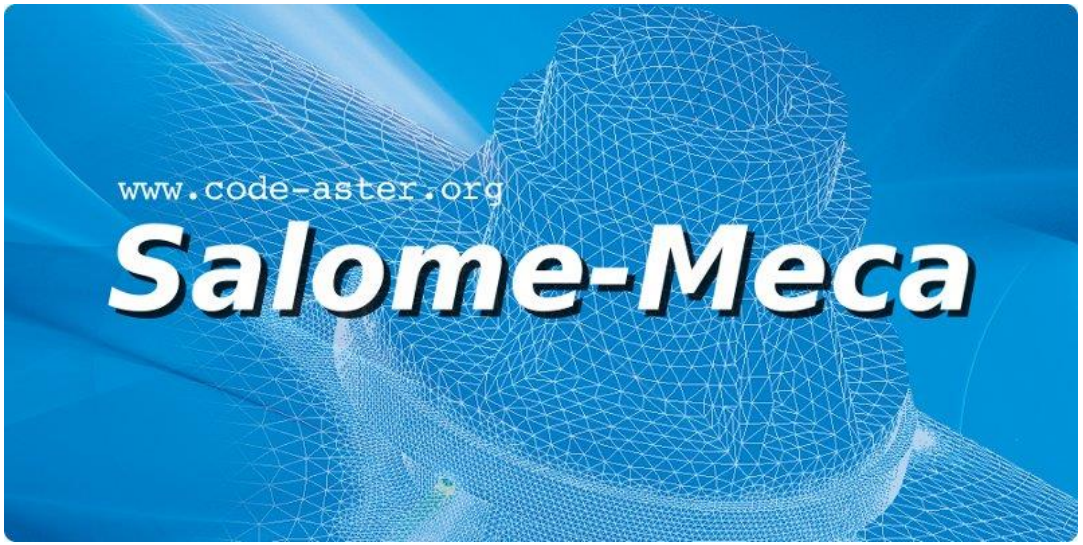
Development in code_aster
Using Mercurial

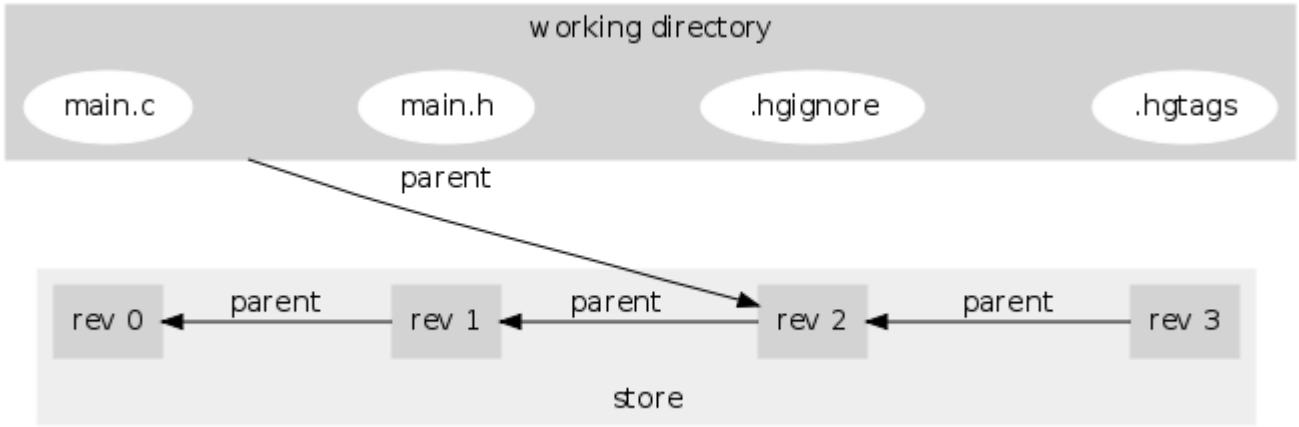


Code_Aster, Salome-Meca course material

EDF

Version Control System

- Version control is the management of changes to documents, computer programs, web sites, and other collections of information.
- Mercurial repositories contain a **working directory** coupled with a **store**:
  - The store contains the **complete** history of the project.
  - The working directory contains a copy of the project's files at a given point in time (e.g. rev 2).

