Git

Principes et utilisation pour les projets tuteurés

Philippe Dosch

Philippe.Dosch@loria.fr

15 novembre 2012

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Problématiques générales

- Comment gérer l'historique des fichiers sources d'un projet ?
  - archivage
  - comparaison de la version courante par rapport à une ancienne
  - récupération d'une ancienne version
- Comment gérer les différentes versions d'un projet ?
  - version 1, version 2... : une version est un ensemble de fichiers dans un état donné
  - développements parallèles : version stable, de correction de bugs, d'ajout de fonctionnalités...

Problématiques spécifiques au travail en groupe

- Comment partager du code source ?
- Comment travailler à plusieurs sur des sources ?
- Comment travailler au même moment sur des sources ?
- Comment réconcilier les changements de contributeurs ?
- Comment ne pas perdre de travail ?

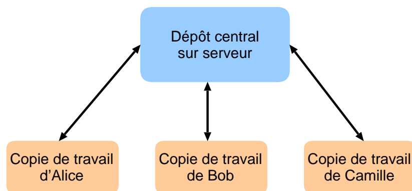
UNIVERSITÉ DE LORRAINE | IUT nancy Charlemagne

La solution : les systèmes de gestion de version (VCS)

Ensemble de méthodes et d'outils qui maintiennent les différentes versions d'un projet à travers tous les fichiers qui le composent

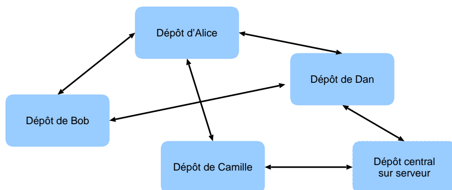
- permet le développement collaboratif et simultané
- permet de garder tout l' historique de tous les fichiers
- permet le développement parallèle : version stable, de développement, introduction de fonctionnalités, correction de bugs...
- permet de savoir pourquoi, quand et par qui une portion spécifique de code a été introduite

Systèmes de gestion de version centralisés (CVCS)



- Le serveur détient tout l'historique du projet
- Les utilisateurs possèdent seulement une copie des fichiers correspondant au code
- Toutes les opérations sont réalisées par l'intermédiaire du serveur (*i.e. online*)
- Les échanges de code sont obligatoirement effectués grâce au serveur

Systèmes de gestion de version décentralisés (DVCS)



- Chaque utilisateur possède un *dépôt* local complet, contenant tout l'historique du projet
- Les opérations sont réalisées localement (*i.e. offline*)
- Des serveurs *peuvent* assurer les échanges de code
- Du code *peut* aussi être échangé avec d'autres utilisateurs sans serveur centralisé (SSH ou mail typiquement)

Historique des systèmes de gestion de version

- Systèmes de gestion de version *centralisés*
  - CVS (*Concurrent Versions System*), 1990
  - SVN (*Subversion*), 2004
- Systèmes de gestion de version *décentralisés*
  - BitKeeper, 1998
  - Mercurial, 2005
  - Git, 2005
  - et d'autres : GNU Arch, Bazaar, Monotone, Darcs...

Git

- Créé en 2005 par Linus Torvalds pour la gestion des sources de Linux, en remplacement de BitKeeper
- Part de marché parfois estimée à 90% sur le segment des DVCS utilisés par la communauté logiciel libre
- Exemples de projets gérés : Linux (!), Gnome, Eclipse, KDE, X.org, Qt, Perl, Debian, Android, Facebook, Twitter, Google...

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

UNIVERSITÉ DE LORRAINE
IUT nancy Charlemagne

Possibilités

- Les VCS travaillent principalement sur les fichiers texte (`.txt`, `.c`, `.java`, `.xml`...)
- Les fichiers binaires (`.jpg`, `.doc`, `.pdf`...) peuvent également être intégrés mais ne peuvent prétendre qu'au versionage, pas à l'édition collaborative
- Que faut-il stocker dans un dépôt ?
  - **toutes** les ressources nécessaires à la construction d'un projet...
  - ...**à l'exception** de celles qui sont générées automatiquement (`.o` en C, `.class` en Java...)

