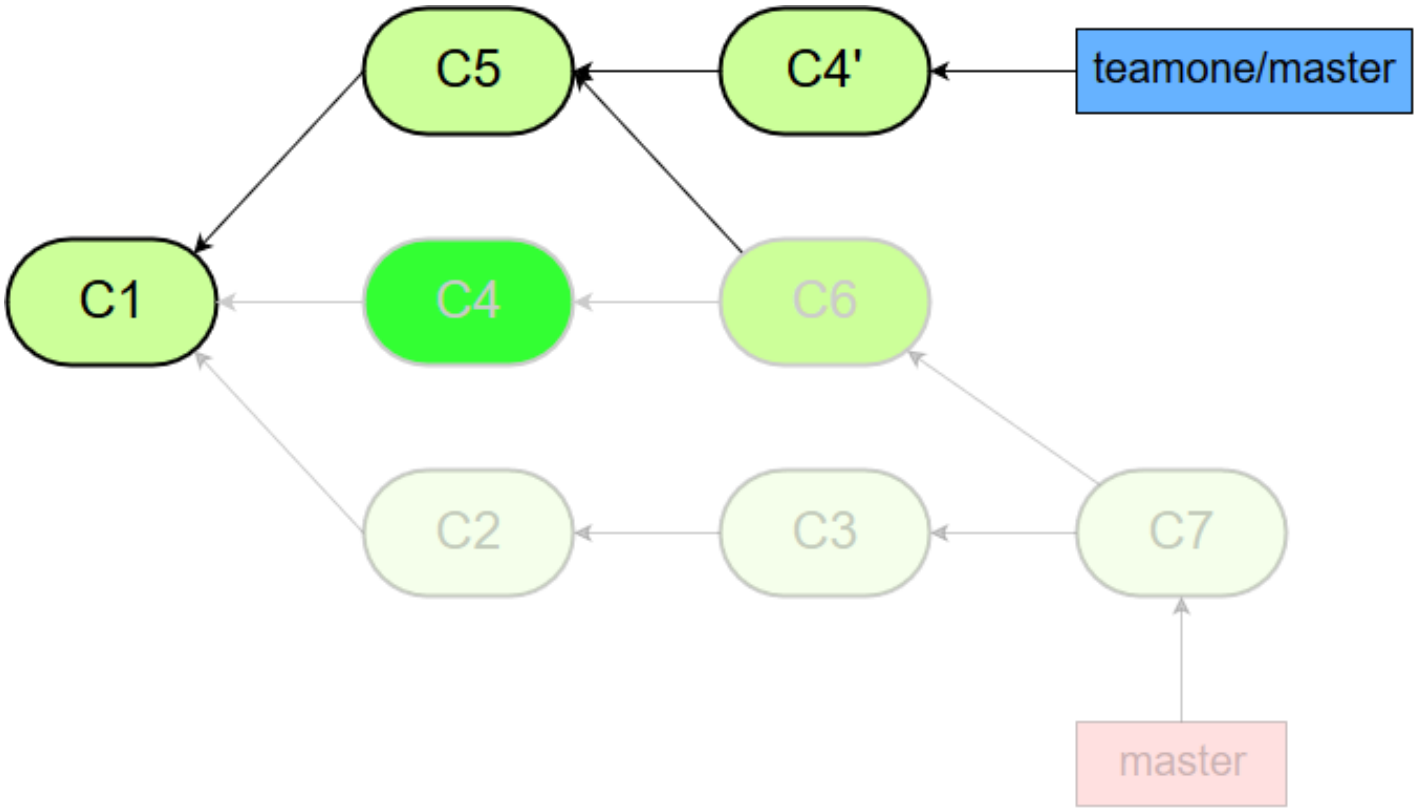


B will do rebase and commits C4 and C6 will no longer exist

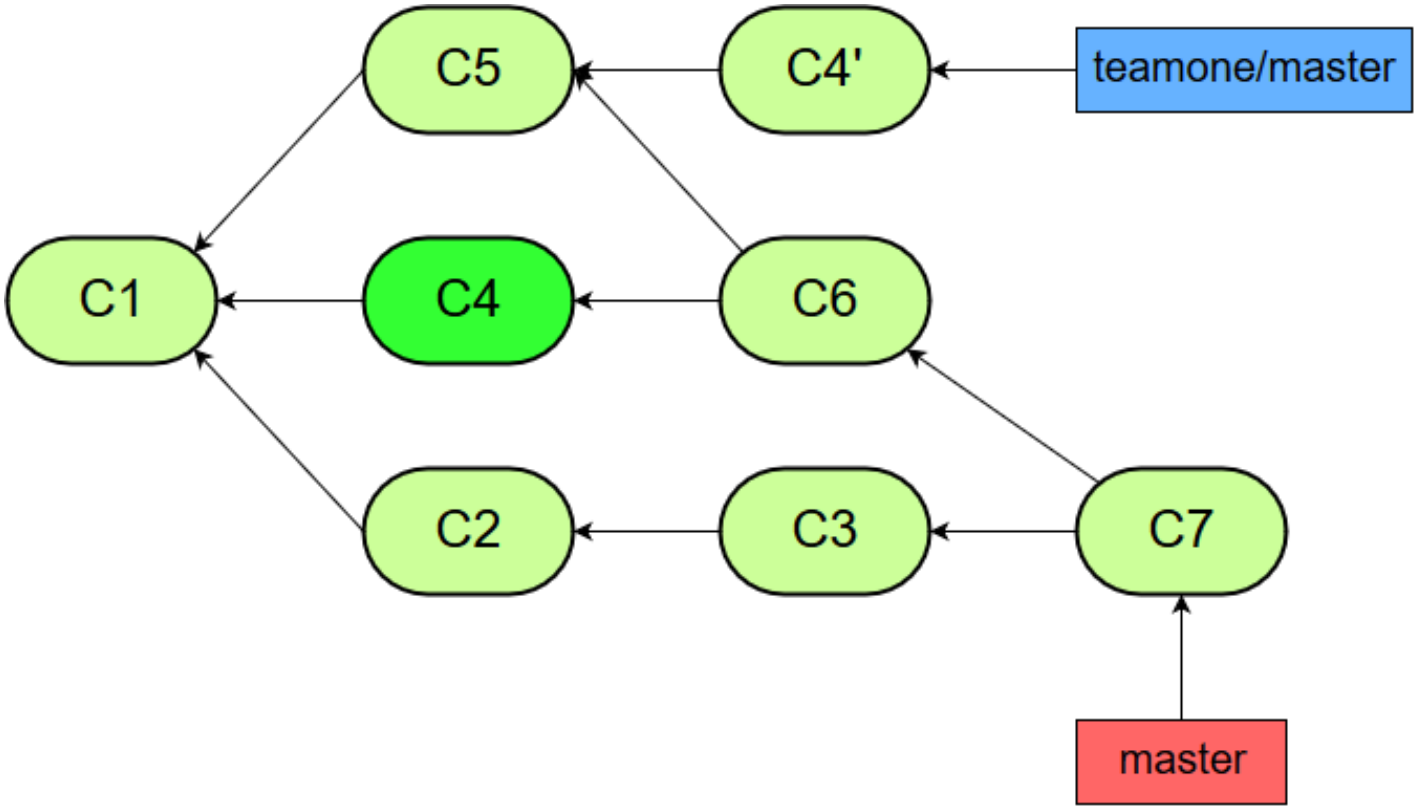


Rebase problem

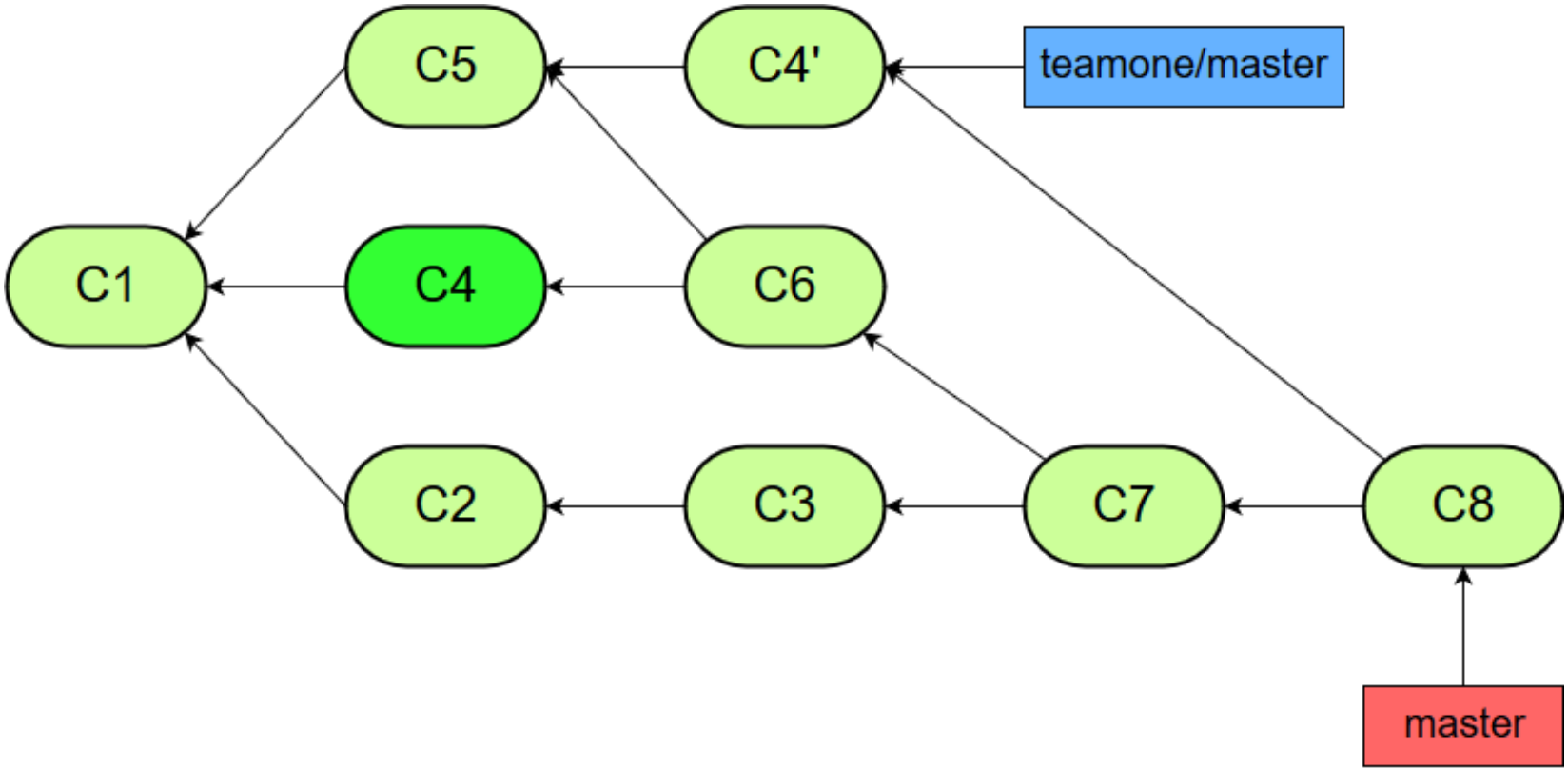
B will push it



But A still have it . . .



. . . and if A will merge the master, he have C4 dupplicated



Remote repositories

Remote repositories

- Essential for teamwork
  - Each programmer would only have their own copy without them
- There can be multiple remote repositories
