

Git Tutorial Tour

Jon Loeliger



Git Tutorial Tour Outline

- Repository Management
 - Git Repository Concepts
 - Inter-Repository Data Flow
 - Branches
 - Merges
- Local Use Cases
 - Pull Updates
 - Generating Patches
 - Examples
- Supplemental Slides

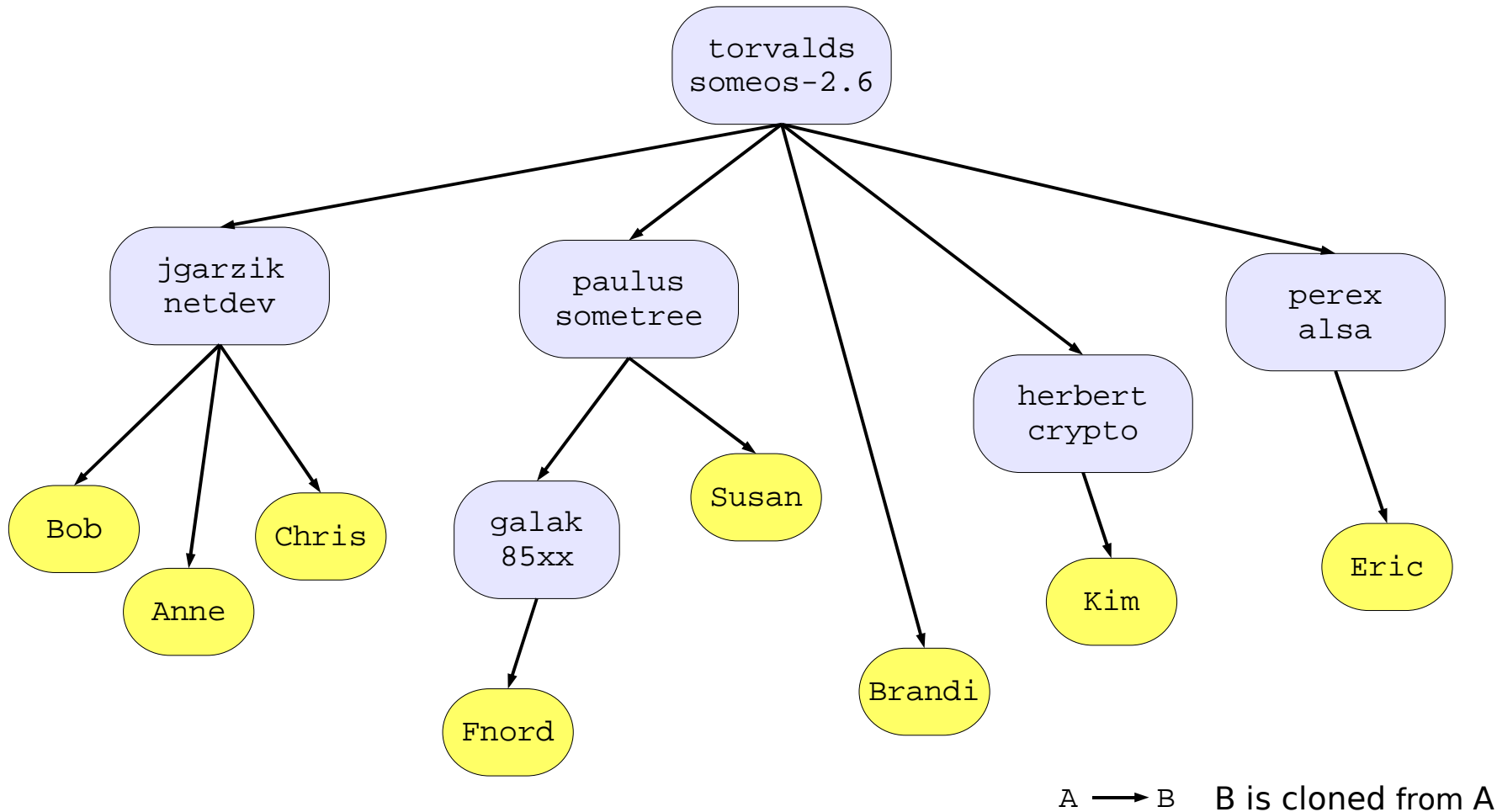
Git Concepts: Repositories

- Git repository is self-contained, local and complete
 - Doesn't require external content
 - No centralized repository
 - Includes complete history of every file and change log
- Git repository is not distributed
 - No: part here, part there, part over there...
- No git repository is inherently authoritative
 - Only authoritative by convention and agreement