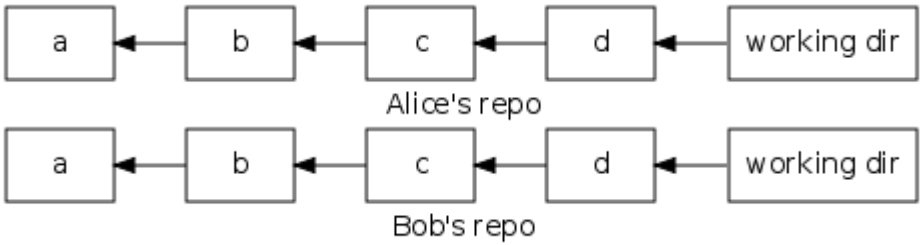


Distributed Version Control System (1)

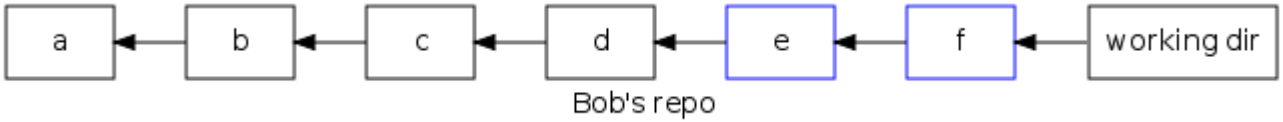
- Mercurial is a **distributed version control system** (DVCS) also known as **decentralized version control system**.
- All contributors have a clone of the repository.
- Allow to work on a given project without requiring them to share a common network.
- Common operations (such as **commits**, viewing **history**, and **reverting** changes) are fast, because there is no need to communicate with a central server.

Distributed Version Control System (2)

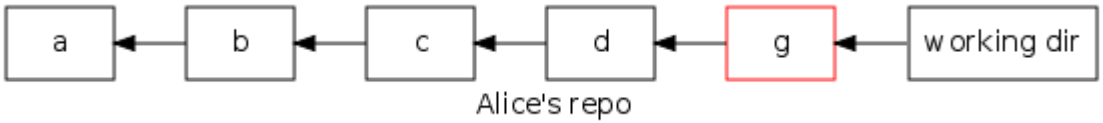
Alice & Bob clone the same repository:



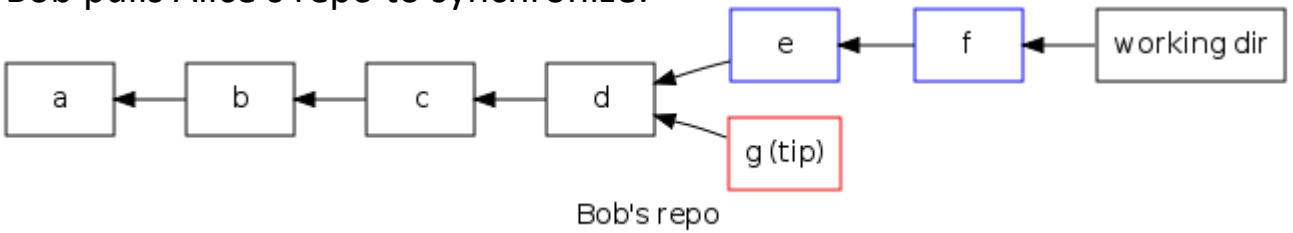
Bob work independently of Alice. He then commits two changes e and f:



Alice then makes her own change g in parallel:



Bob pulls Alice's repo to synchronize:



Mercurial guide - Initialisation

As first step, you should teach Mercurial your name. For that you open the file ~/.hgrc in a text-editor:

```
[ui]
username = Mr. Johnson <johnson@smith.com>
editor = nano        # or: gedit --new-window
```

Initialize the project.