- Main remote repository is called **origin**
- When cloning a remote repository, it is automatically set as **origin**
- Remote repository can also be added later:

```
git remote add origin <source>
```

Source can be:

ssh

- git@github.com:example/example.git
- ssh://example.com/git/example.git

http

- https://example.com/example.git

directory on a hard drive

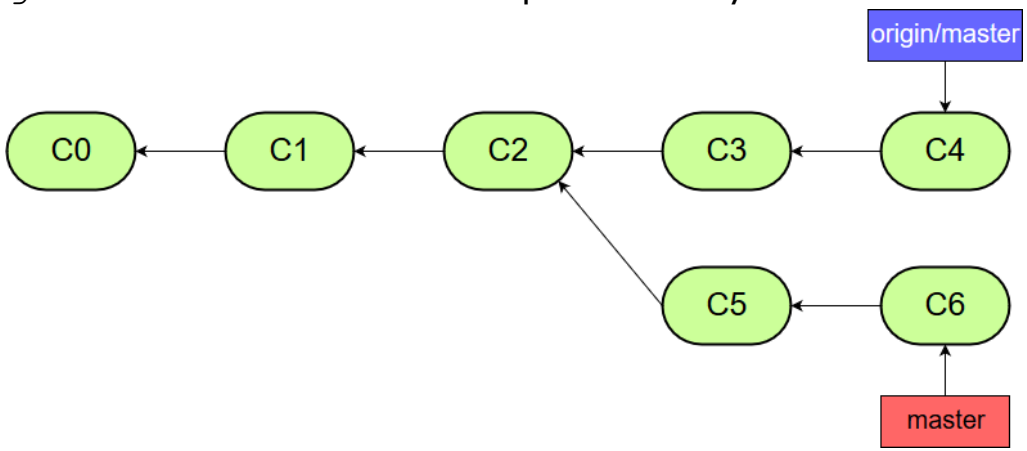
- /opt/repos/example.git

git protocol

- git://example.com/example.git

Remote repositories

- Download updates from repository:
  - `git fetch` **`origin`**
- Branches from remote repository are separated!
  - **origin** contains branch **master**
  - **developer** has branches **master** and **origin/master**
  - it is up to the developer to merge them
    - `git merge` creates merge commit
    - `git rebase` creates simpler history



- Shortcut:
  `git pull` **`origin master`**
  1. `git fetch` **`origin`**
  2. `git merge` **`origin master`**