Git Concepts: Distributed Development

- Development can be distributed
 - Multiple repositories can create different histories and changes
 - ◆ Even if they originated from a common ancestor
 - Multiple repositories with different development can be combined
- Development can be shared
 - Multiple developers can use and update a common repository
 - ◆ No matter if the repository is local or remote
- Development can be private
 - Revision control your Address List at home

Repository Network



Git Concepts: Branches in git

- Cheap, fast and easy
- Topic/Development Branches
 - Stable, development, bug-fix, testing branches
 - Small development lines, per-feature, per-developer, etc
 - Collect, reorder or organize changes
 - Cherry-pick particular patches
- Tracking Branches
 - Follow upstream changes in local repository
 - Don't commit to tracking branches
 - Identified as RHS of `PULL : refspecs`
- Branch names refer to the current branch HEAD revision
- Branches don't have a “beginning” per se
 - \$ `git merge-base <original-branch> <branch-name>`

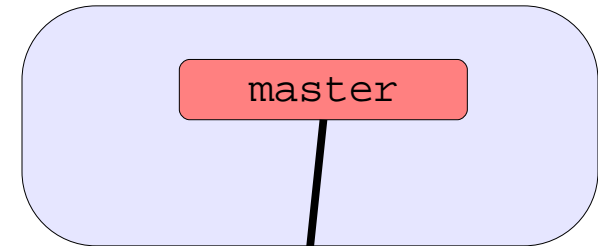
Branches: Cloning

- Clone from upstream URL

```
$ git clone  
git://www.jdl.com/software/dtc.git
```

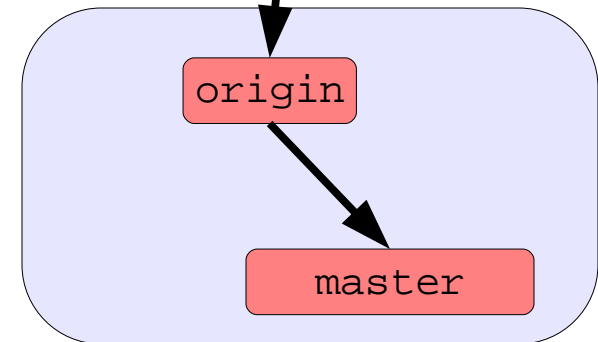
- Copies complete upstream repository into a local repository
- Creates `origin` tracking branch
 - Tracks upstream `master` branch
- Introduces `master` development branch
 - Initially the same place as `origin`
 - For *your* development

Original Repository



git clone

Cloned Repository



Branches: Commit Onto Branches

- Master branch in your repo is for *your* development!

```
$ git checkout master
# Edit a file.
$ git commit -a -m "Add copyright.  Fix 80-column line."
```

- Introduce your own topic development branches too

```
$ git checkout -b jdl
# Edit a file.
$ git commit -a -m "Remove dead code."
```

- Gotcha: Commits require an author!