Now you add a new folder in which you want to work:

```
$ hg init project
```

Clone an existing project:

```
$ hg clone http://aster-repo.der.edf.fr/scm/hg/formation project
```

Mercurial guide – Adding files and tracking them

Enter the project folder, create some files, then add and commit them.

```
$ cd project
$ echo 'print("Hello")' > hello.py
$ hg add
adding hello.py

$ hg commit
(your default editor opens,
 enter the commit message,
 save and close.)
```

You can also enter the commit message on the command-line:

```
$ hg commit –m "Initial commit."
```

You can then look into your initial history with hg log:

```
$ hg log
changeset:   0:a5ecbf5799c8
user:        Mr. Johnson
date:        Sun Nov 20 11:00:00 2011 +0100
summary:     Initial commit.
```

Mercurial guide – Save changes

First do some changes:

```
$ echo 'print("Hello World")' > hello.py
```

See which files changed, which have been added or removed, and which aren't tracked yet:

```
$ hg status
M hello.py
```

See the exact changes:

```
$ hg diff
diff --git a/hello.py b/hello.py
--- a/hello.py
+++ b/hello.py
@@ -1,1 +1,1 @@
-print("Hello")
+print("Hello World")
```

Commit the changes:

```
$ hg commit –m "Say Hello World, not just Hello."
```

Mercurial guide – History

Your history now looks like this:

```
$ hg log
changeset:   1:487d7a20ccbc
user:        Mr. Johnson
date:        Sun Nov 20 11:11:00 2011 +0100
summary:     Say Hello World, not just Hello.

changeset:   0:a5ecbf5799c8
user:        Mr. Johnson
date:        Sun Nov 20 11:00:00 2011 +0100
summary:     Initial commit.
```

To see a certain revision, you can use the -r switch (--revision). To also see the diff of the displayed revisions, there's the -p switch (--patch)

```
$ hg log -p -r 487d7
```

All Mercurial commands and options are documented, see:

```
$ hg help <command name>
```

Mercurial guide – Copy/rename

When you copy or move files, you should tell Mercurial to do the copy or move for you, so it can track the relationship between the files.

```
$ hg cp hello.py copy
$ hg mv hello.py target
```

Now you have two files, "copy" and "target", and Mercurial knows how they are related.

```
$ hg diff --git
diff --git a/hello.py b/copy
rename from hello.py
rename to copy
diff --git a/hello.py b/target
copy from hello.py
copy to target
$ hg commit –m "Copy and move."
```

Note:

Should you forget to do the explicit copy or move, you can still tell Mercurial to detect the changes via *hg addremove --similarity 100*. See also the option *--after* of *hg mv*.

Mercurial guide – Nonlinear history

Check your history. This prints a list of changesets graphically with their id and their commit message:

```
$ hg log -G --template="{rev}:{node|short} {branch}: {desc|firstline}\n"

@  2:70eb0ca9d264 default: Copy and move.
|
o  1:487d7a20ccbc default: Say Hello World, not just Hello.
|
o  0:a5ecbf5799c8 default: Initial commit.
```

Let's assume the bug was introduced in revision 1.

```
$ hg update 1
$ echo 'print("Hello Mercurial")' > hello.py
$ hg commit –m 'Greet Mercurial'
created new head
```

Heads are current states of your project living side by side.

It is a good practice to merge them together before propagating them.

Mercurial guide - Merging

Now the fix is already stored in history. We just need to merge it with the current version of your code.

```
$ hg merge
merging hello.py and copy to copy
merging hello.py and target to target
```

If there are conflicts use *hg resolve* - that's also what merge tells you to do in case of conflicts:

- First list the files with conflicts

  ```
  $ hg resolve --list
  ```

- Then resolve them one by one. resolve attempts the merge again

  ```
  $ hg resolve conflicting_file
  (fix it by hand, if necessary)
  ```

- Mark the fixed file as resolved

  ```
  $ hg resolve --mark conflicting_file
  ```

- Commit the merge, as soon as you resolved all conflicts. This step is also necessary when there were no conflicts!

  ```
  $ hg commit -m 'merge greeting and copy+move.'
  ```

Mercurial guide – Sharing changes

Mercurial allows you to share your changes with others very easily by including a simple webserver.

Using the builtin webserver:

```
$ hg serve
listening at http://IPADDR:8000/ (bound to *:8000)
```

Now all others can point their browsers to this IP address at port 8000. They will then see all his history there and can decide if they want to pull his changes.

```
$ firefox http://IPADDR:8000
```

If they decide to include the changes, they just pull from the same URL

```
$ hg pull http://IPADDR:8000 --rev REV_TO_PULL
```

Prefer the use of a shared repository with write access to push changesets to others.