Remote repositories

- Send changes to remote repository:
  - `git push` **`origin master`**
- Your local branches need to be up-to-date, otherwise the action results in an error (non-fast-forward updates were rejected). You should update using `git pull`:
  - `git pull origin master`
- Tags are sent individually on branches:
  - `git push` **`origin v0.8`**
  - `git push` **`origin --tags`**

Remote repositories – Fetch

- Fetch downloads and updates objects and references from another repository
- Fetch does not create local branches automatically

- **origin** contains **master** and **issue123**
- **developer** has branches **master** and **origin/master**

- After downloading changes (`git fetch origin`), the **developer** has branches **master**, **origin/master** and **origin/issue123**

- Before he can start working, he needs to create local branch:
  - `git checkout` **`-b issue123 origin/issue123`**
  - **issue123** tracks **origin/issue123**

Tracking branch

- Local branch can track a remote branch
  - typically, both branches have the same name (branch and origin/branch)
  - "branch is tracking origin/branch"

- Pull and push works even without arguments
- Tracking is set automatically:
  - `git clone` (master tracks origin/master)
  - `git checkout -b branch` **`origin/branch`**
  - `git checkout --track origin/branch`
- Additionally:
  - `git branch -u origin/master`

1. Set up git
   - mainly name and email
     ```
     git config --global user.name "John Doe"
     git config --global user.email "jd@example.com"
     ```
2. Create central repository (the first developer only)
   - `ssh dytrych@ivs.fit.vutbr.cz`
   - `cd /<shared directory>`
   - **`git init --bare`**
3. Create local clone
   - **`git clone`** `ssh://dytrych@ivs.fit.vutbr.cz/<shared directory>`
4. Create README and `.gitignore` and add them (the first developer only)
   - **`git add`** `<file>`
5. Create first revision (commit) (the first developer only)
   - **`git commit`** `-m "Initial commit"`
6. Push first revision (commit) (the first developer only)
   - **`git push`** `origin master`

7. Create new branch
   - `git branch myFeature`
   - `git checkout myFeature`
8. Work on your feature (write your program)
   - `vim main.c / emacs main.c / subl main.c`
   - `git add / git rm / git mv`
9. Create another revision (commit)
   - `git commit -m "add main.c"`
10. Push your branch
    - `git push origin myFeature`
11. Integrate
    - `git pull origin master`
    - `git checkout master`
    - `git pull origin master`
    - `git merge myFeature`
    - `git pull origin master`
    - `git push origin master`

Push local repository to new empty remote

- **`git remote add`** `origin ssh://dytrych@ivs.fit.vutbr.cz/<shared>`
- **`git push`** `-u origin master`
  - `-u` adds upstream (tracking) reference

- `https://github.com/`
  Free private repositories for students
  `https://education.github.com/pack`
  - Project management, isue tracking, pull requests, wiki, GitHub Pages (web managed using GIT), GitHub Gist (file sharing), ...
  - A social network
- `https://bitbucket.org/`
  - Free private repositories for up to 5 users.
  - Trello / Jira integration for issue tracking, ...
- `https://about.gitlab.com/`
  - Open-source (GitLab Community Edition)

Other useful features

Finding out the author of the change

`git blame <file>`

- For each line of the file, it lists information about the last commit that changed it.

`git blame <file> -L 40,45`

`git blame <file> -L 40,+5`

`git blame <file> -L /regexp/`

`git blame <file> -L :function`

- `-L` allows you to limit the range.

- Stash is used before switching to another branch, if we do not want to commit (save changes to the stash and apply when returning).

`git stash`

- Saves changes in the working tree to the stack (one set of changes in the stack).

`git stash pop`

- Removes a set of changes from the stack and applies them to the working tree.

`git stash apply`

- Applies the change set from the top of the stack to the working tree, but keeps it in the stack.

`git stash apply <n>`

- Applies a set of changes from a given position in the stack to the working tree, but keeps it in the stack.

`git stash drop <n>`

- Discards a set of changes on a given position in the stack.
- Without specifying a position, it discards the set from the top of the stack – needed, for example, after an error when applying changes (conflict).

`git stash list`

- Lists the contents of the stack.

```
stash@{0}: gitg auto stash ...
stash@{1}: WIP on master ...
```

(Work In Progress)

`git stash branch`

- Creates a new branch from the change set at the top of the stack.
- Commit where the change set was created + the contents of the change set.

`git cherry-pick <commit>`

- Apply the selected commit to the current branch.
- With the `--edit` parameter, it will ask for a new commit message.
- With the `--no-commit` parameter, it does not create commit, but only applies the changes to the working tree.

`git help`

- Help

`git help <command>`

References

- https://git-scm.com/docs/
- https://git-scm.com/book/en/v2
- https://www.vogella.com/tutorials/Git/article.html