Usages

- Utiliser un VCS suppose que les développeurs travaillent en concertation !
- Les VCS supposent que les développeurs ne modifient pas la même partie d'un même fichier
- Les VCS peuvent fusionner deux modifications relatives à un même fichier si elles concernent des parties différentes
- Dans le cas contraire, un *conflit* est généré et doit être réglé manuellement (par les développeurs)

Principe de fonctionnement d'un dépôt

- Création d'un dépôt (*repository*) vide
- Alimentation du dépôt par l'intermédiaire de *commits*
  - ensemble de modifications de données, suite aux manipulations des fichiers du projet (création, édition, suppression, renommage...)
  - *log* associé : commentaire sur la nature des modifications
  - méta-informations : identifiant de *commit*, auteur, date

Différents niveaux de stockage

Distant

- **Dépôt distant** *(remote repository)*

Local

- **Dépôt local** *(local repository)*
- **Index**
- **Répertoire de travail** *(working directory)*

Différents niveaux de stockage

- *Répertoire de travail*
  - contient la copie locale des sources du projet
  - contient, à sa racine, le répertoire `.git` de configuration
- *Index*
  - espace temporaire utilisé pour préparer la transition de données entre le répertoire de travail et le dépôt local
  - permet de choisir quel sous-ensemble de modifications, présentes dans le répertoire de travail, répercuter dans le dépôt local lors d'un *commit*

Différents niveaux de stockage

- *Dépôt local*
  - contient la totalité de toutes les versions de tous les fichiers du projet, par l'intermédiaire des *commits*
  - contient toutes les méta-informations : historique, *logs*, *tags*...
  - propre à un utilisateur donné
- *Dépôt distant*
  - est intrinsèquement similaire à un dépôt local
  - configuré et déployé pour pouvoir être partagé entre utilisateurs

Principe de fonctionnement intrinsèque

- Contrairement à d'autres VCS, Git s'intéresse aux **contenus**, pas aux fichiers en tant que tels
- Les noms de fichiers, les dates de modification, n'interviennent donc pas directement pour déterminer les modifications réalisées depuis un *commit* donné
- Git calcule pour chaque fichier une *signature* SHA-1 lui permettant de détecter des changements de contenu
- Les noms de fichiers, les dates associées, ne sont considérées que comme des méta-informations

SHA-1

Définition

- Fonction de hachage cryptographique conçue par la NSA
  - prend en entrée un texte de longueur maximale 2^{64} bits, soit environ 2.3×10^{18} caractères ($\sim$ 2.3 Eo)
  - produit une signature sur 160 bits, soit 20 octets, soit 40 caractères hexadécimaux ($\sim 1.5 \times 10^{48}$ possibilités)
- Exemples

```
% echo salut | sha1sum
```

```
3775e40fbea098e6188f598cce2a442eb5adfd2c –
```

```
% echo Salut | sha1sum
```

```
06d046c7fefde2a0514cb212fd28a5a653d8137e –
```

SHA-1

Signatures, aspects mathématiques

- Un *même* contenu fournit toujours la *même* signature
- D'un point de vue mathématique, il est possible que deux contenus différents génèrent une même signature (une *collision*)
- Mais en pratique, la probabilité est infinitésimale et peut être ignorée sans risque
- D'ailleurs, les 7 ou 8 premiers caractères d'une signature sont quasi systématiquement suffisants pour désigner sans ambiguïté un contenu...

SHA-1

Collisions et probabilités

- Il faudrait que 10 milliards de programmeurs fassent 1 *commit* par seconde pendant presque 4 millions d'années pour qu'il y ait 50% de chance qu'une collision se produise
- « *Il y a plus de chances que tous les membres d'une équipe soient attaqués et tués par des loups dans des incidents sans relation la même nuit* »

Usage des signatures SHA-1

- Sous Git, les signatures SHA-1 permettent d'identifier les contenus
  - de fichiers
  - de versions d'un projet (à travers ses fichiers)
  - de *commits* (en y associant des infos relatives à leur auteur)
- À chaque fois, la signature obtenue est supposée unique et constitue un identifiant fiable
- Cette gestion de signatures est à l'origine des performances de Git
- Elle lui permet aussi de garantir l'intégrité d'un projet dans un contexte distribué

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Principales commandes, par thème

