Distributed Version Control – Git

Jakob Lykke Andersen
`jlandersen@imada.sdu.dk`

Tuesday 17th September, 2019

Contents

1 Introduction

Systems for version control can be briefly described as tools to keep track of changes in files. It thereby becomes easier both to find old versions of files and to be multiple people working on the same collection of files at the same time. Most version control systems are built either on a client-server architecture, or on a distributed architecture, like Git. Examples of other distributed version control systems (DVCS) are Mercurial and Bazaar. A widespread client-server system is Subversion.

Version control systems are useful in many different contexts. They can be used to keep track of large software products like the Linux kernel or the Android operating system, but they are also very useful even in small one-person projects like those faced in a programming course. Most open source projects are developed using some kind of version control system, with Git being a very common choice. However, the use of version control is not restricted to software development. It can be very beneficial to also use it for documentation and reports (for example LaTeX files).

The aim of this guide is not to explain every detail of Git, but only the basic concepts so it is easy to get started with basic version control. Git can be installed on both macOS, Windows, and Linux systems, but for the exercises we assume that a Ubuntu system similar to the one in the IMADA Computer Lab is used.

1.1 Notation

In the following section there will be snippets of text from a terminal, both with commands to be executed and their output. Lines on the form

```
some/path/to/a/folder$ command arg1 arg2 arg3
```

means that the command `command` with the arguments `arg1`, `arg2`, and `arg3` is executed in the folder `some/path/to/a/folder`. The remaining lines will be output from commands.

When a path starts with ~ it means that it is relative to the users home folder (`/home/`*`username`*). For example, the path `~/Downloads` for the user `john42` has the full path `/home/john42/Downloads`. Note: if you use execute `cd` without arguments, you will go to your home folder.

1.2 Getting Help

Git has a built-in help system: executing `git help` *`some-command`* will give you information about the command `git some-command` (press `q` to exit the help screen).

1.3 The Default Editor

Sometimes you need to write messages to describe your changes, which we here for illustrative purposes will write as a command line arguments. However, good descriptions are longer and spans multiple lines, so the recommended usage is to let Git spawn an editor. As default this editor is the terminal-based program Vim, which may be somewhat strange at first. While Vim is very powerful, we here just need the following basics:

- The editor can be in different *modes*; with "normal" and "insert" being the most common.
- Press `i` to go from normal to insert mode. Press `Esc` to go back to normal mode.
- When in normal mode, type :w to save, `:q` to quit, and `:q!` to quit and discard changes.

1.4 Installation (on non-IMADA machines)

Git can be installed in Ubuntu with

```
$ sudo apt install git
```

See the Git website, `https://git-scm.com/` for installation on other operating systems. In the following section we additionally assume the program `kdiff3` to be installed. The programs `gitk` and `gitg` may also be useful.

1.5 Setup

Whenever Git records changes it also wants to store who did it. To set this information up once and for all on a machine you can execute the following commands:

```
$ git config --global user.name "Your Name''
$ git config --global user.email "you@somewhere.com"
```

If you have not done this, Git will complain when it would otherwise have used the information.

Note, this configuration data is stored in "`~/.gitconfig`". This file can simply be copied to each new machine you are using instead of executing the commands above.

2 Mini Guide

When Git keeps track of file changes it is done by converting a folder into a *repository*. For example

```
gittest$ mkdir ex
gittest$ cd ex
gittest/ex$ git init
```

will create a folder `ex` and convert it into a repository. The exact same can be done in one step with

```
gittest$ git init ex
```

The initialization command creates a special folder `ex/.git`, which will contain the change history of files in the folder `ex/`, including those in subfolders.

Git needs to be told which files to keep track of, so it is not enough to just move files into the repository folder. Let us start with two files:

```
gittest/ex$ echo 'print("Hello world")' > hello.py
gittest/ex$ echo 'print(42)' > answer.py
gittest/ex$ ls -a
.  ..  answer.py  .git  hello.py
```

The first two lines, on the form `echo "data" > someFile`, deletes the content of `someFile` and writes `data` into it, while the command `ls -a` show the content of the current folder, including hidden files and folders.