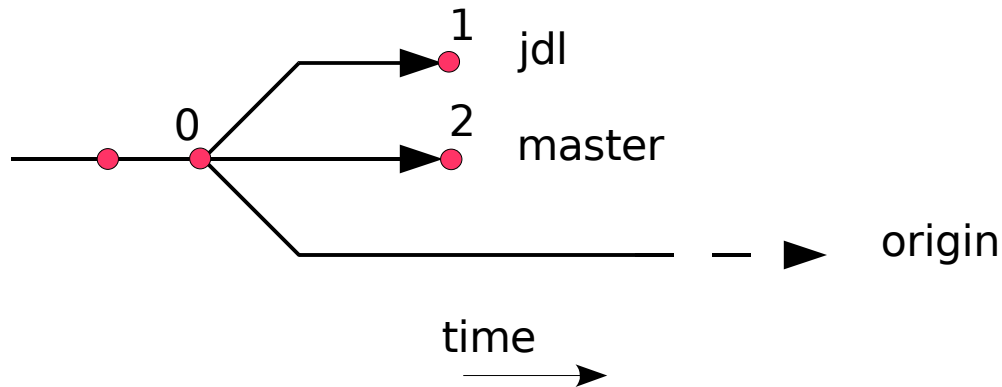
- Environment variables: `GIT_AUTHOR_NAME`, `GIT_AUTHOR_EMAIL`
- Config file: `$ git repo-config user.name 'Eric S. Raymond'`
- Gecos value: `/etc/passwd`

Branches: Visualizing Branches

```
$ git show-branch
```

```
! [jdl] Remove dead code.  
* [master] Add copyright. Fix 80-column line.  
! [origin] dtc: add setting of physical boot cpu  
---  
* [master] Add copyright. Fix 80-column line. ← 2  
+ [jdl] Remove dead code. ← 1  
+*+ [origin] dtc: add setting of physical boot cpu ← 0
```

Branch Timeline



Git Concepts: Merges

- Merge: Combine directory and file contents from separate sources to yield one combined result.
 - Sources for merges are local branches
 - Merges always occur in the current, checked-out branch
 - A complete merge ends with a new commit
- Merge resolution is inherently ambiguous
 - Git uses several merge heuristics:
 - ◆ Several merge strategies: resolve, recursive, octopus, ours
 - ◆ Techniques: fast-forward, three-way
- Some merge conflicts may need to be resolved by the developer

Merge jdl branch To master

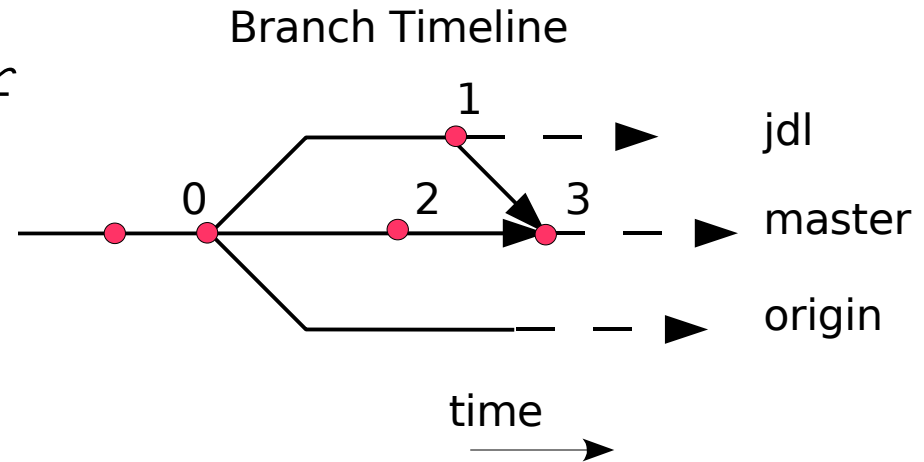
- Merge jdl branch into master

```
$ git checkout master  
$ git pull . jdl
```

- Resulting Branch History

```
$ git show-branch
```

```
! [jdl] Remove dead code.  
* [master] Merge branch 'jdl'  
! [origin] dtc: add setting of physical boot cpu  
---  
- [master] Merge branch 'jdl'  
+* [jdl] Remove dead code.  
* [master^] Add copyright. Fix 80-column line.  
+*+ [origin] dtc: add setting of physical boot cpu
```



Outline: Use Cases

- Repository Management
- Local Use Cases
 - Pulling Updates
 - Making Patches
 - Finding a Commit
 - Cherry Picking
 - Reverting a Commit
 - Resolving Merges
 - Rebasing Local Changes
- Supplemental Slides

