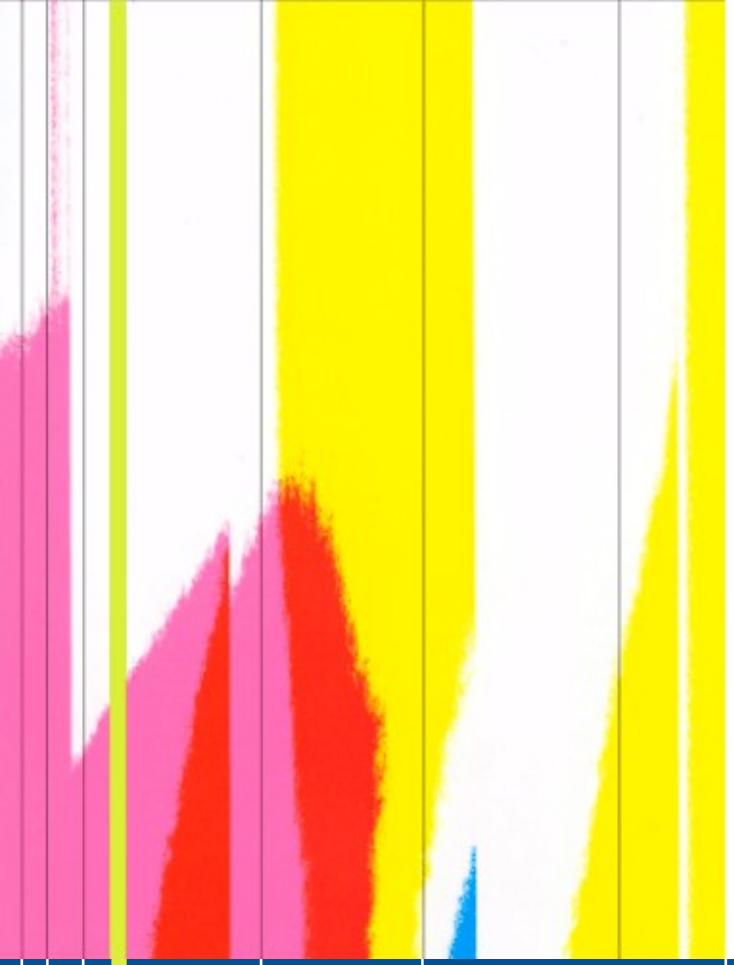


GIT

Brice Goglin
EPI Runtime

Séminaire SED – 11 mai 2010

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
BORDEAUX - SUD-OUEST

Planning

- Introduction
  - Pourquoi des systèmes décentralisés ?
  - Présentation rapide de GIT
  - Exemples comparant SVN et GIT
- Entrons dans les détails
  - Manipulations de base
  - Committer
  - Les branches
  - Accès à un dépôt distant
  - Au secours !
- A la carte
  - Détails sur les objets et dépôts
  - L'Index
  - Opérations avancées
  - GIT SVN

Introduction

Concepts de base

- Objectifs des systèmes de gestion de révisions/sources
  - Garder un historique
    - Pour retourner à une version précédente
    - Pour voir d'où vient une ligne
    - Pour trouver qui a cassé une fonctionnalité
  - Gérer les travaux concurrents de différents utilisateurs
    - Pas forcément en publiant immédiatement tous leurs travaux

- Des tonnes de solutions
  - Centralisées
    - CVS, SVN, …
  - Décentralisées
    - GIT, Mercurial, Darcs, Bazaar, BitKeeper, ...

Les limites du modèle centralisé

- Comment travailler dans le train ou dans l'avion ?
  - Accès à l'historique sans accès au dépôt ?
  - Développement de plusieurs patchs indépendants ou incrémentaux
    - *Oui, c'est important de faire plein de petits commits indépendants*
- Comment gérer ses différents travaux en cours ?
  - Profiter des avantages des commits intermédiaires...
    - Sauvegarder les étapes d'avancement du travail
      - Sans sauver plein de patchs manuellement ?
  - … sans en avoir les inconvénients
    - Dépôt public qui ne compile/marche pas tout le temps ?
  - Créer plein de branches ?

Deux modèles décentralisés répandus

- *Push*
  - Tous les développeurs développent dans leur coin puis poussent leurs changements dans un dépôt central
  - Ne passe pas bien à l'échelle (en cas de conflits)
    - http://www.selenic.com/pipermail/mercurial/2008-July/020116.html
  - Pour des projets avec un nombre limité de développeurs (e.g. quelques dizaines) et pas trop hiérarchiques
    - ex : Glibc, X.org, Mesa, Puppet, ...
- *Pull*
  - Un chef prend quand il le souhaite les modifications publiées par ses sous-lieutenants dans leurs dépôts
    - Les sous-lieutenants prennent dans des dépôts de développeurs
  - Pour les projets hiérarchiques et/ou avec beaucoup de devs
    - ex : Linux

Et concrètement, ca change quoi ?

- Il y a souvent un dépôt central/principal malgré le modèle décentralisé
- Les autres dépôts sont surtout utilisés pour faire du travail temporaire dans un coin
  - Avant de le publier dans le dépôt central
    - Quand le code fonctionne
  - Parfois, c'est aussi pour maintenir une version modifiée
- On peut créer des branches locales facilement sans impliquer le serveur central
  - Pour gérer toutes nos tentatives intermédiaires et travaux indépendants simultanément

Histoire de GIT

- 2002 : Linux passe à BitKeeper, le développement est grandement amélioré
  - Linus torvalds peut prendre facilement les changements préparés par ses sous-lieutenants
- 2005/04 : BitMover arrête la licence libre de BitKeeper
  - Le dépôt de Linux n'évolue plus pendant plusieurs semaines
  - Linus Torvalds débute le développement de GIT
- 2005/06 : Linux passe à GIT
- 2007/02 : GIT 1.5 releasé
  - Gros effort d'ergonomie

Il paraît que c'est compliqué ?

« [Linus] is a guy who delights being cruel to people. His latest cruel act is to create a revision control system which is expressly designed to make you feel less intelligent than you thought you were. [...] So Linus is here today to explain to us why on earth he wrote a software tool which, eh, only he is smart enough to know how to use. »

Linus Torvalds

présentation de GIT chez Google en 2007

Réfléchir avant de passer un logiciel à GIT...

Lu récemment sur la mailing list d'un logiciel après le passage de SVN à GIT :

« Can we please go back to SVN? This thing is just not worth the learning curve. […] Git may have made you more efficient, but it is killing the rest of us. What is the reason you can't just use git-svn, and let the rest of us stop having to fight this monstrosity ? »

Et donc ?

- 33 commandes dans SVN 1.6, plus de 100 de GIT 1.7
- Des concepts **très différents**

- Une façon de développer très différente
  - Ne pas développer avec GIT comme on l'aurait fait avec SVN
  - **Créer une branche locale est trivial, il faut en abuser**
    - Sauver des commits pour y revenir plus tard
    - Pas besoin de sauver des patchs à la main
    - Pas besoin de plusieurs checkouts
    - Un seul checkout pour toutes les branches, corrections de bogues … en même temps
    - Transfert de commits facile d'une branche à l'autre

Exemple 1, avec SVN

- On travaille sur une amélioration, pas encore prête à être rendue publique
- Un bogue est rapporté, il faut le corriger rapidement
- Comment faire avec SVN ?

- Faire un deuxième checkout, corriger le bug dedans, puis updater le premier checkout où on développait
- Sauver le patch actuel (svn diff > fichier), restaurer la version non-modifiée (svn revert -R .), corriger le bug, réappliquer le patch en espérant éviter des conflits
  - Penser à réajouter les fichiers ajoutés/supprimés
- Ca ne passe pas à l'échelle...

Exemple 1, avec GIT

- On travaille sur une amélioration, pas encore prête à être rendue publique
- Un bogue est rapporté, il faut le corriger rapidement
- Comment faire avec GIT ?

- Committer vite-fait l'état actuel de son amélioration
  - On pourra le corriger plus tard
- Créer une branche de la version publique et la checkouter
- Corriger le bug et publier le correctif
- Reprendre la branche initiale
- Merger/Rebaser pour récupérer le correctif du bug si nécessaire

Exemple 2, avec SVN

- Travail sur une nouvelle fonctionnalité dans l'avion

- Sauver des versions intermédiaires
  - svn diff > patch1-quifaitXmaispasencoreY
- Committer les différentes étapes du travail
  - svn up -r <revision au moment du travail>
  - patch < patch1
  - svn up
  - Résoudre les conflits, notamment si ajout de fichiers
  - svn commit …
  - Recommencer pour chaque patch

Exemple 2, avec GIT

- Travail sur une nouvelle fonctionnalité dans l'avion

- Sauver des versions intermédiaires
  - git commit
- Ecraser une version précédente
  - git commit --amend
- Committer les différentes étapes du travail
  - git rebase
  - git push

Exemple 3

- On maintient une version modifiée d'un logiciel A
- Comment mettre à jour depuis une nouvelle version de A ?