Création

```
git init
git clone
```

Ajout

```
git add
git commit
```

Interrogation

```
git log
git show
git status
git diff
```

Opérations

```
git reset
git mv
git rm
git tag
```

Synchronisation

```
git push
git pull
```

git init

- Création d'un dépôt local vide
- Peut suffire pour gérer l'historique d'un projet pour un seul utilisateur...
- Crée une *branche* par défaut, appelée `master`
- Penser à ajouter un fichier `README` décrivant succinctement le projet

```
% git init
```

```
Initialized empty Git repository in /home/phil/tmp/.git/
```

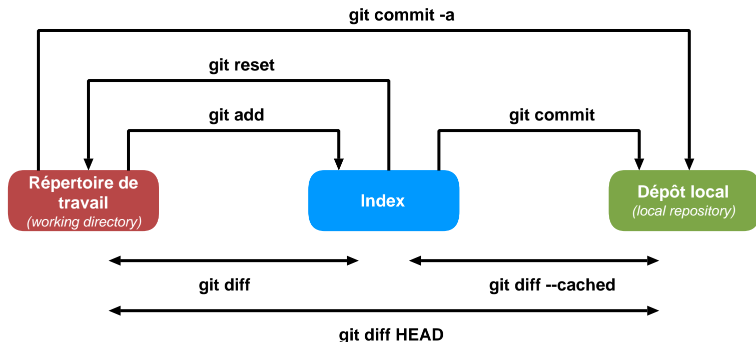
git clone

- Création d'un dépôt local à partir d'un dépôt existant (local ou distant)
- Met à jour la configuration du dépôt local pour garder une référence vers le dépôt distant
- Permet ensuite la communication entre les deux dépôts, typiquement par le biais des commandes `git push` et `git pull`

```
% git clone git@github.com:dosch/test.git
```

```
Cloning into test...
remote: Counting objects: 54, done.
remote: Compressing objects: 100% (54/54), done.
remote: Total 54 (delta 20), reused 0 (delta 0)
Receiving objects: 100% (54/54), 11.25 KiB, done.
Resolving deltas: 100% (20/20), done.
```

Index et commandes Git



`git add`

- Indexe le *contenu* des fichiers du répertoire courant passés en paramètre
- Rappel : Git travaille sur les contenus, pas sur les fichiers
- Conséquence : si des fichiers sont modifiés après leur indexation, c'est la version indexée qui sera répercutée dans le dépôt (et donc pas celle du répertoire courant)
- Un fichier qui a été indexé au moins une fois est ensuite suivi par Git (typiquement par `git status`)
- Mais l'indexation de chaque nouvelle version de ce fichier doit être réalisée par un nouveau `git add`

`git add`

- Il n'est pas nécessaire d'indexer en seule fois tous les changements d'un projet
- On peut donc typiquement utiliser `git add` sur un sous-ensemble des fichiers concernés
- Cela permet de créer par la suite des répercutions (*commits*) séparées

git commit

- Répercute le contenu de l'index dans le dépôt local
- L'index est ainsi complètement vidé suite au *commit*
- Un message de *commit* doit obligatoirement être défini à cette occasion
  - `git commit` : un éditeur externe sera lancé pour la saisie du message
  - `git commit -m "xxx"` : le message est fourni en ligne de commande

```
% git add README
% git commit -m "New feature described"
```

```
[master 7b01f01] New feature described
 1 files changed, 1 insertions(+), 1 deletions(−)
```

Que mettre dans un log ?

Techniquement...

- Une première ligne (obligatoire)
  - synthétise les changements
  - apparaît comme description courte du *commit*
- Une ligne vide (facultative si pas de description longue)
- Une description longue (facultative), de taille arbitraire

Que mettre dans un log ?

Et dans l'intention...

- Fondamentalement, doit expliquer le « pourquoi » d'un *commit*
- Trouver un « bon » message de log s'apparente à un exercice de style, presque un art...
- Intuitivement, doit être proche d'un résumé ($\sim$ une phrase) que l'on pourrait faire à un *collègue* (initié donc !)
- Exemples
  - Remplacement de conditionnelles imbriquées en `switch` pour améliorer la lisibilité (forme)
  - Suppression de la fonctionnalité DDFD_08 entravant la stabilité du code (fond)