Pulling Updates

- The default origin “remotes” file

```
$ cat .git/remotes/origin
URL: git://www.jdl.com/software/dtc.git
Pull: refs/heads/master:refs/heads/origin
```

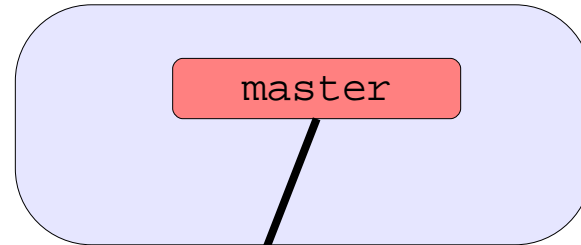
- Get updates from remote repository

```
$ git checkout master
$ git pull origin
```

- Pull is a fetch then merge

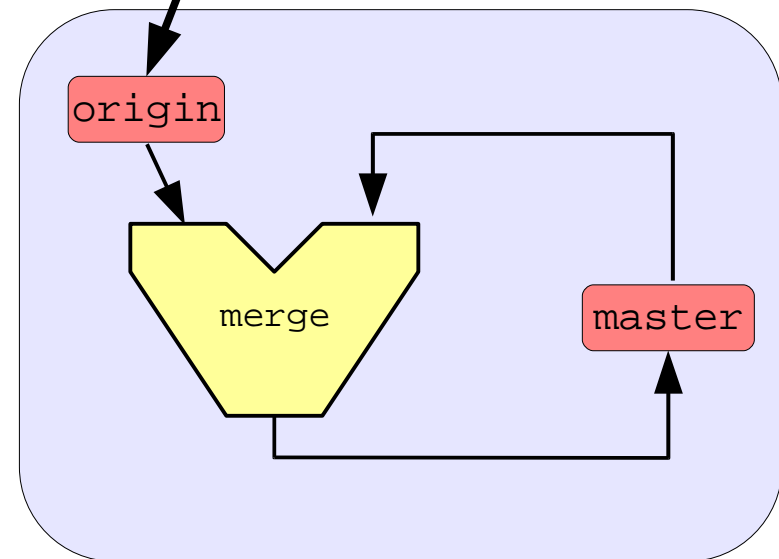
- Places fetched updates in `ref/heads/origin`
- Merges `origin` into the current, checked-out branch, `master`