Mercurial guide – Sharing changes

Using patches:

On the sender side, export the changeset 4:

```
$ hg export 4 > change4.diff
```

Now attach the patch to an email and your colleagues can just run import on the diff to get your full changes, including your user information.

On the recipient side:

```
$ hg import change4.diff
```

Using bundles:

More robust, bundles are snippets of your history:

```
$ hg bundle --base FIRST_REV_TO_BUNDLE changes.bundle
(all revs after FIRST_REV)
$ hg bundle --base FIRST_REV_TO_BUNDLE --rev LAST_REV changes.bundle
(all revs after FIRST_REV and up to LAST_REV)
```

Suppose FIRST_REV is available on both sides.

```
$ hg bundle –rev LAST_REV changes.bundle shared_repository
(all revs up to LAST_REV that are not in shared_repository)
```

Send the bundle by email and extract it by executing:

```
$ hg unbundle changes.bundle
```

Mercurial guide – Removing history

You tried some code changes you prefer to forget:

```
@  5:bcc4ddc197c6 A very bad idea.
|
o    4:3d5a5444a374 merge greeting and copy+move.
|\
| o  3:0d958b2f3202 Greet Mercurial
| |
...
```

Just clone your own repository up to rev 4:

```
$ hg clone project newproject --rev 4
```

Enable mq extension and strip the bad changeset:

Add in ~/.hgrc:

```
[extensions]
mq =
```

And:

```
$ hg strip 5
saved backup bundle to .hg/strip-backup/bcc4ddc197c6-backup.hg
```

Mercurial guide – Creating a linear history (1)

Fix another bug in revision 1:

```
$ hg strip 4       # "cancel" the merge
$ hg update 1
$ echo 'new feature' > main.c
$ hg add main.c
$ hg commit -m 'Add a new feature.'
```

Current history:

```
@  4:dadb923b22a4 default: Add a new feature.
|
| o  3:0d958b2f3202 default: Greet Mercurial
|/
| o  2:0e2e31bde483 default: Copy and move.
|/
o  1:f5eeae4563de default: Say Hello World, not just Hello.
```

The recommended practice is to merge each fix into the main branch.

Mercurial guide – Creating a linear history (2)

Merging each branches will give:

```
@    6:63e267d7a21d default: merge fix B
|\
| o    5:0508329928d5 default: merge fix A
| |\
| | o  4:dadb923b22a4 default: Add a new feature.
| | |
o---+  3:0d958b2f3202 default: Greet Mercurial
 / /
o /  2:0e2e31bde483 default: Copy and move.
|/
o  1:f5eeae4563de default: Say Hello World, not just Hello.
```

We use another workflow for code_aster repositories.

As there are sometimes 30 or 40 features or bugfixes per week, the main repository does not show all these 'feature branches' merged, but a linear history for readability.

https://bitbucket.org/code_aster/codeaster-src/wiki

Mercurial guide – Creating a linear history (3)

hg rebase :

Rebase uses repeated merging to graft changesets from one part of history (the source) onto another (the destination). This can be useful for linearizing *local* changes relative to a master development tree (*from hg rebase --help*).

```
@  4:dadb923b22a4 default: Add a new feature.
|
| o  3:0d958b2f3202 default: Greet Mercurial
|/
| o  2:0e2e31bde483 default: Copy and move.
|/
o  1:f5eeae4563de default: Say Hello World, not just Hello.
```

We would obtain a linear history:

```
o  4:6def3a75ab33 default: Greet Mercurial
|
o  3:8a3b38e2e34c default: Copy and move.
|
@  2:dadb923b22a4 default: Add a new feature.
|
o  1:f5eeae4563de default: Say Hello World, not just Hello.
```

Mercurial guide – Creating a linear history (4)

Build a linear history:

```
$ hg rebase -s 0e2e31 -d tip
$ hg rebase -s 0d958b -d tip
```

Prefer use the unique revision ID since numbers can be different for other people… and change when you rewrite the history!

```
o  4:6def3a75ab33 default: Greet Mercurial
|
o  3:8a3b38e2e34c default: Copy and move.
|
@  2:dadb923b22a4 default: Add a new feature.
|
o  1:f5eeae4563de default: Say Hello World, not just Hello.
```

Note that hg rebase doesn't necessarily update the working directory.