- Avec SVN
  - Trouver la dernière révision X qu'on a déjà récupérée
  - Trouver la révision courante Y qu'on veut récupérer
  - svn merge URL -rX:Y .
  - svn commit

- Avec GIT
  - git pull A Y
    - Il découvre tout seul ce qu'il doit merger

Exemple 4

- On veut tester une modification sur différentes machines
- Avec SVN
  - On crée une branche pour un seul (ou quelques) commits ?
  - On envoie un/des patch(s) à la main sur les différentes machines ?
    - Ne pas oublier svn up à chaque fois...
- Avec GIT
  - Créer une branche puis git push ou pull vers/depuis chaque machine
    - Pas besoin de passer par un dépôt central

Résumé

- L'abus de branches n'est pas dangereux pour la santé du développement logiciel
  - Au contraire !
- Sauver des versions intermédiaires
- Travailler sur plusieurs choses en même temps
- Consulter/manipuler des branches locales et distantes en même temps
- Transférer facilement entre branches
  - Maintenir des branches stables
  - Récupérer rapidement un patch d'une autre branche
- Plus besoin d'accès au dépôt distant en permanence
  - Pratique et rapide
  - On peut travailler dans l'avion ou quand la forge est en rade !

Prêt à passer à GIT ?

- Beaucoup de projets passent à GIT par effet de mode ?
  - Est-ce une bonne idée quand on utilise uniquement des fonctionnalités de base ?
    - Le coût de formation des développeurs est-il justifié ?
- Ne pas oublier GIT-SVN
  - Passerelle permettant d'utiliser GIT par dessus un dépôt SVN
  - N'impose pas GIT à tous les autres développeurs
  - Fournit *quasiment* tous les avantages de GIT à celui qui utilise GIT-SVN
- A suivre : pleeeeeeiiiiiin de détails

Issue de secours
(au fond à gauche)

Manipulations de base

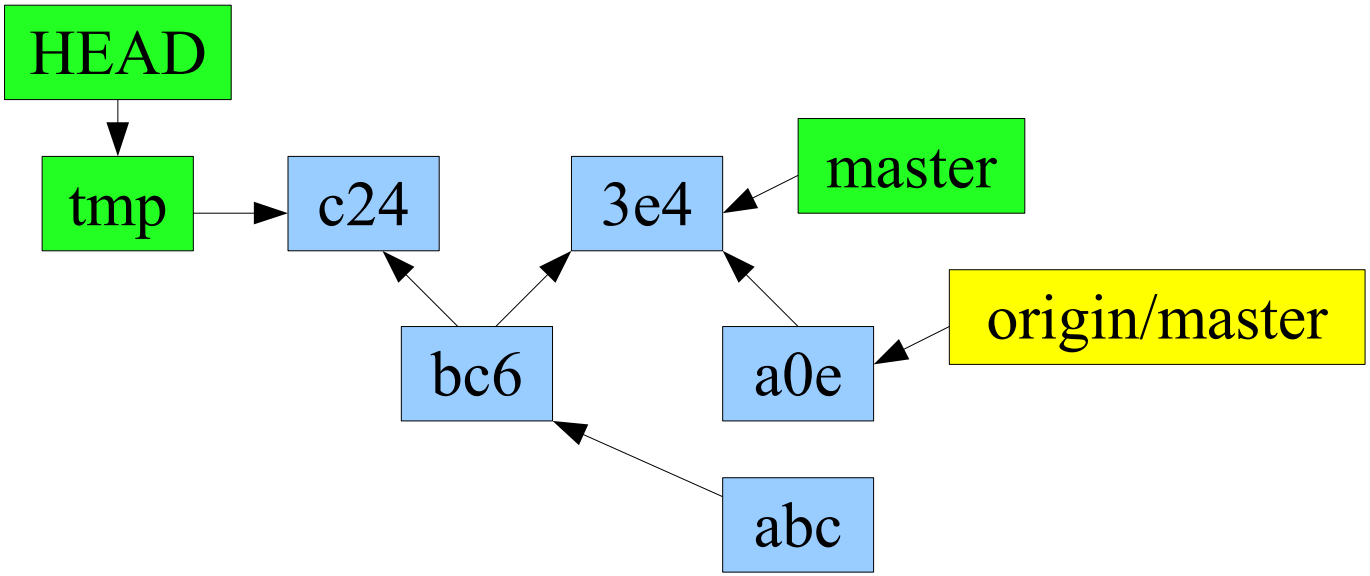
A quoi ressemble un commit ?

- Un commit est identifié par un hash (SHA1)
  - Des numéros de révisions à la SVN auraient peu de sens en décentralisé
- Le hash dépend du contenu, du message et du prédécesseur
  - Un **même** commit dans deux branches a deux hashs différents

```
$ git log
commit e02346148c5032d9e6f7d377bd0b3ada781d7446
Author: bgoglin <bgoglin@4b44e086-7f34-40ce-a3bd-00e031736276>
Date:   Tue Apr 6 15:36:17 2010 +0000

    Stop using HWLOC_NBMAXCPUS in Linux hwloc_linux_set/get_tid_cpubind

commit 235f84643e583d61f94ff32165472b0cdf1d69c6
Author: bgoglin <bgoglin@4b44e086-7f34-40ce-a3bd-00e031736276>
Date:   Tue Apr 6 15:35:53 2010 +0000

    Move HWLOC_NBMAXCPUS out of hwloc/config.h
```

A quoi ressemble une branche ?

- Une branche est un pointeur/étiquette sur un commit
  - Chaque commit pointe vers son prédécesseur
    - Ou vers plusieurs commits en cas de merge
  - Le pointeur avance si on committe dans la branche courante
- La branche courante HEAD pointe vers une branche



Identifier des commits

- Le dernier commit de la branche courante ou d'une autre
  - HEAD
  - mabranche
- L'avant-dernier et les précédents d'une branche
  - HEAD^, mabranche^^, ...
  - HEAD~3, mabranche~12, ...
- Plein d'autres moyens
  - HEAD@{yesterday}
  - mabranche@{June.1}
- Le hash SHA1 peut être tronqué tant qu'il reste unique
  - git le tronque souvent à 8 caractères

Consulter le log

- Log jusqu'à un certain point (en suivant les branches qui y mènent), jusqu'à HEAD par défaut
  - git log <identifiant>
- Log entre deux points (en suivant les branches qui les relient)
  - git log <identifiant1>..<identifiant2>
- Les identifiants peuvent être des hashs, noms de branches, tags, HEAD^, tmp~10, …

- Voir aussi --since, --until, --author, --grep, …
- Log concernant certains fichiers uniquement
  - git log .... -- monfichier monrepertoire/

Consulter des changements

- Voir un commit
  - git show <identifiant>
  - git show <identifiant> -- monfichier monrepertoire/
- Consulter des différences entre points
  - git diff <identifiant1>..<identifiant2>
  - git diff <identifiant1>..<identifiant2> -- monfichier monrep...

Trouver d'où vient un changement

```
$ git blame configure.ac
028288a8 (stordeur  2010-02-18 16:17:24 +0000   1) # General informations
028288a8 (stordeur  2010-02-18 16:17:24 +0000   2) AC_INIT(Open-MX,
028288a8 (stordeur  2010-02-18 16:17:24 +0000   3)       1.2.90,
028288a8 (stordeur  2010-02-18 16:17:24 +0000   4)       http://open-mx.org,
028288a8 (stordeur  2010-02-18 16:17:24 +0000   5)       open-mx)
028288a8 (stordeur  2010-02-18 16:17:24 +0000   6)
cc0414ad (bgoglin   2009-10-26 06:30:11 +0000   7) AC_PREREQ(2.59)
[...]

$ git blame 028288a8^ -- configure.ac
ab69af89 (bgoglin   2007-07-26 13:50:45 +0000   1) # Open-MX
a05a756b (bgoglin   2010-02-08 14:30:31 +0000   2) # Copyright Â© INRIA
ab69af89 (bgoglin   2007-07-26 13:50:45 +0000   3) #
ab69af89 (bgoglin   2007-07-26 13:50:45 +0000   4) # The development of
ab69af89 (bgoglin   2007-07-26 13:50:45 +0000   5) #
[...]
```

Committer

- Voir toutes les modifications locales actuelles
  - git diff HEAD
    - Ne pas oublier HEAD, sinon il faudra comprendre la notion d'index (voir plus loin)
- Committer toutes les modifications locales
  - git commit -a
  - git commit monfichier monrepertoire
  - Ne commite que les fichiers déjà connus de GIT
- Ca ajoute un commit au dépôt
  - puis avance le pointeur de la branche courante

Ajouter ou supprimer des fichiers

- Ajouter ou supprimer un fichier du dépôt
  - git add monfichier
  - git rm monfichier

- Renommer un fichier
  - git mv ancienfichier nouveau fichier
- En fait, git découvre automatiquement les renommages
  - Même si le fichier est un peu modifié entre temps !
    - mv foo bar ; vim bar ; git add bar ; git commit -a

- Modifications sauvées dans le dépôt lors de git commit
  - Elles sont enregistrées dans l'*index* entre temps