Original Repository



`git pull`

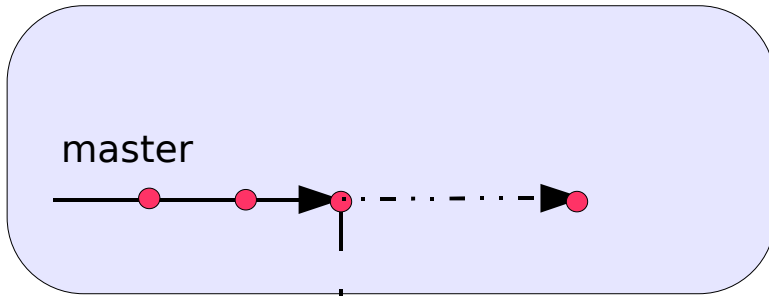
Cloned Repository



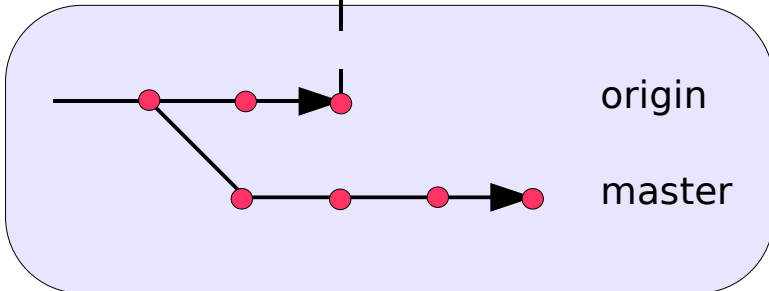
Pull: Fetch and Merge Branch Timeline

Before

Public Repository



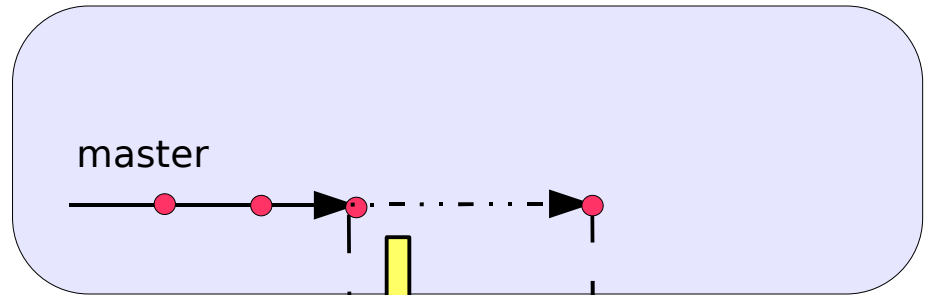
Your Repository



time →

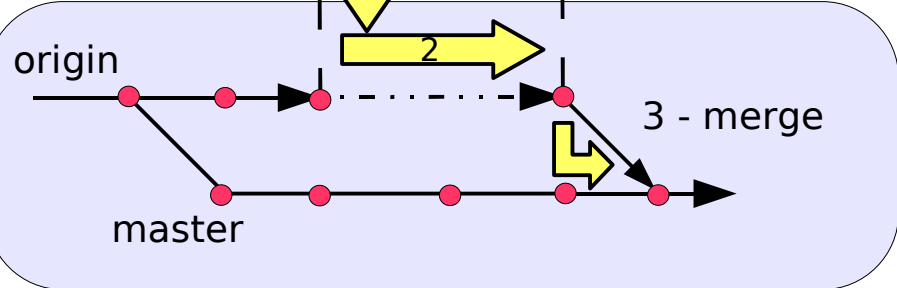
After

Public Repository



1- fetch

2- fast-forward



Your Repository

time →

Pull: What Else?

- Pull is a `fetch` followed by a `merge`
 - Merges should start with some common base, though
- Can fetch/pull any branch from any repository
 - Supply URL on command line directly
 - Maybe add new “remotes” file with URL and `PULL:` line
 - Even this repository: `git pull . <branch>`
- Could just `git fetch` an upstream branch too
 - Able to then `cherry-pick` commits rather than full merge
 - Able to use `git diff`, `git log`, etc.

Pull: When?

- When you want to!
- When it is convenient, stable, Wednesday...
- When the upstream tree is stable
- When your working directory is stable
 - ...and you have the right branch checked out
 - Your working directory should probably be clean
 - ♦ `git diff` or `git status` should be empty
 - Technically not necessary, but...

Sending Changes Upstream

- Generate and send patches via email
 - Most developers send patches to a maintainer or list
 - Highly visible public review of patches on mail list
- Maintainer pulls updates from a downstream developer
 - Maintainer can directly pull from your published repository
 - Initiated by upstream maintainer
- Developer pushes updates to an upstream maintainer
 - Some developers have write permissions on an upstream repository
 - Initiated by downstream developer

Patches: Generate a Patch

- Use `git format-patch` command

```
$ git format-patch --signoff origin..jdl
```

```
From: Jon Loeliger <jdl@jdl.com>  
Date: Sat, 24 Jun 2006 15:42:51 -0500  
Subject: [PATCH] Remove dead code.
```

```
Signed-off-by: Jon Loeliger <jdl@jdl.com>
```

```
---
```

```
data.c | 18 -----  
1 files changed, 0 insertions(+), 18 deletions(-)
```

```
diff --git a/data.c b/data.c  
index 911b271..d3b55f9 100644  
--- a/data.c  
+++ b/data.c  
@@ -20,24 +20,6 @@
```

Patches: A Side Note

- Read the README
- Follow the coding standards
- Compile and test your code
- Fix the whitespace issues
- Know who the upstream maintainer is
- Understand any “sign-off” policy