Cancel rebase with: hg rebase --abort

Mercurial guide - Phases

Phases:

```
$ hg log -G --template="{rev}:{node|short} {branch}: {desc|firstline}
  ({phase})\n"
@  4:6def3a75ab33 default: Greet Mercurial (draft)
|
o  3:8a3b38e2e34c default: Copy and move. (draft)
|
o  2:dadb923b22a4 default: Add a new feature. (draft)
|
o  1:f5eeae4563de default: Say Hello World, not just Hello. (draft)
```

Changeset phase is **draft** by default, **public** when the changeset is exchanged (pull/push).

Can be **secret** (see *hg help phase* and *hg help phases*).

You should not rebase changesets that have already been shared with others. This will end up with duplicated changesets.

Mercurial guide – Amend last commit

Let’s print a message in our code:

```
$ echo 'print("end of file")' >> target
$ hg commit –m 'print a message'
```

History:

```
@  5:4f63ff739a05 default: print a message
|
o  4:6def3a75ab33 default: Greet Mercurial
```

It could be better:

```
$ sed -i 's/file/execution/g' target    # replace file by execution
$ hg commit --amend -m 'print a message at the end of execution'
saved backup bundle to .hg/strip-backup/4f63ff739a05-amend-backup.hg
```

New history:

```
@  5:fc8ba70c52a4 default: print a message at the end of execution
|
o  4:6def3a75ab33 default: Greet Mercurial
```

Mercurial guide – Rewriting history

Sometimes changesets are meaningless:

```
$ echo 'print("exit")' >> target
$ hg commit -m 'print exit'
```

History:

```
@  6:42a43a2f9e73 default: print exit
|
o  5:fc8ba70c52a4 default: print a message at the end of execution
|
o  4:6def3a75ab33 default: Greet Mercurial
```

We would like to fuse the last 2 commits:

```
$ hg histedit fc8ba70
```

Your editor opens, file instructions:

```
pick fc8ba70c52a4 5 print a message at the end of execution
fold 42a43a2f9e73 6 print exit
```

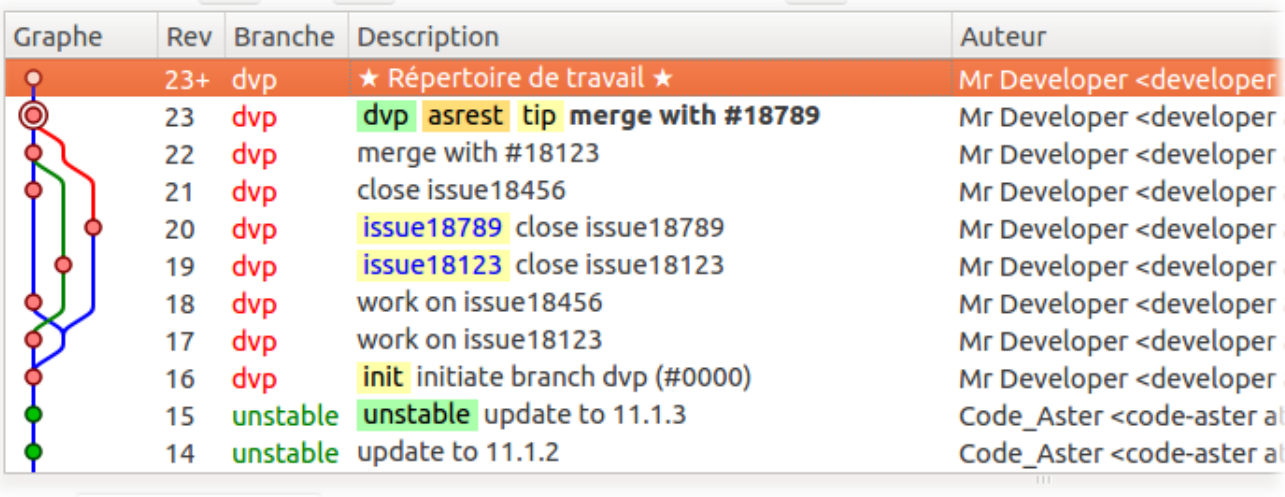
Save and quit the editor.