`git commit`

- La commande `git commit -a` permet
  1. d'indexer automatiquement tous les fichiers qui ont déjà été indexés au moins une fois
  2. de répercuter l'index dans le dépôt local
- Les fichiers qui n'ont jamais été indexés (typiquement, les nouveaux fichiers du projet) ne peuvent donc pas être concernés

git log

- Affiche l'historique des *commits* du projet dans l'ordre chronologique inverse
- Affiche, pour chaque *commit*, son identifiant, l'auteur, la date et la première ligne du log
- `git log commit1...commit2` : affiche les logs entre 2 *commits* spécifiques (le premier *commit* fourni doit être le plus récent)

git log

`% git log`

```
commit 7b01f019730696862c434a81c3e3d8ac9014b183
Author: Philippe Dosch <Philippe.Dosch@loria.fr>
Date: Wed Feb 8 21:55:38 2012 +0100

    New feature described

commit 087aee7891fb79f7d5bf527e8ace994ad89a03d2
Author: Philippe Dosch <Philippe.Dosch@loria.fr>
Date: Wed Feb 8 21:53:55 2012 +0100

    Bug #7767 fixed

...
```

`git show`

- Affiche le détail d'un *commit* (ou d'autres entités Git)
- L'identifiant (court / long) correspondant doit être fourni en paramètre, sinon c'est le dernier *commit* qui est considéré
- Sur un *commit*, `git show` affiche en particulier la différence de contenu avec le *commit* précédent
  - lignes ajoutées : préfixées par un `+`
  - lignes supprimées : préfixées par un `-`

git show

`% git show`

```
commit 7b01f019730696862c434a81c3e3d8ac9014b183
Author: Philippe Dosch <Philippe.Dosch@loria.fr>
Date: Wed Feb 8 21:55:38 2012 +0100

    New feature described

diff --git a/README b/README
index 3942e23..6ab0c5e 100644
--- a/README
+++ b/README
@@ -1,2 +1,3 @@
 These styles are used for courses support at university.
+Support of git
```

git status

- Affiche des informations sur l'état du répertoire de travail et de l'index
- Permet de savoir ce que contient l'index (et donc ce qui sera concerné par le prochain *commit*)
- Permet de savoir quels fichiers sont suivis par Git et quels sont ceux qui ne le sont pas

git status

`% 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: README
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# README~
no changes added to commit (use "git add" and/or "git commit -a")
```

git diff

- Sans paramètre, affiche les différences de contenu entre le répertoire de travail et l'index
- `git diff commit1...commit2` : affiche les changements de contenus entre 2 *commits* spécifiques (le premier *commit* fourni doit être le plus récent)
- `git diff --cached` : différences entre l'index et le dernier *commit*
- `git diff HEAD` : différences entre le répertoire de travail et le dernier *commit*

git diff


```
% git diff
```

```
diff --git a/phil/inter.c b/phil/inter.c
index 2e1e492..2aa37fc 100644
--- a/phil/inter.c
+++ b/phil/inter.c
@@ -42,8 +42,8 @@ void inter_init(Interprete *t)
 int inter_generer(int maxniveau, Grammaire *g, Interprete *t)
 {
   char *ch, *buffer;
- int nb; /* nb < Num */
- int j; /* j < taille_max_regle */
+ int nb; /* nb < Num */
+ int j; /* j<taille_max_regle */
   int k;
   int tailleBuffer, tailleMaxBuffer, lch; /* longueur de ch,ch0,Rule[i] */
   int max;
@@ -59,56 +59,48 @@ int inter_generer(int maxniveau, Grammaire *g, Interprete *t)
   ch = tools_nouvelle_chaine(lch);
   strcpy(ch, g->axiome);
```

`git tag`

- Associe une balise (une étiquette textuelle) à un *commit*
- `git tag xxx` : associe le tag `xxx` au dernier *commit* réalisé
- `git tag` : liste tous les tags existants
- Intérêt : identifier un *commit* particulier plus facilement qu'à partir de sa signature SHA-1
- Exemples typiques de balises : v1.0, prod2.0, final4.4...

git reset

- Supprime des modifications effectuées dans l'index ou le dépôt local
- À utiliser avec précaution, certaines suppressions deviennent irrévocables...
- Peut souvent être remplacé avantageusement par un nouveau *commit*...
- Un exemple utile toutefois pour rétablir l'état du répertoire de travail au dernier *commit* effectué (et supprimer ainsi toutes les opérations effectuées depuis)