Patches: Formatting and Sending Mail

- Send plain ASCII, not HTML
 - Send patches inline
 - Don't use attachments
- Send only your changes
- One line “commit summary” first
- Be careful of cut-n-paste solutions
 - Use file browse and insert if possible
- Maybe move pre-formatted patches to your MUA draft folder
- Send git-format-patch output directly

```
$ git send-email --to  
maintainer@example.com my.patch
```

Finding a Commit

- SHA-1 hash names are constant
- Symbolic commit names like `master~4` change over time!
 - They are relative to the current HEAD
- Ways to determine the name of a commit you want
 - `git show-branch => paul~38^2~10^2`
 - Paw around in `gitk => 9ad494f62444ee37209e85173377c67612e66ef1`
 - Use `git log -- <some/file/path>`
 - Use `gitweb`
- Note:

```
$ git rev-parse paul~38^2~10^2
9ad494f62444ee37209e85173377c67612e66ef1
```

Cherry Picking Example

- You want one or more particular commits from some other branch applied to your current branch
 - Bug fix from some other branch
 - Transfer partial functionality from development branch
- Get that commit into your repository
 - Already present on a different branch
 - Perhaps using `git fetch` from another repository
- Cherry-pick it into appropriate branch

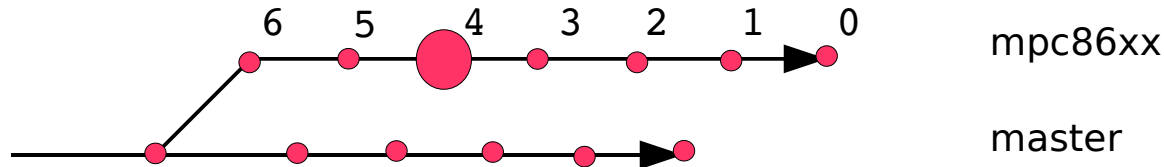
```
$ git checkout master
$ git cherry-pick 9ad494
```

Cherry Picking: Before and After

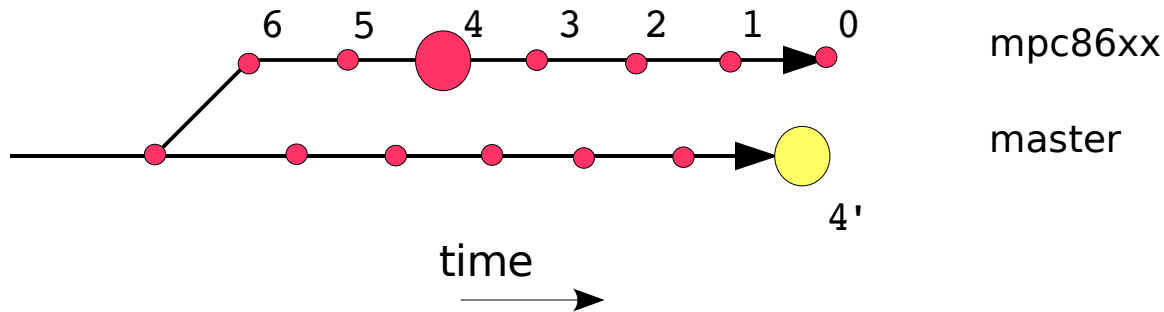
- Branch Timeline Picture

- 4' is a different commit with the same content as 4

Before



After



Revert Example

- Situation:
 - Some commit buried in the past needs to be undone
 - ... and it has already been merged to master too!

```
$ git show-branch --more=10 master mpc86xx
```