Consulter l'état courant du checkout

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README
#
# 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:   configure.ac
#
# Untracked files:
#   (use "git add <file>..." to include in what will
#
#       INSTALL
#       Makefile.in
```

Fichiers modifiés (après git add)

Fichiers modifiés (sans git add)

Fichiers inconnus par GIT

Fichiers committables par **`git commit -a`**

Annuler le dernier commit

- Annuler le dernier commit sans perdre les modifications
  - git reset HEAD^
- Annuler le dernier commit et les modifications des fichiers
  - git reset --hard HEAD^
- Par extension, ramener la branche courante à un autre commit
  - git reset --hard <identifiant>
- Annuler des commits sans perdre les modifications
  - git reset <identifiant>
- Eviter d'annuler un commit déjà publié (voir plus loin)

Corriger le dernier commit

- J'ai oublié un bout dans mon commit précédent ?
  - ou mon message de commit n'est pas bon ?

- Défaire puis refaire le commit ?
  - git reset HEAD^, git add si nécessaire, git commit
- Ou mieux, amender le commit précédent !
  - Juste corriger le message de commit
    - git commit --amend
  - Ajouter les derniers changements, dans tout ou un fichier
    - git commit -a --amend
    - git commit monfichier --amend

- Eviter de modifier un commit déjà publié...

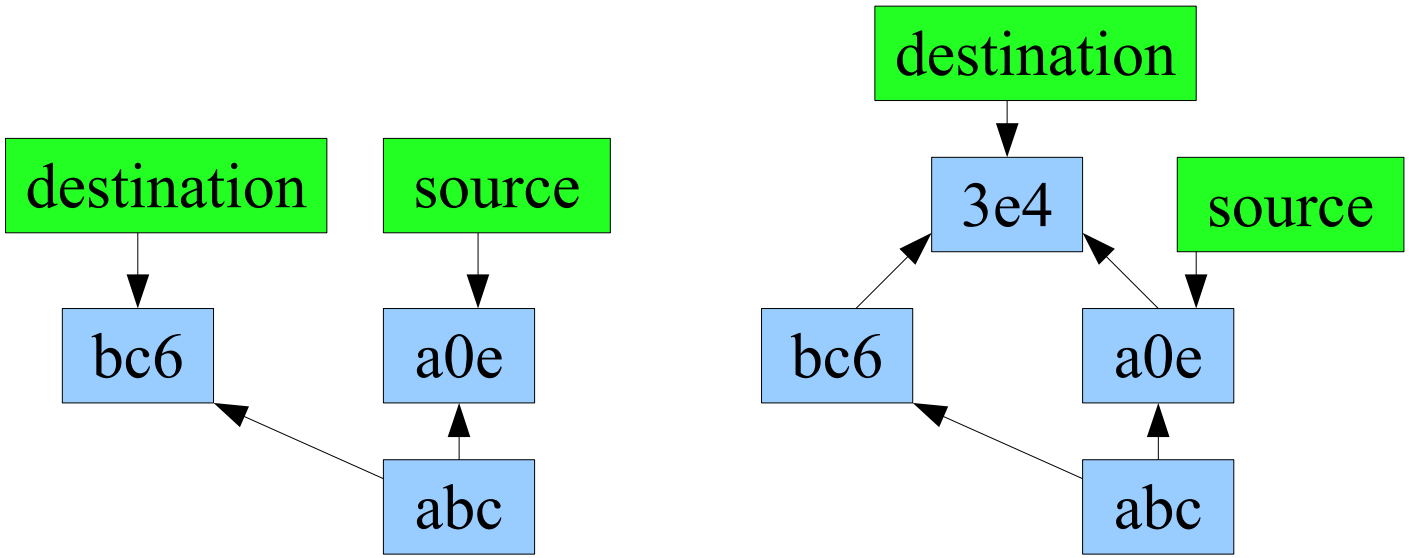
Les branches

Les branches

- Lister les branches locales
  - git branch -l
    - La branche courante (HEAD) est marquée par une étoile
  - git branch -rl pour les branches distantes
- Créer une branche à la position courante
  - git branch mabranche
  - Ou depuis un autre commit
    - git branch mabranch <identifiant>
  - Ne pas oublier de la checkouter ensuite !
- Checkouter une autre branche
  - git checkout mabranche
- Créer et checkouter d'un seul coup
  - git checkout -b mabranche [<identifiant>]

Merger des branches

- git checkout branchedestination
- git merge branchesource
- Crée d'un commit parent des deux branches
  - La branche courante avance à ce commit
  - La source ne bouge pas, mais devient un fils du nouveau

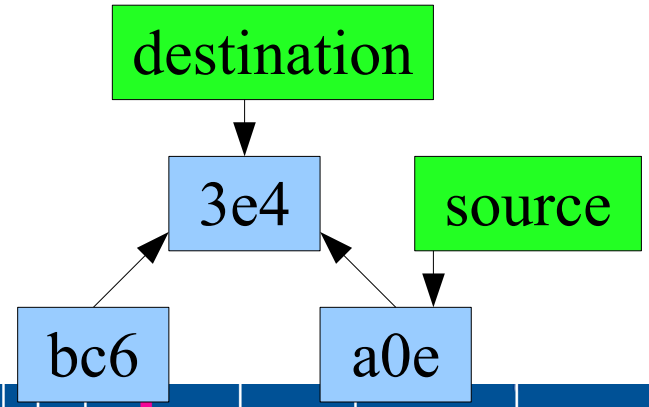


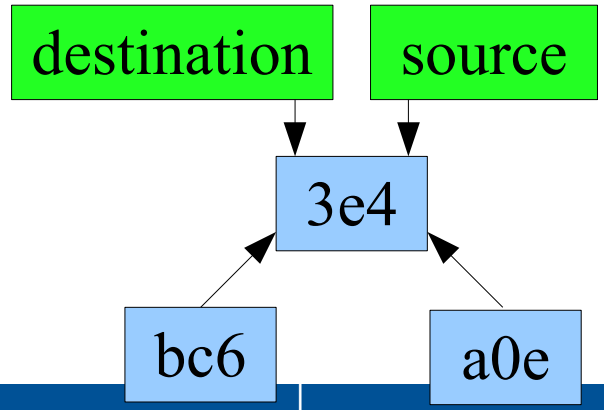
Merger des branches (2/2)

- Que s'est-il passé ?
  - On détermine les travaux depuis un ancêtre commun
  - On applique les travaux de la source dans la destination
    - La branche destination avance au nouveau commit
- L'historique des deux branches est conservé
  - Mais pas évidemment à observer...
    - git log fusionne les historiques par ordre chronologique
- Le commit du merge ne contient rien
  - à part des informations de merge
  - git diff <avantmerge>..<merge> montre les vraies différences
  - git show ne montre pas de code
    - sauf s'il a fallu résoudre un conflit
      - Montrer quelles lignes on a gardé de chaque branche

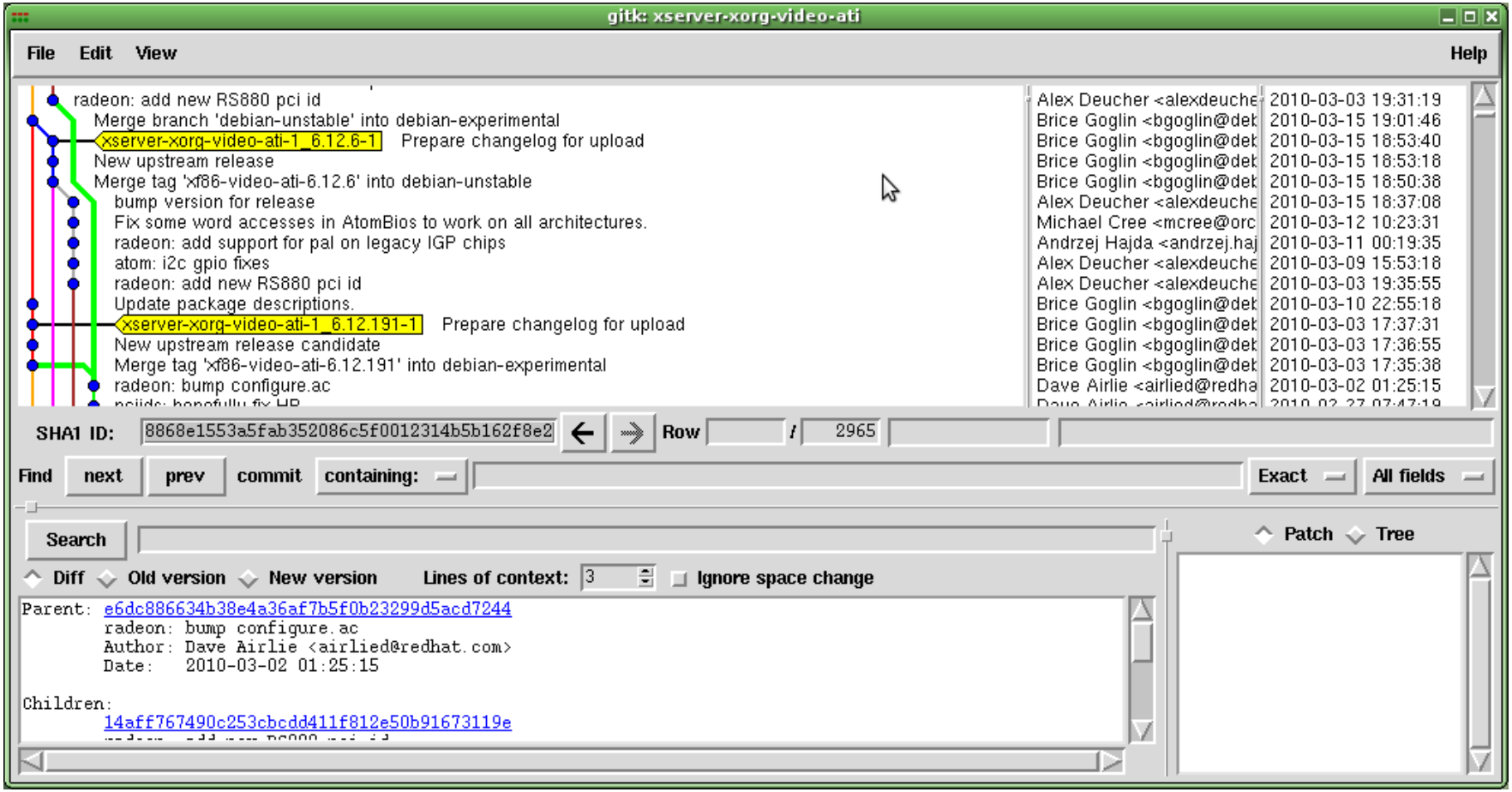
Fast-Forward ou non

- Et si une des branches est un ancêtre de l'autre ?
  - Ex: si je merge à nouveau ?
- Si on remerge source dans le merge, rien à faire
  - Le merge contient déjà source
- Si on merge le merge dans source, il faut que source contienne le merge
  - On avance le pointeur source jusqu'au merge
    - On parle de *Fast-Forward*
      - Juste un ajout de commits existants en haut de la branche





Voir l'arbre des branches - gitk



Résolution de conflits

- git tente de fusionner les travaux des deux branches
  - Avec différentes stratégies (configurable)
- En cas de conflit insoluble automatiquement
  - Le commit du merge n'est pas fait
  - Des chevrons sont insérés aux endroits problématiques
  - Les fichiers incriminés sont marqués en conflit
- L'utilisateur doit corriger les conflits
  - Enlever les chevrons
  - Marquer les fichiers devant être committés avec git add
    - Ou git rm si un fichier doit disparaître lors du merge
      - ex : fichier modifié dans une branche et effacé dans l'autre
  - Les fichiers fusionnés sans conflits sont déjà dans l'index
  - Committer le résultat avec git commit

Résolution de conflits (2/2)