We can now use the command `git status` to see what Git see as changes in the folder `ex`:

```
gittest/ex$ git status
On branch master

Initial commit

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

        answer.py
        hello.py

nothing added to commit but untracked files present (use "git add" to track)
```

The first lines indicates which version we are currently looking at. We will get back to that later. Then there is a section of *untracked* files, where our two files are listed. This means that when we change something in those files, then Git will not be able to tell us what we have done. The status command always gives hints at how to change the file status. In this case, we can use `git add` on our files to let Git know it should begin version control of our files:

```
gittest/ex$ git add answer.py hello.py
gittest/ex$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   answer.py
        new file:   hello.py
```

There are no longer any untracked files, but instead a list of changes that we have *staged*. In this case we are told that the changes are actually the addition of new files.

In Git the changes are grouped into *commit*s. We must therefore decide which sets of changes we find logically to be recorded as a single unit. For example, we can decide that actually we are not ready to commit `answer.py` yet, so we only want to stage `hellp.py` for the next commit. Luckily, the status command told us how to *unstage*:

```
gittest/ex$ git rm --cached answer.py
rm 'answer.py'
gittest/ex$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   hello.py

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

        answer.py
```

(Note: the terms "staging area", "the index", and "cached changes" all refers to the same concept)

We can now commit the staged changes using `git commit`:

```
gittest/ex$ git commit -m "Addition of a Python script"
[master (root-commit) 4fda6be] Addition of a Python script
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

Every commit must have a non-empty log message. If it is not given as an argument (with `-m "message"`) then some text editor will be opened automatically.

With `git status` we can see that only `answer.py` is listed now, as untracked. Let us quickly make a second commit with that file (try just typing `git add` and the press the tab key[1]):

```
gittest/ex$ git add answer.py
gittest/ex$ git commit -m "Another Python script"
[master 39e5e6e] Another Python script
 1 file changed, 1 insertion(+)
 create mode 100644 answer.py
```

We can see a list of all commits using `git log`:

```
gittest/ex$ git log --graph
* commit 39e5e6e24387dffe85997d001982169ab3928626
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:04:11 2017 +0200
|
|     Another Python script
|
* commit 4fda6beea0a7bede5d72bfe9717ab3504a3bd9b1
  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
  Date:   Sun Sep 10 15:03:18 2017 +0200

      Addition of a Python script
```

The optional `--graph` flag prints the left-most depiction, so we can see which parent(s) each commit has. Here the commit `39e5e6` has a single parent, `4fda6b`. From the log we can see that each commit has an assigned string of hexadecimal digits as the id of each commit. It is called the *hash* and is calculated by Git based on both which changes we made in that particular

[1]Actually, the tab key can be used all the time, even with half-written arguments, to find out what could go next.

commit, but also which commit those changes were based on, the author information, and the time stamps. This also means that when you are trying out the same commands your hashes most likely will be different.

Generally a commit may contain changes to many files at the same time, and it is therefore a snapshot of the complete repository.

Now change one of the files:

```
gittest/ext$ echo 'print("Hello universe")' > hello.py
```

Using the `status` command we see that the file has been marked as "modified", but that the changes are not yet staged:

```
gittest/ex$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   hello.py

no changes added to commit (use "git add" and/or "git commit -a")
```

We add the file and commit the changes:

```
gittest/ex$ git add -u
gittest/ex$ git commit -m "Update of hello.py"
```

Instead of specifying the file to add, we used `-u` which adds all modified and deleted files (but not new files).

To see an old version of the repository we can use the `checkout` command with the hash of some commit:

```
gittest/ex$ git checkout 39e5e6e
Note: checking out '39e5e6e'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 39e5e6e... Another Python script
```

Note that we only need specify enough of the prefix of a hash so there is no doubt which we mean (though at least 4 characters must be given). Looking at `hello.py` we can verify that it now does not have the latests changes:

```
gittest/ex$ cat hello.py
print("Hello world")
```

The three commands `add`, `commit`, and `checkout` thus transfers data between the files we can modify (called the working tree), the staging area (aka. the index), and the repository history. The specific transfer patterns are visualised in Figure 1.

In the last line of the checkout message Git tells us which commit we have now checked out. A special reference called the `HEAD` points to this commit. Giving the `--decorate` flag to `log` we can visualize both the commit history with this information:

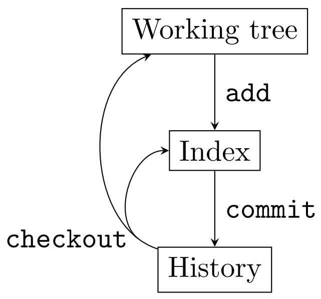

Figure 1: Depiction of how the commands `add`, `commit`, and `checkout` transfer data. You can see the difference between the working tree and the index with `git diff`, and the differences between the index and the current commit with `git diff --cached`. Specific files can be specified to limit the view of differences. (The `diff` command can do much more and has many useful options.)

```
gittest/ex$ git log --graph --decorate --all
* commit 1c80d8f41e8c7dd70f83737c7527401e1b87915c (master)
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:45:19 2017 +0200
|
|     Update of hello.py
|
* commit 39e5e6e24387dffe85997d001982169ab3928626 (HEAD)
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:04:11 2017 +0200
|
|     Another Python script
|
* commit 4fda6beea0a7bede5d72bfe9717ab3504a3bd9b1
  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
  Date:   Sun Sep 10 15:03:18 2017 +0200

      Addition of a Python script
```

We now have parentheses with some aliases for commits, and see as expected that `HEAD` is at the commit we have checked out into our working tree. By default the `log` command only lists commits starting from `HEAD`, but with `--all` it lists all commits.

In the checkout message Git also warns us that if we are in a so-called "detached HEAD" state, and that if we make new commits now they can potentially be lost if we do not create a new branch. A branch designates a logical line of work that we decide one. In software development it is common to have a main branch, and then branches for doing small experiments and new developments that are not ready for everyone else to be bothered with. Technically, a branch in Git is simply a reference to a commit with a nice name. The default branch is called `master` and from the log above we see that it indeed points to the newest commit in our repository. Note that a branch name can contain slash characters, which can help organize them, e.g., `feature/something`, `release/v42`, and `way/too/many/levels`.

To continue work on a branch we must first do a checkout on it:

```
gittest/ex$ git checkout master
Previous HEAD position was 39e5e6e... Another Python script
Switched to branch 'master'
gittest/ex$ git log --graph --decorate --all
* commit 1c80d8f41e8c7dd70f83737c7527401e1b87915c (HEAD -> master)
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:45:19 2017 +0200
|
|     Update of hello.py
|
* commit 39e5e6e24387dffe85997d001982169ab3928626
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:04:11 2017 +0200
|
|     Another Python script
|
* commit 4fda6beea0a7bede5d72bfe9717ab3504a3bd9b1
  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
  Date:   Sun Sep 10 15:03:18 2017 +0200

      Addition of a Python script
```

The `HEAD` no longer references a commit, but instead the `master` branch (which in turn references a commit).

Management of branches (e.g., listing, creating, deleting, removing) can be done using the **`git branch`** command. However, when creating branches we will often need to first do a checkout to get to the right starting point to start working. Therefore there is a shorthand:

```
gittest/ex$ git checkout -b test 39e5e6
Switched to a new branch 'test'
gittest/ex$ git log --graph --decorate --all
* commit 1c80d8f41e8c7dd70f83737c7527401e1b87915c (master)
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:45:19 2017 +0200
|
|     Update of hello.py
|
* commit 39e5e6e24387dffe85997d001982169ab3928626 (HEAD -> test)
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:04:11 2017 +0200
|
|     Another Python script
|
* commit 4fda6beea0a7bede5d72bfe9717ab3504a3bd9b1
  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
  Date:   Sun Sep 10 15:03:18 2017 +0200