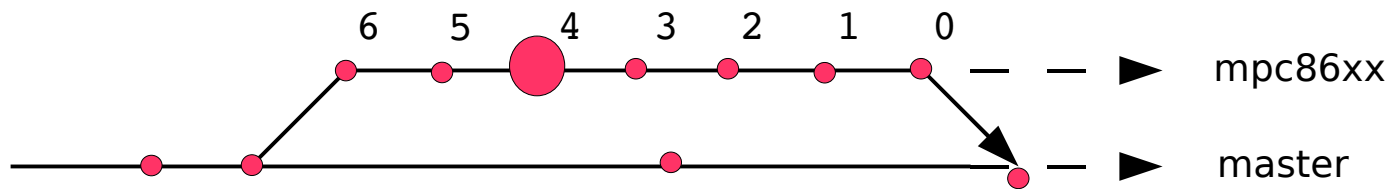
```
* [master] Merge branch 'mpc86xx'  
! [mpc86xx] Remove obsolete #include <someos/config.h>.  
--  
- [master] Merge branch 'mpc86xx'  
*+ [mpc86xx] Remove obsolete #include <someos/config.h>.  
*+ [mpc86xx^] Remove redundant STD_MMU selection.  
*+ [mpc86xx~2] Move I8259 selection under MPC8641HPCN board.  
*+ [mpc86xx~3] Remove redundant PPC_86XX check.  
*+ [mpc86xx~4] Reworked the IRQ mapping. Now reading IRQ and ... <===== ARGH!  
*+ [mpc86xx~5] Make 86xx secondary CPU start be more generic.  
*+ [mpc86xx~6] Updated 86xx defcnofig after merge with Paul
```

- This is a job for `git revert`

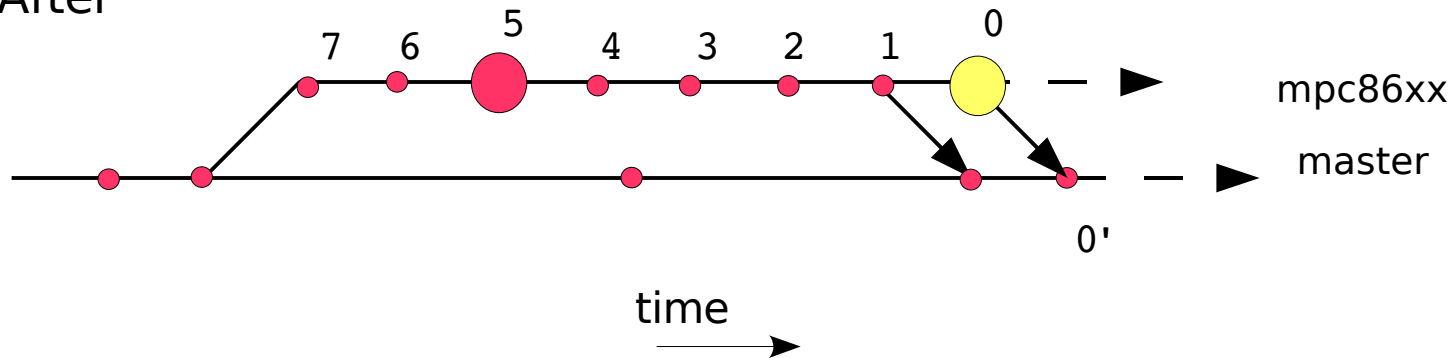
Revert: Before and After

- Commit 4 is reverted, creating commit 0 in yellow
- Commit 0 is merged to master as commit 0'

Before



After



Revert: Commands

- Revert applies a new “inverse” patch for a given commit
- Revert works without rewriting the commit history
- Can revert multiple commits at the same time

- Revert commit on `mpc86xx` topic branch:

```
$ git checkout mpc86xx  
$ git revert mpc86xx~4
```

- Reflect it onto `master` branch as well:

```
$ git checkout master  
$ git pull . mpc86xx
```

Revert: Branch History After

```
$ git show-branch --more=10 master mpc86xx

* [master] Merge branch 'mpc86xx'
! [mpc86xx] Revert "Reworked the IRQ mapping. Now reading IRQ and ..."
--
- [master] Merge branch 'mpc86xx'
*+ [mpc86xx] Revert "Reworked the IRQ mapping. Now reading IRQ and ..."
*+ [mpc86xx^] Remove obsolete #include <someos/config.h>.
*+ [mpc86xx~2] Remove redundant STD_MMU selection.
*+ [mpc86xx~3] Move I8259 selection under MPC8641HPCN board.
*+ [mpc86xx~4] Remove redundant PPC_86XX check.
*+ [mpc86xx~5] Reworked the IRQ mapping. Now reading IRQ and ...
*+ [mpc86xx~6] Make 86xx secondary CPU start be more generic.
*+ [mpc86xx~7] Updated 86xx defcnofig after merge with Paul
```