  ```
  git reset --hard
  ```

`git mv`

- Permet de déplacer ou de renommer un fichier ou répertoire
- L'historique de la ressource concernée est alors conservé
- À utiliser plutôt qu'un simple `mv` système qui ne permet pas la conservation de l'historique
- Le changement est répercuté dans l'index (et nécessite ensuite d'être répercuté par un *commit*)

`git rm`

- Efface un fichier ou un répertoire physiquement, ainsi qu'au niveau du suivi Git
- À utiliser plutôt qu'un simple `rm` système qui n'informerait pas Git de la suppression
- Le changement est répercuté dans l'index (et nécessite ensuite d'être répercuté par un *commit*)

git pull

- Récupère les changements du dépôt distant et les fusionne dans le dépôt local et le répertoire de travail
- Peut d'ailleurs être utilisé pour récupérer des changements de n'importe quel dépôt distant...
- À utiliser avant de propager les changements du dépôt local vers le dépôt distant (`git push`) s'il y eu des changements sur le dépôt distant

git push

- Propage les changements du dépôt local vers le dépôt distant

```
% git push
```

```
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 356 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:dosch/test.git
   ff367fc..f20eb85 master -> master
```

Récapitulatif des commandes fréquentes

- `git init` : création d'un dépôt local vide
- `git clone` : création d'un dépôt local à partir d'un dépôt existant (local ou distant)
- `git add` : « indexe » des fichiers en prévision d'un *commit*
- `git commit` : répercute les changements de l'index dans le dépôt local, sous forme d'un *commit*
- `git log` : examine l'historique du projet
- `git show` : affiche un objet (un *commit* par exemple)
- `git status` : affiche le status du répertoire de travail

Récapitulatif des commandes fréquentes

- `git diff` : affiche les différences entre le répertoire de travail et l'index
- `git tag` : associe une balise à un *commit*
- `git reset` : supprime des modifications effectuées dans l'index ou le dépôt local
- `git mv` : déplace des fichiers
- `git rm` : supprime des fichiers
- `git pull` : répercute les changements du dépôt distant vers le dépôt local
- `git push` : répercute les changements du dépôt local vers le dépôt distant

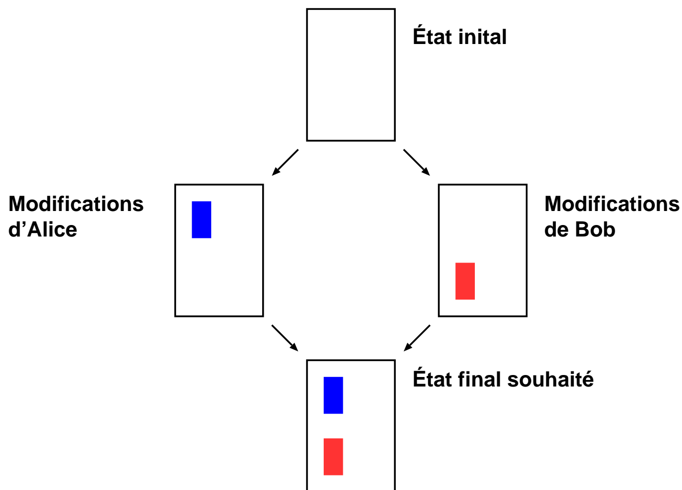
Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Introduction

- Si les modifications de différents contributeurs portent sur des fichiers différents, la situation ne pose pas de problème...
- Si les modifications sont relatives à un même fichier
  - si les portions modifiées sont différentes, il n'y a toujours pas de problème
  - si une même portion a été modifiée par plusieurs contributeurs, il y a un *conflit* (à résoudre manuellement)

Cas idéal



Cas idéal

Du côté de chez Alice...