      Addition of a Python script
```

Equivalently we could have done:

```
gittest/ex$ git checkout 39e5e6
gittest/ex$ git branch test
```

When we commit while on a branch, Git first creates the actual commit and then moves the branch to it:

```
gittest/ex$ echo "print('Hello IMADA')" > hello.py
gittest/ex$ git add -u
gittest/ex$ git commit -m "Local hello"
[test 9efed7b] Local hello
 1 file changed, 1 insertion(+), 1 deletion(-)
gittest/ex$ git log --graph --decorate --all
* commit 9efed7bf3e25bc7b1c5842f96aad002c73ac1764 (HEAD -> test)
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 19:00:33 2017 +0200
|
|     Local hello
|
| * commit 1c80d8f41e8c7dd70f83737c7527401e1b87915c (master)
|/  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
|   Date:   Sun Sep 10 15:45:19 2017 +0200
|
|       Update of hello.py
|
* commit 39e5e6e24387dffe85997d001982169ab3928626
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:04:11 2017 +0200
|
|     Another Python script
|
* commit 4fda6beea0a7bede5d72bfe9717ab3504a3bd9b1
  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
  Date:   Sun Sep 10 15:03:18 2017 +0200

      Addition of a Python script
```

From the log we can now see that the two branches `master` and `test` have diverged. They each have changes the other branch does not.

(Note that the `diff` command can also be used to view the differences to other than the `HEAD` commit, and even between to commits/branches. Try for example `git diff --word-diff master test`)

Sometimes it is useful to work on a specific thing in another branch, say our `test` branch, and then merge it into the main branch later. First, we add a few more changes:

```
gittest/ex$ echo "print('Hello again')" >> hello.py
gittest/ex$ rm answer.py
gittest/ex$ echo "This is a test repository" > README
gittest/ex$ git add -u
gittest/ex$ git add README
gittest/ex$ git commit -m "A lot of stuff"
[test e69cb73] A lot of stuff
 3 files changed, 2 insertions(+), 1 deletion(-)
 create mode 100644 README
 delete mode 100644 answer.py
```

Merging two branches is an asymmetric operation; one branch gets the changes of the other, while the other is not modified.[2] In this case we want our changes to go back into the `master` branch, so we first switch to that branch and then do the merge:

```
gittest/ex$ git checkout master
gittest/ex$ git merge test
Auto-merging hello.py
CONFLICT (content): Merge conflict in hello.py
Removing answer.py
Automatic merge failed; fix conflicts and then commit the result.
```

Most often Git can merge changes but in this case it was not sure how to proceed for all files. It could for example mean that there are different changes at positions too close to each other, or it is ambiguous in which order to put two additions of data. In general it can be difficult

[2]You can actually merge many branches into one branch at the same time.

to resolve these conflicts, because we semantically need to judge what the right thing to do is. For example, imagine you are writing a report with another person. One of you rewrites a sentence while the other are fixing a typo in the old sentence. There is no way for Git to know what the result is supposed to be so the one person doing the merge must make a choice. However, if changes are happing in different lines of a file, then generally Git can merge the changes automatically.

Back to our example: to see exactly which files are in conflict we use the status command:

```
gittest/ex$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:

        new file:   README
        deleted:    answer.py

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   hello.py
```

We see that some changes are ready to commit, but there is another section of "unmerged paths", i.e., files in conflict. Note that the index and working tree is in a special merge state, so if we change our minds we should use the suggested command to completely abort the merge.

It is just `hello.py` and we must now manually edit the file to look like we want, and then add and commit as usual. Git has inserted special merge markers in `hellp.py` to help us decide what to do:

```
gittest/ex$ cat hello.py
<<<<<<< HEAD
print("Hello universe")
=======
print('Hello IMADA')
print('Hello again')
>>>>>>> test
```

In this case the whole content is related to the markers, but in general if there are multiple positions in the file with conflicts, then there will be a marker set for each of them. A merge is as mentioned asymmetric: we merge a branch into the current branch. The markers reflect that: the first section is the content as it appears on the current branch (also called the "local" branch in this context) and the second section is how it appears in the other branch (also called the "remote" branch in this context). Actually there is a third piece of useful data which may be useful, but is not displayed: the merge base. This is how the file looked like at the most recent point in the history just before the branches diverged. When using tools to assist in resolving conflicts, such as `kdiff3`, they will usually display the merge base as well, just for reference to help us.

In this case we will manually fix the problem. Assume we edit the conflicted file as shown below, we can now complete the merge:

```
gittest/ex$ cat hello.py
print("Hello universe")
print("Hello again")
gittest/ex$ git add hello.py
gittest/ex$ git commit # an editor opens with a prepared message about the merge
[master 6691272] Merge branch 'test'
gittest/ex$ git log --graph --decorate --all
*   commit 66912724603858 2b800b40d72b9dac2989cac6e8 (HEAD -> master)
|\  Merge: 1c80d8f e69cb73
| | Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| | Date:   Sun Sep 10 22:17:13 2017 +0200
| |
| |     Merge branch 'test'
| |
| * commit e69cb730c31a5b08b67853441e4e960a723d3ee7 (test)
| | Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| | Date:   Sun Sep 10 20:08:53 2017 +0200
| |
| |     A lot of stuff
| |
| * commit 9efed7bf3e25bc7b1c5842f96aad002c73ac1764
| | Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| | Date:   Sun Sep 10 19:00:33 2017 +0200
| |
| |     Local hello
| |
* | commit 1c80d8f41e8c7dd70f83737c7527401e1b87915c
|/  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
|   Date:   Sun Sep 10 15:45:19 2017 +0200
|
|       Update of hello.py
|
* commit 39e5e6e24387dffe85997d001982169ab3928626
| Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| Date:   Sun Sep 10 15:04:11 2017 +0200
|
|     Another Python script
|
* commit 4fda6beea0a7bede5d72bfe9717ab3504a3bd9b1
  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
  Date:   Sun Sep 10 15:03:18 2017 +0200

      Addition of a Python script
```

2.1 Working With Multiple Clones

While using version control locally to organize changes is already useful, the usual workflow is to have copies of the same repository on multiple machines, with multiple people contributing changes.

For this part we assume that a copy of the repository above is our central copy, and now Alice and Bob wants to contribute. We emulate this by having distinct folders for them:

```
gittest$ mkdir Server && cd Server && git clone --bare ../ex Test.git && cd ..
gittest$ mkdir Alice
gittest$ mkdir Bob
```

The actual cloning by Alice is the following, with Bob being the same in his folder:

```
gittest/Alice$ git clone ../Server/Test
```

Note that when cloning we do not get the index or the working tree from the original.

From the view of a single repository another clone is called a *remote*, and the set of those can be managed with the `remote` command. When a repository is created with `clone` then the original is automatically added as a remote called `origin`. When Alice looks at all her branches[3]

```
gittest/Alice/Test$ git branch --all -vv
* master                 6691272 [origin/master] Merge branch 'test'
  remotes/origin/HEAD    -> origin/master
  remotes/origin/master 6691272 Merge branch 'test'
  remotes/origin/test    e69cb73 A lot of stuff
```

she sees a branch `master`, which is her own, and some branches inside `remote/`⟨remote name⟩`/`. The remote branches are simply a copy of all the branches from each of the remotes, but they can not be modified. When there is no naming conflict we can refer to remote branches without the prefix, that is as ⟨remote name⟩`/`⟨branch name⟩. The verbose output (the `-vv` flag) tells us that the `master` branch is *tracking* a remote branch, namely `origin/master`. This tracking information can be used in many commands automatically, instead of giving all the information as arguments.

Alice realizes that there were some changes missing in the `test` branch, so she must first create a local branch:

```
gittest/Alice/Test$ git checkout -b test origin/test
Branch test set up to track remote branch test from origin.
Switched to a new branch 'test'
```

Note that we can call the local branch anything we want, but often it is easier to use the same name as on the remote. Therefore Alice could also have done `git checkout --track origin/test`. Actually, as she only has one remote she could have even done `git checkout test`.

She decides that the Python script should actually be an executable program, and changes `hello.py` accordingly:

```
gittest/Alice/Test$ cat hello.py
#!/usr/bin/env python
print("Hello IMADA")
print("Hello again")
gittest/Alice/Test$ chmod +x hello.py
gittest/Alice/Test$ mv hello.py hello
```

Using the `status` command she sees that Git claims she deleted a file and has created a new untracked file. However, after `git add hello.py hello` Git has deduced, based on he similarity of the file contents, that it was most likely a renaming of the file along with a few changes. Alice commits and uses the `push` command to update the remote with the new commit and the modification of the `test` branch:

```
gittest/Alice/Test$ git commit -m "Make hello.py into a program"
[test 6956545] Make hello.py into a program
 1 file changed, 1 insertion(+)
 rename hello.py => hello (64%)
 mode change 100644 => 100755
 gittest/Alice/Test$ # try doing a git status here
gittest/Alice/Test$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 345 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /gittest/Alice/../Server/Test
   e69cb73..6956545  test -> test
```

A push is usually done on a single branch at a time, here `test`. Alice used a short form of the command, as the branch to be pushed was deduced from the current branch (i.e., what `HEAD`

[3]Try also to look at the decorated log for visualizing the relationship between remote and local branches.

points to). The longer, longer form is `git push origin test`, but that is a again a shorthand as the tracking information is used to decide which remote branch to push to. The even longer form is `git push origin test:test`, where the first `test` is the local branch name and the second, after the colon is the branch name on the remote to update.

Meanwhile, Bob has been busy modifying both the `master` and `test` branch:

```
gittest/Bob$ git clone ../Server/Test; cd Test
gittest/Bob/Test$ echo "Write some documentation at some point" > TODO
gittest/Bob/Test$ git add TODO
gittest/Bob/Test$ git commit -m "Add TODO list"
[master 197c9e5] Add TODO list
 1 file changed, 1 insertion(+)
 create mode 100644 TODO
gittest/Bob/Test$ git checkout test
Branch test set up to track remote branch test from origin.
Switched to a new branch 'test'
gittest/Bob/Test$ echo 'print("Hello again again")' >> hello.py
gittest/Bob/Test$ git add hello.py
gittest/Bob/Test$ git commit -m "More hello"
[test 81a14b5] More hello
 1 file changed, 1 insertion(+)
```

He is now done and wants to push both branches:

```
gittest/Bob/Test$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 345 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /gittest/Bob/../Server/Test
   6691272..197c9e5  master -> master
gittest/Bob/Test$ git push
To /gittest/Bob/../Server/Test
 ! [rejected]        test -> test (fetch first)
error: failed to push some refs to '/gittest/Bob/../Server/Test'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