Edit the commit message for the fused changeset, remove ***…

Cancel history editing with: hg histedit --abort

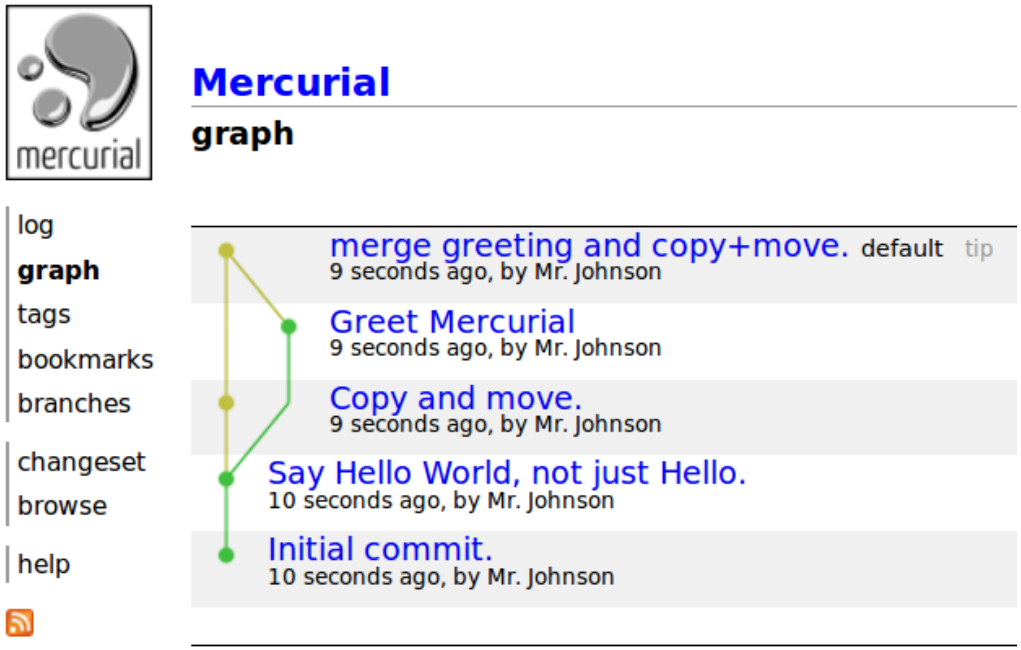
The same with a GUI : TortoiseHg

- TortoiseHg is a Mercurial revision control client
- Most Mercurial operations can be invoked from the graphical interface.
- On Calibre9, just launch « `thg` » (if not available install it by running « `pkcon install tortoise-hg` »)



View history in your browser

Simply use *hg serve* and open http://localhost:8000 in your browser



How to write a commit message?

- In English.
- The first line should be enough to understand what was done. In less than 80 characters and starting with the number of the relevant issue in the format [#12345] .
- The following lines allows to detail how it was done.
- Empty messages are forbidden (at least two words).
- Describe what was done, not what the problem was.
  - ☹ *sdll136a fails in version 3.2.1*
  - ☺ *fix uninitialized variable in cpmute*
- No need to list modified files.
- http://aster-rex.der.edf.fr/mercurial/faq.html#comment-ecrire-le-message-de-commit

Resources

- A few references :
  - Site of the project: https://www.mercurial-scm.org
  - Tutorial: https://www.mercurial-scm.org/guide
  - Book : http://hgbook.red-bean.col
  - Code_Aster repository documentation : http://aster-services.der.edf.fr/mercurial/annexes/memo_dummy.html
  - Mercurial help : `hg help, hg rebase -h…`

End of presentation

Is something missing or unclear in this document?

Or feeling happy to have read such a clear tutorial?

Please, we welcome any feedbacks about Code_Aster training materials.

Do not hesitate to share with us your comments on the Code_Aster forum dedicated thread.