```
alice% emacs README
alice% git add README
alice% git commit -m "Minor updates"
```

```
[master c70cf61] Minor updates
 Committer: Alice <alice@merveilles.com>
 1 files changed, 2 insertions(+), 1 deletions(-)
```

```
alice% git push
```

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 533 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To /var/git/test.git
   c70cf61..8f999f1 master -> master
```

Cas idéal

Du côté de chez Bob...

```
bob% emacs README
bob% git add README
bob% git commit -m "Bug fixed"
```

```
[master 70fcf61] Bug fixed
 Committer: Bob <bob.leponge@cartoon.net>
 1 files changed, 2 insertions(+), 1 deletions(-)
```

```
bob% git push
```

```
To /var/git/test.git
 ! [rejected] master -> master (non-fast-forward)
error: failed to push some refs to '/var/git/test.git'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes (e.g. 'git pull') before pushing again. See the
'Note about fast-forwards' section of 'git push --help' for details.
```

Cas idéal

Du côté de chez Bob...

```
bob% git pull
```

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /var/git/test
   8f999f1..0455941 master -> origin/master
Auto-merging README
Merge made by recursive.
 README | 1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

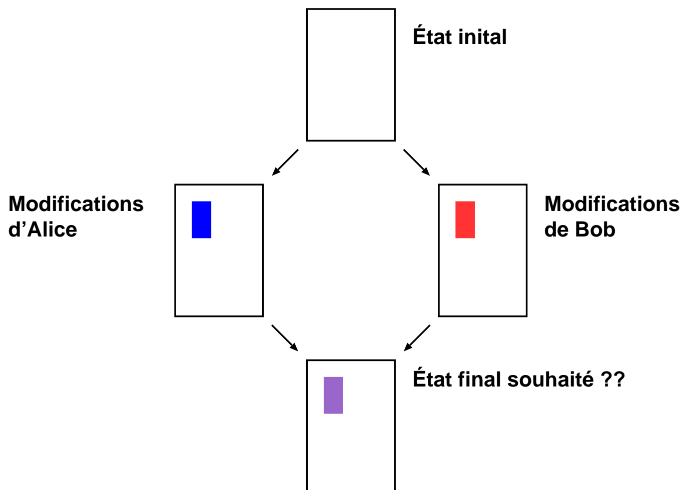
Cas idéal

Du côté de chez Bob...

```
bob% git push
```

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 592 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To /var/git/test.git
   0455941..78b618f master -> master
```

Conflit



Conflit

Du côté de chez Alice...

```
alice% emacs README
alice% git add README
alice% git commit -m "Minor updates"
```

[master c70cf61] Minor updates
Committer: Alice <alice@merveilles.com>
1 files changed, 2 insertions(+), 1 deletions(−)

Conflit

Du côté de chez Bob...

```
bob% emacs README
bob% git add README
bob% git commit -m "Bug fixed"
```

```
[master 70fcf61] Bug fixed
 Committer: Bob <bob.leponge@cartoon.net>
 1 files changed, 2 insertions(+), 1 deletions(-)
```

```
bob% git pull
```

```
remote: Counting objects: 5, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /var/git/test
   55ad751..c70cf61 master -> origin/master
Auto-merging README
CONFLICT (content): Merge conflict in README
Automatic merge failed; fix conflicts and then commit the result.
```

Conflit

Du côté de chez Bob...

- Si la même portion de code est modifiée de part et d'autres, un conflit est généré
- Le fichier en cause contient une zone délimitée par des chevrons

```
% cat README
```

```
<<<<<<< HEAD
Dessus...
Yopla
=======
Yopla
Dessous...
>>>>>>> c70cf6111a19192fed3b7aebd1e0a3e7088b28d0
```

Conflit

Du côté de chez Bob...

- Il faut alors éditer la zone en question, supprimer les chevrons, la ligne séparatrice (composée de =)
- Il faut ensuite répercuter le changement et on peut au final se synchroniser avec le dépôt distant

Conflit

Du côté de chez Bob...

```
bob% git add README
bob% git commit -m "Conflict fixed"
```

```
[master 8f999f1] Conflict fixed
```

```
bob% git push
```

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 533 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To /var/git/test.git
    c70cf61..8f999f1 master -> master
```

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Fichiers de configuration

- *Au niveau projet*
  Fichier `.git/config` à la racine du projet
- *Au niveau utilisateur*
  Fichier `~/.gitconfig`
- *Au niveau système*
  Fichier `/etc/gitconfig` (rarement utilisé)

Configuration utilisateur

- Positionnement du nom utilisateur
  `git config --global user.name "Philippe Dosch"`
- Positionnement de l'adresse mail
  `git config --global user.email "dosch@loria.fr"`
- Sorties en couleurs
  `git config --global color.ui "auto"`