However, Alice has updated the `test` branch on the server so there is no safe way to push the unrelated changes from Bob. He therefore uses the `fetch` command to transfer commits and branch updates from the remote:

```
gittest/Bob/Test$ git fetch # or in general: git fetch remote-name
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /gittest/Bob/../Server/Test
   e69cb73..6956545  test       -> origin/test
gittest/Bob/Test$ git log --graph --decorate --all
* commit 81a14b59b7e9a84a54351663886563ce89f70e59 (HEAD -> test)
| Author: Bob <bob@imada.sdu.dk>
| Date:   Mon Sep 11 12:07:13 2017 +0200
|
|     More hello
|
| * commit 197c9e5777ac3bc9f709aef1b09d34ba8a21cb81 (origin/master, origin/HEAD, master)
| | Author: Bob <bob@imada.sdu.dk>
| | Date:   Mon Sep 11 12:04:27 2017 +0200
| |
| |     Add TODO list
| |
| *   commit 669127246038582b800b40d72b9dac2989cac6e8
| |\  Merge: 1c80d8f e69cb73
| |/  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
|/|   Date:   Sun Sep 10 22:17:13 2017 +0200
| |
| |       Merge branch 'test'
| |
| * commit 1c80d8f41e8c7dd70f83737c7527401e1b87915c
| | Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| | Date:   Sun Sep 10 15:45:19 2017 +0200
| |
| |     Update of hello.py
| |
| | * commit 6956545aa67c3d8461feedf8176f931edf847528 (origin/test)
| |/  Author: Alice <alice@imada.sdu.dk>
|/|   Date:   Mon Sep 11 11:49:52 2017 +0200
| |
| |       Make hello.py into a program
| |
* | commit e69cb730c31a5b08b67853441e4e960a723d3ee7
| | Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| | Date:   Sun Sep 10 20:08:53 2017 +0200
| |
| |     A lot of stuff
# ...
```

Bob sees the commit from Alice, which is where the remote `test` branch points, and uses a shorthand of the `merge` command, which uses the tracking information, to combine the changes:

```
gittest/Bob/Test$ git merge # an editor opens with a prepared message
Auto-merging hello
Merge made by the 'recursive' strategy.
 hello.py => hello | 1 +
 1 file changed, 1 insertion(+)
 rename hello.py => hello (75%)
 mode change 100644 => 100755
 gittest/Bob/Test$ git log --graph --decorate --all
 *   commit aae221ea0021b777080a5328b2990ad626a29929 (HEAD -> test)
|\  Merge: 81a14b5 6956545
| | Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| | Date:   Mon Sep 11 12:19:35 2017 +0200
| |
| |     Merge remote-tracking branch 'refs/remotes/origin/test' into test
| |
| * commit 6956545aa67c3d8461feedf8176f931edf847528 (origin/test)
| | Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
| | Date:   Mon Sep 11 11:49:52 2017 +0200
| |
| |     Make hello.py into a program
| |
* | commit 81a14b59b7e9a84a54351663886563ce89f70e59
|/  Author: Jakob Lykke Andersen <jlandersen@imada.sdu.dk>
|   Date:   Mon Sep 11 12:07:13 2017 +0200
|
|       More hello
# ...
gittest/Bob/Test$ git push
 Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 672 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To /gittest/Bob/../Server/Test
   6956545..aae221e  test -> test
```

Bob could also have used the `pull` command as a shorthand for the common task of first running `fetch` and then `merge` with the remote branch being tracked.

Alice can now get Bobs changes:

```
gittest/Alice/Test$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
gittest/Alice/Test$ git pull
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 9 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
From /gittest/Alice/../Server/Test
   6691272..197c9e5  master     -> origin/master
   6956545..aae221e  test       -> origin/test
Updating 6691272..197c9e5
Fast-forward
 TODO | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 TODO
```

In the output we first see the information from fetching and then from the merge. Note that in this case Alice did not have new changes to the `master` branch, so it can just be moved forward to the appropriate child commit. This is called a *fast-forward merge*.

2.2 Merge Conflicts

We already saw what happens when it is not clear how to merge two branches; markers are inserted in the conflicted files and we must manually edit the file to make it look like we deem correct. Merge algorithms are usually line-based, so if two persons have edited the same line in a file it almost always results in a conflict. There are several programs that can assist the merging procedure, for example `meld` or `kdiff3`. They are so-called 3-way merge programs, because they show both the two commits to merge and the latest ancestor commit they have in common. Running `git mergetool` will start whatever program is available, and if you have not yet installed your favourite 3-way merge program, then it may be the terminal-based `vimdiff` which is started. If no files are specified to `git mergetool` it will in turn start the merge program for each conflicted file.