```
$ git merge mabranche
Auto-merging utils/lstopo-xml.c
CONFLICT (content): Merge conflict in utils/lstopo-xml.c
Automatic merge failed; fix conflicts and then commit the result.

$ git status
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#       both modified:      utils/lstopo-xml.c

$ git commit
fatal: 'commit' is not possible because you have unmerged files.
Please, fix them up in the work tree, and then use 'git add/rm <file>' as
appropriate to mark resolution and make a commit, or use 'git commit -a'.

$ vim utils/lstopo-xml.c
[…]
$ git add utils/lstopo-xml.c

$ git status
# Changes to be committed
#       modified:      utils/lstopo-xml.c

$ git commit
```

Accès à un dépôt distant

Cloner un dépôt

- En général, on ne crée pas de dépôt vide
  - On veut checkouter un dépôt existant
- git clone git+ssh://bgoglin@myserver/git/myrepo

- On obtient un dépôt local avec le checkout correspondant
  - Tous nos commits affectent uniquement ce dépôt
  - Le dépôt d'origine s'appelle **origin**
  - La branche par défaut est checkoutée
    - **master** en général

Consulter les branches distantes

- Pour chaque dépôt lié, on connaît la liste des branches
  - Et on peut les consulter comme d'habitude
    - Les branches et commits sont rapatriés par git clone et git fetch

```
$ git branch -rl
origin/master
origin/mybranch
myotherorigin/master

$ git log origin/master
commit c179ebb8ca2dad2c40f647270fd84dafdd222a97
[…]

$ git show c179ebb8c
[…]
```

Modifier les branches distantes ?

- On peut checkouter les branches distantes
  - Mais on ne doit pas les modifier directement
  - Les branches distantes peuvent être modifiées uniquement lors de synchronisations avec le dépôt distant
    - Voir fetch/pull/push plus loin

```
$ git branch -rl
origin/master
origin/mybranch
myotherorigin/master

$ git checkout origin/master
Note: moving to 'origin/master' which isn't a local branch
If you want to create a new branch from this checkout, you may do so
(now or later) by using -b with the checkout command again. Example:
  git checkout -b <new_branch_name>
```

Lier branches locales et distantes

- Au lieu d'utiliser des branches distantes, on marque des branches locales comme suivant des branches distantes
  - On va modifier les branches locales puis propager au dépôt distant correspondant

- Par défaut, **master** suit **origin/master**
- Pour suivre une autre branche
  - git branch mabranchelocale --track origin/sabranche

Intermède : les fichiers de configuration

- .git/config contient la configuration du repository
  - Les dépôt, les branches, les options, …

```
$ cat .git/config
[...]
[remote "origin"]
        fetch = +refs/heads/*:refs/remotes/origin/*
        url = git://git.debian.org/git/pkg-xorg/driver/xserver-xorg-video-ati
[branch "debian-unstable"]
        remote = origin
        merge = refs/heads/debian-unstable
```

Intermède : les fichiers de configuration (2/2)

- On peut modifier .git/config à la main
  - Ou utiliser des commandes dédiées

```
$ git config --list
[...]
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
remote.origin.url=git://git.debian.org/git/pkg-xorg/driver/xserver-xorg-video-ati
branch.debian-unstable.remote=origin
branch.debian-unstable.merge=refs/heads/debian-unstable
```

- $HOME/.gitconfig peut contenir des variables globales à tous nos dépôts
  - git config --global user.name "Your name"
  - git config --global user.email you@domain.tld

Télécharger des modifications distantes

- git fetch origin
  - Télécharge tous les nouveaux commits du dépôt origin
  - Les stocke dans la base locale
  - Avance les pointeurs origin/<branch>
  - Permet de consulter en local le dépôt distant à jour

```
$ git fetch origin
remote: Counting objects: 195, done.
remote: Compressing objects: 100% (68/68), done.
remote: Total 159 (delta 117), reused 124 (delta 90)
Receiving objects: 100% (159/159), 34.14 KiB, done.
Resolving deltas: 100% (117/117), completed with 27 local objects.
From git+ssh://git.debian.org/git/pkg-xorg/app/xdm
   0fb13ca..19c7684  debian-unstable -> origin/debian-unstable
   6061722..7006d5c  upstream-unstable -> origin/upstream-unstable
From git+ssh://git.debian.org/git/pkg-xorg/app/xdm
 * [new tag]         xdm-1_1.1.9-2 -> xdm-1_1.1.9-2
```

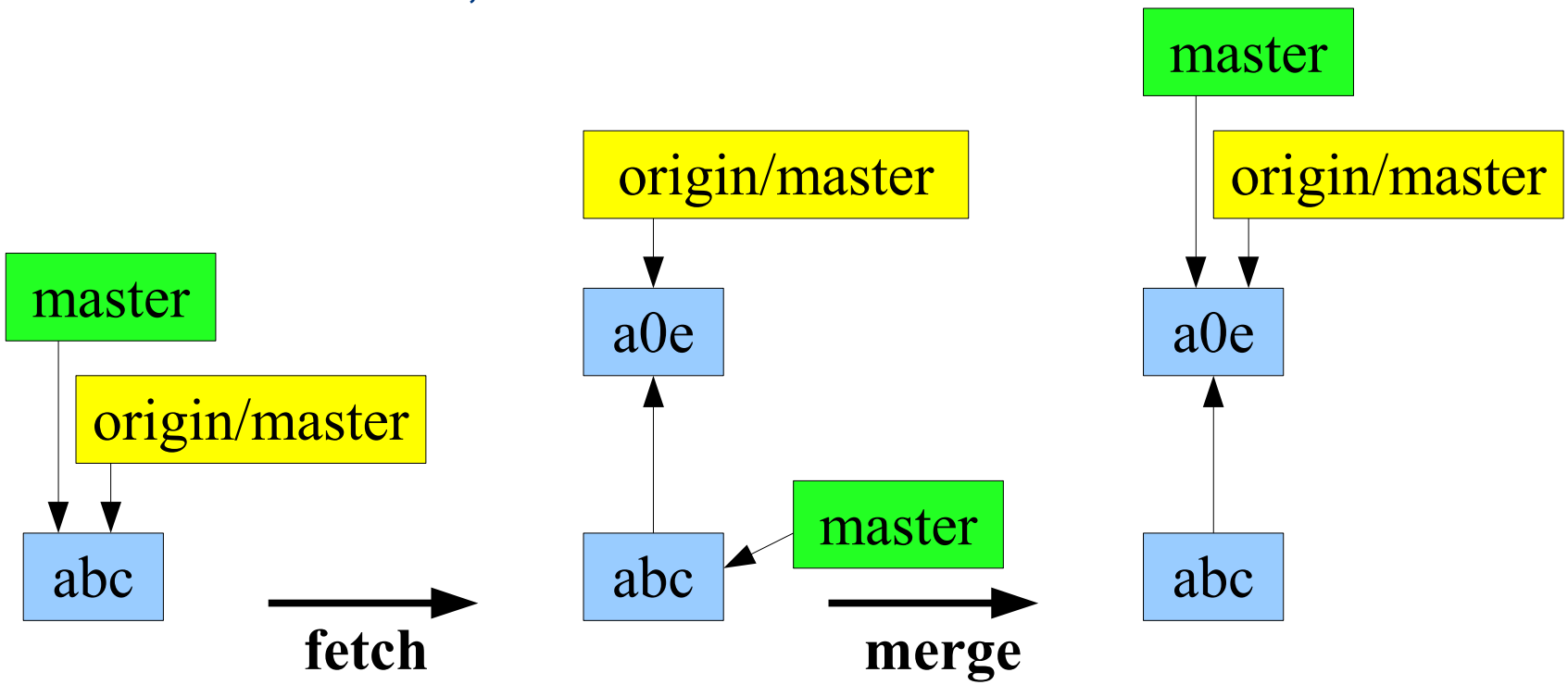
Récupérer des modifications distantes

- git fetch n'adapte pas les branches locales
- Entrer dans une branche nous prévient de son statut vis-à-vis de sa source distante (git status aussi)
  - Soit la branche locale est un ancêtre direct
    - On va pouvoir faire un *Fast-Forward*
  - Soit la branche locale a été modifiée différemment
    - Il va falloir merger ?