Fichiers à ignorer

- Lors de commandes du type `git status` affiche des avertissements sur les fichiers qui n'ont jamais été indexés
- Et certains fichiers ne sont jamais intégrés dans un projet (les fichiers temporaires, les résultats de compilation, les sauvegardes...)
- Il est possible d'indiquer à Git d'ignorer ces fichiers
  1. par un fichier `.gitignore`, à placer à la racine du projet : ce fichier pourra être suivi et partager avec les autres membres du projet
  2. grâce du fichier `.git/info/exclude` : fichier propre au projet, mais qui ne sera pas partagé avec les autres membres du projet

Fichiers à ignorer

- Quel que soit le fichier utilisé, la syntaxe est la même
- On peut y placer des noms de fichiers (un par ligne)
- Le caractère * est autorisé, permettant de désigner facilement des familles de fichiers (typiquement sur l'extension)

Répertoires vides

- Il arrive dans certains cas que l'on souhaite ajouter un répertoire vide (sans contenu) dans un dépôt
- Problème : Git ne gère que les contenus de fichiers, ce type de répertoire n'a pas de fichier et ne peut donc pas être géré en l'état
- Une convention : définir un fichier vide nommé `.gitkeep` dans le répertoire pour qu'il puisse être pris en compte

Sommaire

1 Introduction
2 Les bases de Git
- Principes liés à Git
- Commandes essentielles de Git
- Exemples de travail collaboratif
- Configuration de Git

3 Gestion des projets à l'IUT
- Authentification SSH
- Gérer un projet sur Redmine (IUT)

4 Git : et encore...
- Outils liés à Git
- Autres commandes intéressantes
- Trucs et astuces

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Problématique

- De nombreux serveurs Git utilisent une authentification par clés publiques SSH
- Si le répertoire `~/.ssh` existe et qu'il contient des fichiers `id_rsa` et `id_rsa.pub`, une clé existe déjà !
- Sinon, il faut en générer une pour pouvoir communiquer avec les serveurs...

Génération de clé SSH RSA

- Pour générer une clé, utiliser `ssh-keygen`
  - sous Linux : en standard dans le package `openssh-client`
  - sous Windows : installer MSysGit
  - sous Mac : en standard dans MacOS
- Lors de la génération, une *passphrase* est demandée
  - c'est une sorte de mot de passe (ne pas reprendre le mot de passe de connexion)
  - cette *passphrase* sera demandée lors des connexions SSH

Utilisation d'une clé SSH

- Pour pouvoir communiquer *via* SSH avec un serveur, il est nécessaire d'y déposer sa clé publique (le contenu du fichier `id_rsa.pub`)
- Les serveurs proposent généralement d'effectuer ce dépôt *via* une interface Web
- À chaque connexion SSH, la *passphrase* sera demandée
- Pour éviter de taper la *passphrase* à chaque fois, utiliser `ssh-agent`
  - sous Linux et Mac : en tapant `ssh-add`
  - sous Windows : voir https://help.github.com/articles/working-with-ssh-key-passphrases

Clé SSH et multi-comptes

- Lorsqu'on est susceptible de se connecter depuis plusieurs ordinateurs (ou OS) à un serveur Git, plusieurs solutions
  - générer une clé SSH sur chacun de ses comptes et importer chaque clé publique sur le serveur
  - générer une clé SSH sur le premier compte, la copier sur les autres comptes et importer la partie publique sur le serveur
- Il peut être sain de travailler avec plusieurs clés SSH et ainsi ne pas mettre tous ses œufs dans le même panier...

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Informations générales

- Adresse du serveur, accessible également depuis l'extérieur
  `https://redmine.iutnc.univ-lorraine.fr`
- Tous les projets seront gérés et rendus sur ce serveur
- Les tuteurs s'y rendront donc pour les récupérer...
- Il est nécessaire de s'y connecter **rapidement** une première fois pour y rentrer ses infos
- Cette étape est **nécessaire** pour que les projets tuteurés soient créés

Connexion au serveur

- S'assurer d'avoir une clé SSH
- Se connecter une première fois sur le serveur Redmine
- Saisir ses informations personnelles
- Y importer sa ou ses clés publiques