3 Other use(ful/less) information

3.1 Hosting a Repository and Group Work

Even though one can work with remotes completely distributed (i.e., Alice and Bob adding each other as remotes, without the server), it is most often still easier to have a central clone, hosted on some server. Many open-source projects for example use GitHub (`https://github.com`), which is free for public repositories. At IMADA we have a server at `https://git.imada.sdu.dk` which may be used for repositories related to your studies. If you have access to the IMADA Computer Lab, you should be able to log into this server as well. In some courses you may be provided with a special repository on this server. For group work you can give access to a repository to other group members by adding them as collaborators in the repository settings. Note that they must have logged into the server before they can be added.

3.2 Repository Contents

Not all kinds of data easy to merge in the way we saw above. LaTeX-documents, Python and Java code, and other files consisting of raw text are not a problem, while files like Word documents (which are zip-files), PDF files and compiled programs are what is called *binary files* (try opening them in `gedit`). The problem is that in order to merge binary files, you must have intricate knowledge of the file format, which ordinary merge programs do not have. In general it is therefore a bad idea to put binary files in a repository, if those files can be generated from text files. This is for example the case with PDF documents generated from LaTeX code, or compiled Java programs (i.e., the `.class` files).

It can become quite annoying that when using the `status` command all these generated files show up, even though we will never commit them. Therefore Git looks for a special file in the root of the repository called `.gitignore` (note the leading dot). In this file we can put patterns for file names that Git should pretend does not exist. For example, assume we are making a Java project with our source code in a folder called `src` and a report, written in LaTeX, in a folder `report`. We could then use the following `.gitignore` file to make our lives a bit less cluttered:

```
myJavaProject> cat .gitignore
# Java
*.class

# Latex (there can be more file types for more complicated documents)
*.aux
*.log
*.out
*.toc

# Not all PDFs should be ignored, but our report should
report/report.pdf
```

Note that you can/should add and commit the ignore file to your repository.

Use `git help gitignore` to learn more about the format of the ignore file.

4 Exercises

See Section 1 about setting up your machine.

1. Perform the steps in the mini guide, while using commands to query the state of the repository to be sure what is going on. Use for example the commands `status`, `log`, `branch`, and `remote` to list information. Try also the programs `gitk` and `gitg` to view the history.

2. What do you do if there are changes you want to discard? (hint: what does `status` tell?)

3. What do you do if you are in the middle of something and someone (maybe your self) tells you to just quickly make this other small change? There is a way to do this by committing

on another branch and clean it up later, but there is another way. Look into the `stash` command and try it out.

4. Instead of manually merging conflicted files, try one of the merge programs.

5. The question of "what has changed?" come in many shapes. Explore the use of the `diff` command, for example the `--cached` option. Try also when files have been deleted and when new files has been added. The `log` command can also be useful, try for example the `--stat` and `--patch` flags. To see the diff for a single commit you can also use `git diff` ⟨hash⟩`^!`.

6. A branch is a symbolic name for a developing set of changes, but sometimes it is useful to add a symbolic name that always refers to the same commit. For example, for software it is common to tag commits corresponding to finished versions. Look into the `tag` command for this purpose.

Additional exercises:

- Log into the IMADA Git server, `https://git.imada.sdu.dk`.
- For each of your courses create a repository to store notes, code, and other material. Note that when you at some point work on larger programming projects and use an IDE, then the special IDE files should not be committed manually, but should be handled through the IDE.
- When you have group projects, set up additional private repository for just the shared part. Each group member can have their own clone and use the *pull request* feature of the server software.
- Look into the commands `reset` and `rebase`.

5 References

Official website for Git: `https://git-scm.com/`

Visual Git Reference: `https://marklodato.github.io/visual-git-guide/index-en.html`