```
$ git checkout upstream-unstable
Switched to branch 'upstream-unstable'
Your branch is behind 'origin/upstream-unstable' by 38 commits,
 and can be fast-forwarded.

$ git checkout debian-unstable
Switched to branch 'debian-unstable'
Your branch and 'origin/debian-unstable' have diverged,
and have 1 and 50 different commit(s) each, respectively.
```

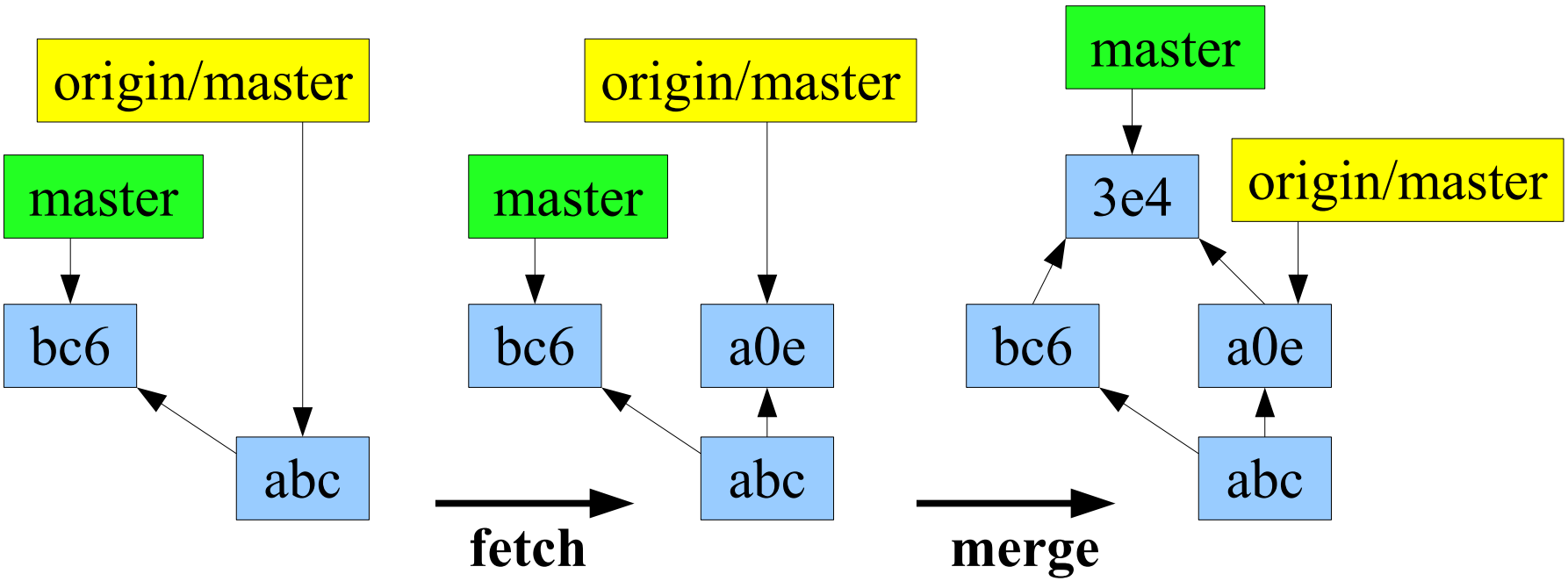
Récupérer des modifications distantes (2/3)

- git merge origin/mabranche
- Si pas de modification locale
  - Fast-Forward, tout va bien



Récupérer des modifications distantes (3/3)

- Si des modifications locales (divergence)
  - Merge de branches, comme précédemment



git pull – Rapatrier et appliquer des changts

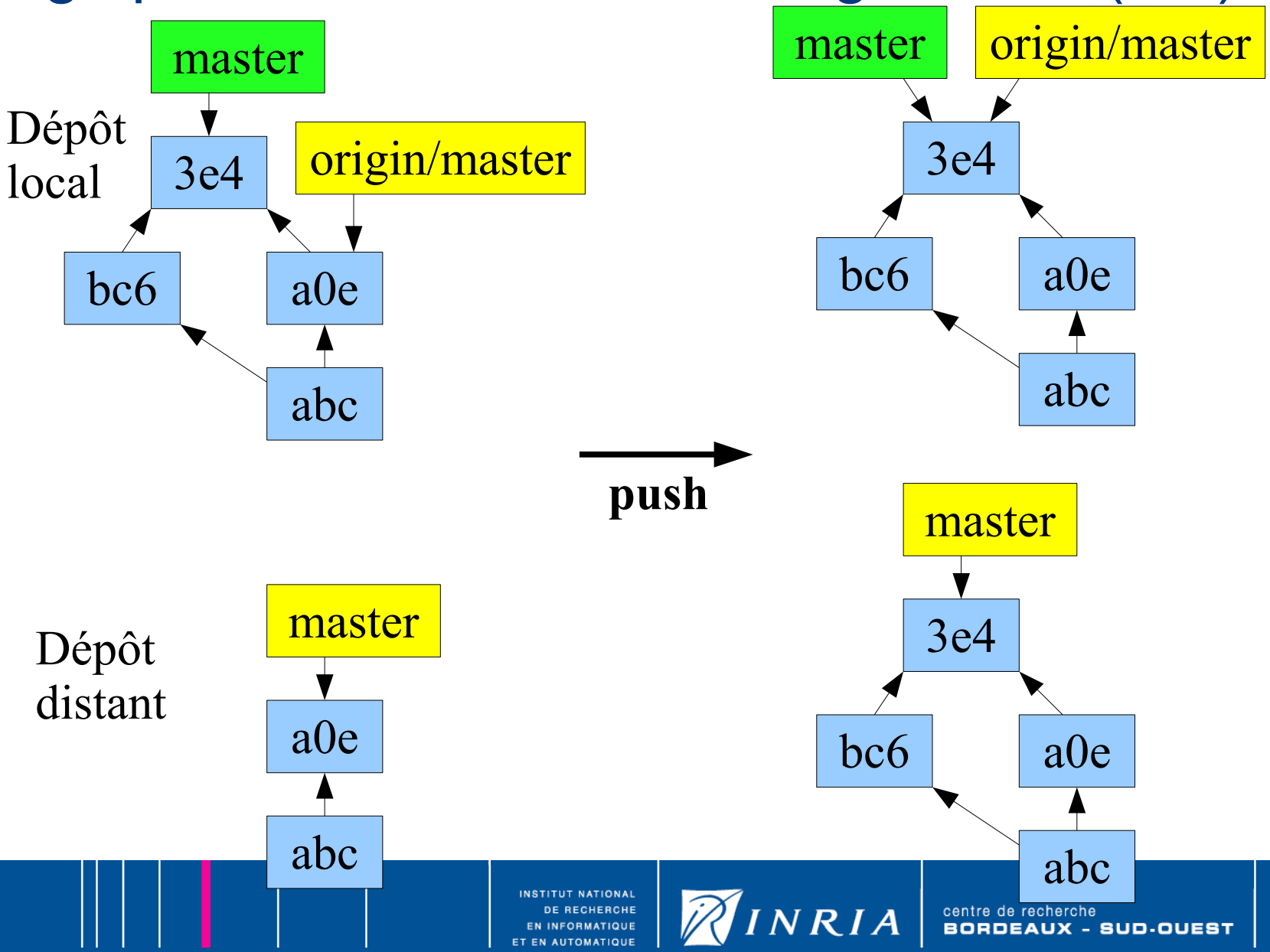
- pull = fetch + merge
- Merger la branche mabranche du dépôt origin dans la branche locale courante
  - git pull origin mabranche
- Par extension, merger la branche locale mabranche dans la branche courante
  - git pull . mabranche
- Avant de puller, penser au fait qu'un merge pourrait avoir lieu
  - git fetch pour observer origin/mabranche avant de merger ?

git push – Publier des changements

- git fetch/pull permet de récupérer des commits distants
  - Et mettre à jour des branches remote
- Comment pousser des commits dans d'autres dépots ?

- git push origin mabranche
  - Avance mabranche sur le dépôt origin au même commit que mabranche en local
  - Et envoie les commits intermédiaires
- git push origin mabranche:tabranche
  - Propage dans la branche distante tabranche

git push – Publier des changements (2/2)



Au secours !

J'ai tout cassé ! Comment je répare ?

- Pas sûr qu'un merge/pull a fait ce qu'on veut ?
  - git reset --hard HEAD^
    - Revient au commit précédent
  - git reset --hard
    - Annule le merge en cours
- Un rebase se passe mal ?
  - git rebase --abort
- J'ai perdu ma branche !
  - git reflog
    - Liste l'historique des modifications de HEAD
      - Permet de retrouver les identifiants précédents
  - git reset --hard <identifiant>

Quelques précautions quand on est pas sûr

- Faire des branches de sauvegardes avant une opération quand on est pas sûr
  - git branch save
  - <travail>
  - Si ca n'a pas marché
    - git reset --hard save
  - git branch -d save
  - C'est gratuit !
  - Ou noter le hash courant et ne pas créer de branche...
  - Ou utiliser git reflog pour retrouver les HEAD précédents
- Lire les messages affichés par les différentes commandes
  - Ils rappellent souvent quoi faire pour résoudre un conflit, préviennent de problèmes potentiels, …
- Ne pas oublier quand on a commencé un rebase

Conclusion préliminaire

Conclusion

- Un modèle très différent
  - Faire des branches pour tout et n'importe quoi
  - Modifier/annuler/déplacer/dupliquer des commits
  - Temps d'adaptation non négligeable
- Un modèle très performant
  - Rapide (offline)
  - Flexible (faire sa tambouille locale avant le push)

Liens

- Les manpages sont souvent bien faites
  - Modulo les termes très précis et donc un peu complexes
- http://eagain.net/articles/git-for-computer-scientists/
  - Introduit bien les termes et objets
- http://www.jukie.net/bart/blog/into-to-git-talk-2
  - Tutorial plus complet et très illustré
- http://git.or.cz
  - http://git.or.cz/gitwiki
- http://gitcasts.com

Merci de votre attention!

Brice.Goglin@inria.fr

Compléments à la carte

Détails sur les objets et dépôts

Dépôt vs. checkout

- En général, on a à la fois un dépôt et un checkout
  - dépôt dans .git, à côté des fichiers du checkout
  - C'est pour cela qu'on travaille en mode déconnecté