Gestion des projets

- Chaque projet tuteuré sera géré par un projet Redmine comprenant
  - un journal des événements du projet
  - un outil de gestion de tickets (gestion des bugs, fonctionnalités à implanter, répartition des rôles...)
  - un porte-documents (pour y déposer de la documentation externe)
  - un wiki (pour y gérer la documentation interne)
  - ...et un dépôt git !
- Les projets Redmine seront créés par le département une fois que tous les étudiants concernés se seront connectés au moins une fois

Accès à un dépôt Git sous Redmine

- Les dépôts Git seront accessibles *via* SSH à `git@webetu.iutnc.univ-lorraine.fr:`*`projet`* où *`projet`* est l'identifiant du projet sous Redmine
- Aucun *login* n'est donc nécessaire, l'authentification se fait grâce à la clé SSH...
- Un premier utilisateur devra initialiser le dépôt
- L'initialisation faite, les autres utilisateurs pourront cloner le projet

Communication avec un dépôt Git sous Redmine

- Le premier utilisateur doit
  1. créer un dépôt local Git et y faire au moins un *commit*
     ```
     cd projet
     git init
     git add .
     git commit -m "Premier commit"
     ```
  2. le propager vers le dépôt Git sous Redmine
     ```
     git remote add origin git@webetu.iuta.univ-nancy2.fr:projet
     git push -u origin master
     ```
- Les autres membres ont juste besoin de cloner le dépôt
  ```
  git clone git@webetu.iuta.univ-nancy2.fr:projet
  ```

UNIVERSITÉ DE LORRAINE | IUT Charlemagne nancy

Utilisation de Git dans le cadre du projet

- Git est utilisable
  - en ligne de commande
  - intégré aux menus systèmes (sous Windows)
  - *via* les IDE traditionnels (Eclipse, Netbeans, Xcode...)
- Sous Eclipse, Git est accessible *via* le plugin Egit
- Les commandes les plus courantes sont alors disponibles dans des menus tels que
  - Team
  - Compare with
  - Replace with

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Outils liés à Git

- `gitk` : un navigateur graphique de dépôt Git (*i.e.* lecture)
- `git gui` : un *front-end* graphique pour dépôt Git (*i.e.* lecture et écriture)
- `gitstats` : un outil de génération de statistiques pour dépôt Git

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Autres commandes intéressantes

- `git checkout` : rétablit le projet dans un de ses états antérieurs (commit, branche, tag...)
- `git blame` : affiche l'auteur et la révision de chaque ligne d'un fichier
- `git remote` : gestion des dépôts distants (centraux ou d'autres utilisateurs)
- `git bisect` : permet de localiser par dichotomie un commit ayant introduit un bug
- `git gc` : compacte le dépôt Git (à utiliser de temps en temps pour gagner de la place)
- `git grep` : recherche d'expression régulière dans les fichiers suivis par Git

Sommaire

1. Introduction
2. Les bases de Git
   - Principes liés à Git
   - Commandes essentielles de Git
   - Exemples de travail collaboratif
   - Configuration de Git
3. Gestion des projets à l'IUT
   - Authentification SSH
   - Gérer un projet sur Redmine (IUT)
4. Git : et encore...
   - Outils liés à Git
   - Autres commandes intéressantes
   - Trucs et astuces

Trucs et astuces

- Première propagation d'un dépôt local vers un dépôt central distant
  ```
  git remote add origin url
  git push -u origin master
  ```
- Indexation partielle de fichiers (pour répartir les modifications d'un fichier sur plusieurs commits)
  ```
  git add -p
  ```
- Suppression de tous les fichiers non suivis et ignorés
  ```
  git clean -fx
  ```

Trucs et astuces

- Quelles modifications depuis les 15 derniers jours ?
  `git whatchanged --since="2 weeks ago"`
- Changer le commentaire du dernier commit
  `git commit --amend -m "Le nouveau commentaire."`
- Compter le nombre de commits par contributeur
  `git shortlog -sn`
- Créer une archive ZIP de son projet
  `git archive --format=zip HEAD > projet.zip`

Liens

- *Homepage* : `http://git-scm.com/`
- *Livre en français* : `http://progit.org/book/fr/`
- *Github* (hébergement de projets) : `https://github.com/`
- *Bitbucket* (hébergement de projets) : `https://bitbucket.org/`
- *Git interactif* : `http://ndpsoftware.com/git-cheatsheet.html`