```
monfichier1
monrepertoire/
monrepertoire/monfichier2
[...]
.git/
.git/HEAD
.git/config
.git/objects/
.git/refs/tags/
[...]
```

```
HEAD
config
objects/
refs/tags/
[...]
```

- Un dépôt peut ne pas avoir de checkout
  - Dépôt central dans lequel on pousse des changements sans directement y développer
    - ex : Sur le gforge

Création d'un dépot

- git init --bare monrepo
  - Dépôt sans checkout (fichiers du .git/ uniquement)
  - Utilisable uniquement par fetch/pull/push depuis d'autres dépôt sans --bare
- git init monrepo
  - Dépôt avec checkout
    - A n'utiliser que quand c'est nécessaire
      - Attention si quelqu'un pushe dedans

- Une branche « **master** » est créée, avec rien dedans
  - L'équivalent du trunk de SVN
- On peut y travailler immédiatement

Lier un dépôt local à des dépôts distants

- On peut lier un autre dépôt distant
  - git remote add myotherorigin git+ssh://login@...

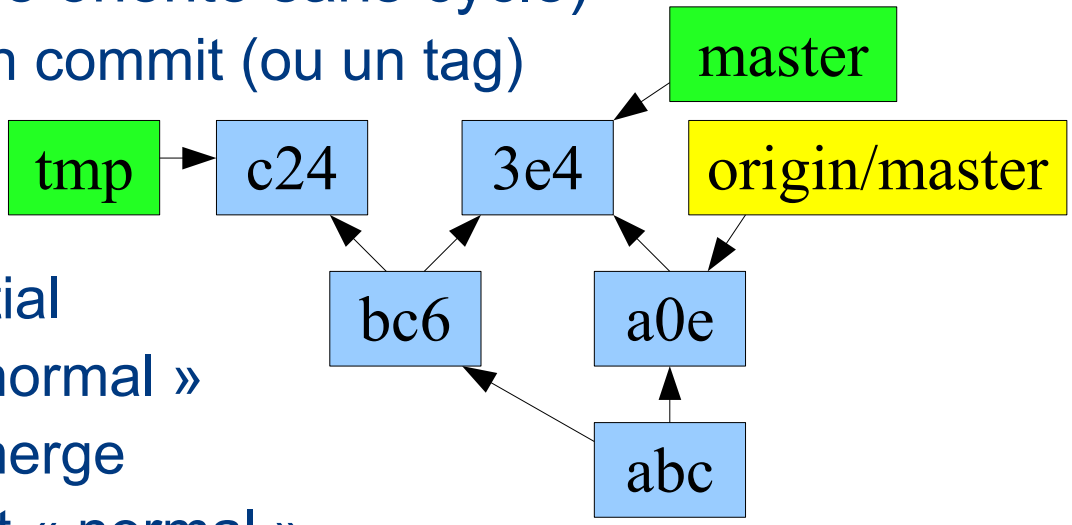
```
$ git remote
origin
myotherorigin
```

- Ensuite on récupère le contenu avec git fetch

- Les pull/merge utilisent un ancêtre commun
  - Qui peut être le commit initial (vide) de création du dépôt
    - Notamment juste après git init

Y a quoi dans un dépôt ?

- C'est un DAG (graphe orienté sans cycle)
  - Chaque noeud est un commit (ou un tag)
- Un noeud peut avoir
  - 0 parent : commit initial
  - 1 parent : commit « normal »
  - Plusieurs parents : merge
  - 1 fils normal : commit « normal »
  - Des fils merges quand la branche est mergée dans d'autres



- On ne peut pas enlever un noeud qui a des fils
- Les objets peuvent être stockés tel quels ou *packés*

Y'a quoi dans un commit ?

- Un commit est une vue de l'arbre des fichiers
  - *blobs* + *filenames* + *subdirs* = *tree*
- Un objet commit contient
  - Un message
  - Une référence vers chaque objet commit parent
  - Une référence vers le *tree* correspondant
- Notion de chemin mais pas vraiment de répertoires
  - Pas la peine de faire git add sur un répertoire vide
    - Faire git add sur les fichiers qu'il contient

Ca change quoi cette façon de stocker ?

- On sait *a priori* quels fichiers sont touchés par un commit
  - git show et git diff sont faciles à implémenter

- On ne sait pas *a priori* quels commits touchent un fichier
  - git blame est difficile
    - Parcourir tous les commits
      - Regarder s'ils touchent notre fichier
      - Regarder s'ils sont le dernier à avoir touché chacune des lignes

Et c'est quoi une branche ou un tag ?

- Une branche est une étiquette sur un objet commit
  - git commit ajoute un objet commit
    - puis déplace l'étiquette de la branche courante
  - HEAD est une étiquette sur l'étiquette de la branche courante
  - Une branche distante (origin/master) est une étiquette de couleur différente

- Un tag est objet qui contient un message et qui pointe sur un objet commit
  - Et il ne bouge pas quand on committe par dessus

L'index

L'index

- Intermédiaire entre le checkout et le repository
- On est censé ajouter à l'index avant de committer
  - git commit -a ajouter implicitement tous les changements locaux avant de committer
    - Ca suffit souvent

- Ajouter des modifications (ou un fichier) à l'index
  - git add monfichier
- Supprimer complètement un fichier de l'index
  - git rm monfichier
- Committer ce qui est dans l'index
  - git commit

Consulter l'état courant du ch
(clarification du slide de tout

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README
#
# 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:   configure.ac
#
# Untracked files:
#   (use "git add <file>..." to include in what will
#
#       INSTALL
#       Makefile.in
```

Fichiers committables par **`git commit`**

Fichiers modifiés (après git add)

Fichiers modifiés (sans git add)

Fichiers inconnus par GIT

Fichiers committables par **`git commit -a`**

Consulter l'état courant du checkout (2/2)

- Tous les changements locaux depuis le dernier commit
  - git diff HEAD
- Afficher uniquement ce qui n'est pas encore dans l'index
  - (pas encore ajouté avec git add)
  - git diff
- Afficher uniquement ce qui est dans l'index
  - (déjà ajouté avec git add)
  - git diff --staged

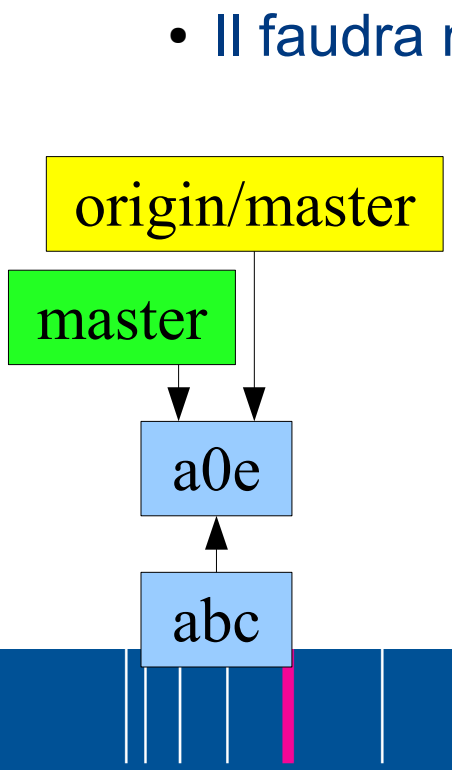
Gérer l'index (clarification du slide de tout à l'heure)

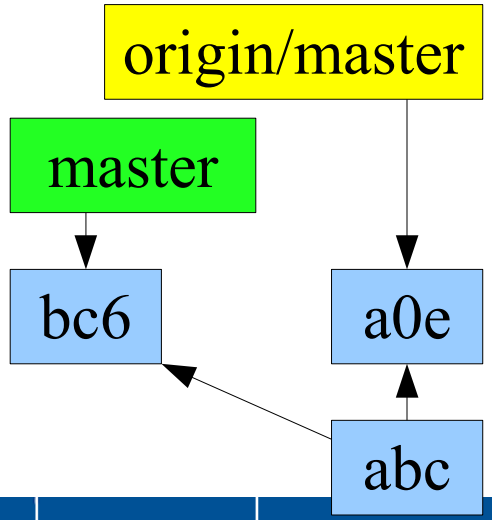
- Vider l'index, sans modifier les fichiers (annuler git add, ...)
  - git reset
- Vider l'index et annuler les modifications des fichiers
  - git reset --hard
- Par extension, revenir à une autre version
  - git reset --hard <identifiant>
- Annuler des commits sans perdre les modifications
  - git reset <identifiant>

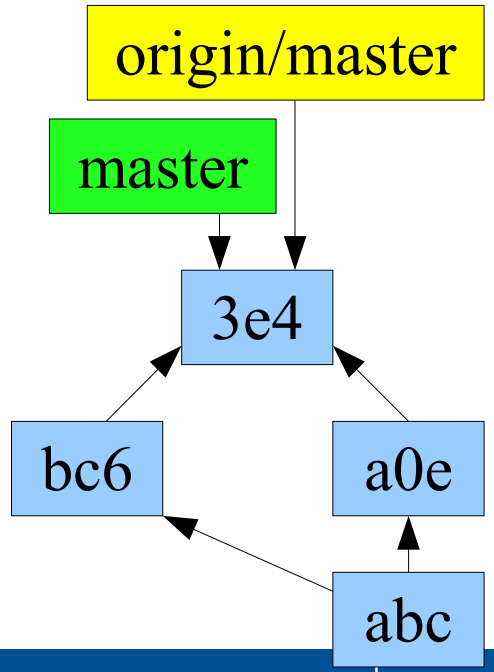
Corriger des commits ?

- Modifier un commit qui n'est pas le dernier ?
  - Voir git rebase

- Si la branche a déjà été publiée, modifier l'historique crée une divergence par rapport à la version publique
  - Il faudra merger plus tard







Opérations avancées

Retrouver des commits

- Les hash, c'est précis mais pas très pratique

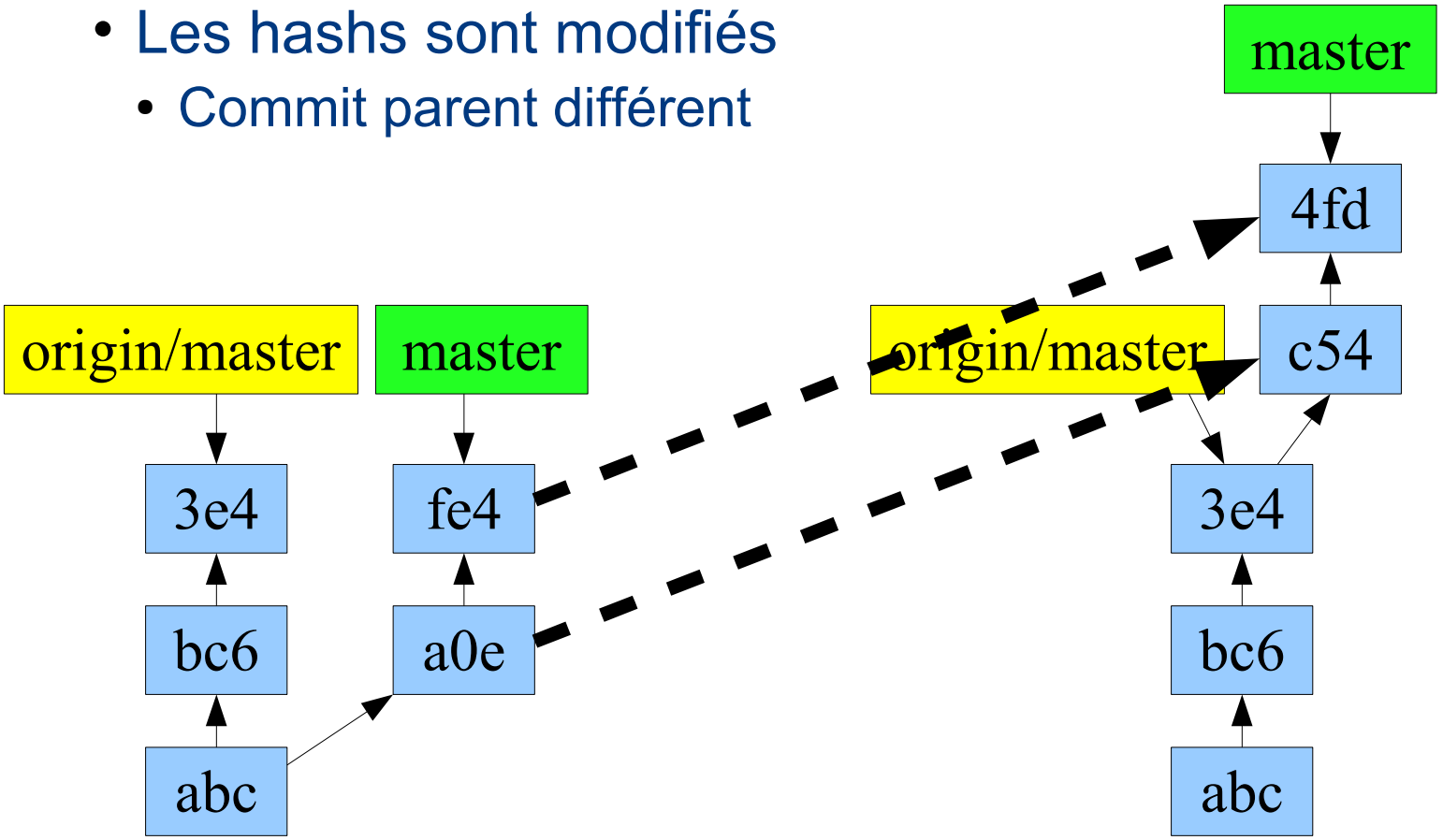
- Certains commits ne sont dans aucune branche **nommée**
  - Peuvent uniquement être retrouvés par leur hash
- Mettre une référence dessus pour simplifier la suite
  - git branch mybranch <hash>
  - git tag mytag <hash>
- Quand un commit est dans une branche nommée, c'est facile de le retrouver...

git rebase – Mettre à jour des commits

- Quand on a des modifications locales et qu'on veut récupérer des modifications distantes
  - Comment les combiner ?
- git merge fusionne deux branches
  - Les deux développements sont considérés comme parallèles
- Et si on réappliquait les commits locaux au dessus des modifications distantes ?
  - Historique propre
    - Pas de branche, pas de merge

git rebase (2/4)

- Les hashs sont modifiés
  - Commit parent différent



git rebase (3/4)

- git regarde la différence entre local et distant
  - Il va réappliquer tous les commits qui manquent
- git met de côté les commits locaux entre temps
  - Puis les réapplique à l'identique ensuite

- A chaque conflit, le rebase est interrompu
  - Les fichiers en conflit doivent être résolus et git add-és
  - git rebase --continue pour reprendre

- Si une modification locale est déjà dans le dépôt distant
  - Il disparaît naturellement lors de son application
- En cas de problème, git rebase --abort

git rebase (4/4)

```
$ git fetch origin
   0fb13ca..19c7684  debian-unstable -> origin/debian-unstable
$ git checkout debian-unstable
Switched to branch 'debian-unstable'
Your branch and 'origin/debian-unstable' have diverged,
and have 1 and 9 different commit(s) each, respectively.

$ git rebase origin
First, rewinding head to replay your work on top of it...
Applying: my local commit
CONFLICT (content): Merge conflict in debian/changelog
Failed to merge in the changes.
Patch failed at 0001 my local commit

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To restore the original branch and stop rebasing run "git rebase --abort".

$ vi debian/changelog
[…]
$ git add debian/changelog
$ git rebase --continue
Applying: my local commit
```

La boîte à outils git rebase

- Le mode interactif de git rebase permet de configurer l'application de nos patchs
  - Les réordonner, découper, fusionner, éditer, supprimer, …
  - Action à préciser sur la ligne de chaque commit dans l'éditeur
- git rebase -i origin
- Par extension, retravailler nos 3 derniers commits
  - git rebase -i HEAD^^^

Divergences avec un dépôt distant

- Par défaut, git push impose un Fast-forward
  - La branche sur le serveur doit être un ancêtre de ce qu'on veut pousser
- Généralement, on pulle/rebase puis on pushe
  - On fait disparaître nos divergences avant de publier
    - Mais pas toujours...

- On peut écraser la branche distante avec git push -f
  - Utile après git rebase ou git reset par exemple
  - Les anciens commits ne sont pas perdus mais la branche distante pointe désormais vers le même commit que notre branche locale

Divergences avec un dépôt distant (2/2)

- Attention aux autres clones !
  - Si on écrase une branche distante, les autres clones peuvent ne plus avoir le même ancêtre de branche
    - ex: Si on a rebasé ou resetté leur ancêtre
    - S'ils sont prévenus, ils peuvent s'en sortir avec fetch + rebase
    - Sinon ils vont merger et avoir les commits dans les 2 branches
      - Ca marche mais l'historique n'est plus très facile à lire...
- Bref, éviter git push -f si d'autres personnes ont cloné...
  - Est-ce que je peux utiliser git rebase ?
    - Pour gérer mes modifications **locales**, oui c'est même le mieux !
  - Sinon prévenir les autres développeurs
  - cf les trolls à ce propos
    - http://kerneltrap.org/Linux/Git_Management
    - http://lwn.net/Articles/291304/

Retour sur git push

- Attention au push dans dépôt initialisé sans --bare
  - La branche qu'on va modifier est peut-être checkoutée
    - Il faudra y faire git reset --hard HEAD pour voir les modifications, alors que git pensera être à jour...
  - C'est pour cela qu'on recommande --bare dans git init
    - Notamment sur gforge

git cherry-pick – Backporter entre branches

- Comment porter vite-fait un commit dans une branche ?
  - git show identifiant > patch ; patch -p1 < patch ; git commit
    - Résolution des conflits, git add/remove, …
- git cherry-pick identifiant
  - Applique le commit proprement
  - Si conflit, il faut résoudre, git add puis git commit

- Permet de backporter rapidement des correctifs de bogues dans une branche stable
  - Sans avoir à faire des branches dédiées
- Pas de trace du cherry-pick dans l'historique
  - En cas de merge plus tard, GIT se débrouille

git bisect – Faire une dichotomie

- Un bug est apparu entre les révisions x et y ? Mais où ?
  - Localiser le commit fautif par dichotomie
    - En gérant les branches/merges
    - En précisant des répertoires et/ou fichiers impliqués

- git bisect start -- monrepertoire monfichier1 monfichier2
- git bisect bad identifiant
- git bisect good identifiant
- git va proposer des révisions intermédiaires à tester
  - Compiler, tester
    - git bisect good ou git bisect bad selon le résultat
- On obtient le commit fautif

Retour sur git reset

- Dans branche1, que fait git reset --hard branche2 ?
  - On reste dans branche1 quoi qu'il arrive
  - On bouge le pointeur branche1 vers le bout de branche2
  - L'ancienne tête de branche1 existe toujours mais n'est plus pointée par branche1
- Que fait git reset branche2 ?
  - Comme ci-dessus, sans modifier les fichiers
  - git commit va créer un commit suivant la tête de branche2 avec le contenu de branche1

Retrouver des commits (2/2)

- Les anciens commits ne sont pas supprimés du dépôt automatiquement
  - Même si git reset, git rebase, git branch -D, ...
- En cas de problème, on peut les retrouver
  - Et faire git reset --hard identifiant pour y retourner
- Comment retrouver leur hash ?
  - git reflog liste les hash successifs de HEAD

- Un commit non référencé n'est pas supprimé du dépôt
  - Il faut nettoyer le dépôt de force pour le supprimer
    - git prune, git fsck, git gc, ...

Fichiers ignorés

- git status affiche plein de *Untracked files*
  - Masquer en mettant dans .gitignore
- Nettoyer les fichiers inutiles
  - git clean -fd voire avec -x

Divers

- git stash
  - Permet de mettre des commits de côté
- git revert
  - Appliquer l'inverse d'un commit
- git format-patch et git am
  - Générer un patch à mailer / Appliquer un patch d'une mailbox
- git prune, git repack, git gc, ...
  - Faire du ménage, optimiser le dépôt, …
- git gui, gitk
  - Interfaces graphiques
- git <tab>
  - Une centaine d'autres commandes plus ou moins obscures :)

GIT SVN

GIT SVN

- GIT du coté client, SVN du côté serveur
  - Permet de profiter de GIT sans l'imposer à tout le monde
- Donne quasiment tous les avantages de GIT
  - Quelques problèmes pour publier des branches entre dépots GIT-SVN
    - Les identifiants GIT de mêmes commits SVN varient parfois pour des raisons obscures

Créer un dépôt GIT à partir d'un SVN

- git svn clone --stdlayout
  - Crée un dépôt GIT en tenant compte de trunk, branches/* et tags/*
    - Les branches et tags ne sont pas importés en tant que répertoires normaux
  - Les branches SVN sont disponibles sous leur nom SVN mais manipulées comme des branches distantes
    - trunk doit être compris comme un origin/trunk

- Toutes les révisions SVN sont rapatriées individuellement
  - Ca peut être trèèèèèèèèèès lent
  - Ajouter -r <revision> pour commencer à une révision précise
    - On perd un bout de l'historique et des branches

Branches SVN et GIT

- La branche git master suit la branche SVN trunk
- On peut suivre d'autres branches SVN
  - git branch mabranchegit --track mabranchesvn
    - Ne pas utiliser le même nom de branche !

```
$ git svn info
URL: svn+ssh://scm.gforge.inria.fr/svn/open-mx/trunk
Repository Root: svn+ssh://scm.gforge.inria.fr/svn/open-mx
Repository UUID: f1ba3bf5-cb5c-402b-92a9-7c6bdc83a356
Revision: 2949
Node Kind: directory
Last Changed Author: bgoglin
Last Changed Rev: 2949
Last Changed Date: 2010-04-01 11:35:21 +0200 (jeu., 01 avril 2010)

$ git svn log
------------------------------------------------------------------------
r2917 | bgoglin | 2010-02-26 11:42:43 +0100 (ven., 26 févr. 2010) | 7 lines
Backport/adaptation of trunk commits r2914-2915
```

Comment faire un svn up ?

- git svn rebase
  - Télécharge les nouveaux commits distants dans la branche SVN correspond à notre branche GIT
  - Rebase nos commits GIT locaux par dessus
  - En cas de conflit, comme git rebase normal
    - Corriger les conflit, git add, git commit, git rebase --continue
- git svn fetch
  - Télécharge les commits de toutes les branches SVN, et les nouveaux tags
    - Visibles dans les branches remote
      - git log trunk
  - Faire un git svn rebase dans chaque branche GIT ensuite

Comment faire un svn commit ?

- Travailler en local comme dans git
  - Tout ce qu'on peut par dessus les branches SVN
  - commit/branch/checkout/merge/rebase/...

- Pousser les derniers changements GIT dans la branche SVN correspondante
  - git svn dcommit
    - Un identifiant git-svn-id est ajouté au commit GIT

```
commit 56f7555f7ba3d7f6e59f8b838eecb91093ea2146
Author: bgoglin <bgoglin@f1ba3bf5-cb5c-402b-92a9-7c6bdc83a356>
Date:   Thu Apr 1 09:35:21 2010 +0000

    Tiny fix in README.devel

    git-svn-id: svn+ssh://scm.gforge.inria.fr/svn/open-mx/trunk@2949
      f1ba3bf5-cb5c-402b-92a9-7c6bdc83a356
```

Merges dans GIT SVN

- Un vrai merge GIT est propagé comme un commit SVN contenant les commits de l'autre branche GIT
  - L'historique GIT du committeur est plus détaillé que l'historique SVN
- Mais un merge SVN est importé comme un commit GIT normal
  - Pas de branche GIT
    - Historiques GIT et SVN linéaires, exactement identiques
- Divergence des dépots GIT selon si les merges ont été faits en local (GIT pur) ou à distance (linéarisé par SVN)

Remarques sur GIT SVN

- git svn clone/fetch/rebase/dcommit ne sont pas des opérations locales
  - Le reste est du git normal ou du git svn local
- Les externals SVN ne sont pas supportés
- Eviter de modifier l'historique après dcommit
  - De la même façon qu'on évite de modifier en local après push
    - git reset, git rebase, … uniquement sur les commits GIT locaux
  - Sinon GIT SVN va reprendre tous les commits SVN puis réappliquer les commits modifiés localement
    - Conflits