Revert versus Reset versus Checkout

- Reset versus Revert: Trying to undo the last commit?
 - Ask: Does someone else have this version of the repository history?
 - Yes: Use `git revert` and do not change history
 - No : Could use `git reset` or `git commit --amend` perhaps
- Reset versus Checkout
 - Reset doesn't change your current branch
 - Checkout establishes your current branch
 - These can be similar unless naming a different branch:

```
git reset --hard topic    # still on original branch,  
                          #   but it now looks like topic!  
git checkout -f topic    # topic branch checked out now
```

Merge Resolution Hell ... er, Examples

- Pulling updates from upstream can go wrong for a number of reasons!
 - Conflicting file contents
 - New or modified files in your working tree
 - Different file modes
 - Possible for your upstream changes to come back differently
 - ◆ Altered upstream by maintainer?
 - ◆ Were they pulled or patched upstream?
 - ◆ Same file *content*, but different *commits*?
 - Merge is a heuristic!

Merge Resolution Strategies

- Before doing the pull or merge

- Check out the correct “merge into” branch, likely `master`
- Ensure a clean working directory first

```
git ls-files --others
git status
```

- After “failed” pull/merge request:

```
git ls-files -u    # Show unmerged files remaining
git diff          # Clean if it matches one of the variants!
```

- Make progress resolving conflicts:

```
git update-index # Tell git when a conflict has been resolved
```

- Be done :

- Fully resolved: `git commit`
- Abandon merge: `git reset --hard ORIG_HEAD`

Failed Merge: Untracked Working Tree File

- Git says:

```
git-read-tree: fatal: Untracked working tree file  
'Documentation/ABI/README' would be overwritten by merge.
```

- Huh? I don't have that file...

- Maybe you *do* have that file and should check!
- But maybe it is leftover from a previous merge effort?
- Are you on the right branch?

```
git branch
```

```
git show-branch
```

Failed Merge: Untracked File Resolution

- Your key:

```
git ls-files --others    # Look for untracked files
git status               # Check for unexpected files
git clean -d            # Remove cruft
```

- Reset the working directory:

- Maybe abandon this merge?
- Look for other files
- Clean out the old files
- Do the merge again!

```
$ git reset --hard ORIG_HEAD
$ git ls-files --others
$ git clean -d
$ git status    # Ah! Nice-n-tidy
```

Failed Merge: CONFLICT Content

- Git message:

```
CONFLICT (content): Merge conflict in drivers/net/phy/Makefile
```

- Semi-traditional "<<<< ==== >>>>" style file content differences.

```
$ git diff drivers/net/phy/Makefile
```

```
@@@ -8,4 -8,5 +8,9 @@@
```

```
obj-$(CONFIG_CICADA_PHY)      += cicada.o
```

```
obj-$(CONFIG_LXT_PHY)        += lxt.o
```

```
obj-$(CONFIG_QSEMI_PHY)      += qsemi.o
```

```
++<<<<<< HEAD/drivers/net/phy/Makefile
```

```
+obj-$(CONFIG_VITESSE_PHY)    += vitesse.o
```

```
++=====
```

```
+ obj-$(CONFIG_SMSC_PHY)      += smsc.o
```

```
+ obj-$(CONFIG_VITESSE_PHY)   += vitesse.o
```

```
++>>>>>>
```

```
501b7c77de3e90519e95fd99e923bf9a29cd120d/drivers/net/phy/Makefile
```

Failed Merge: Resolve Content Conflict

- Your key:

```
git diff
```

```
git ls-files -s arch/somearch/Kconfig # Various stage versions
```

- Edit and fix

- Use your favorite editor, emacs

- Make progress

```
git update-index arch/somearch/Kconfig
```

Failed Merge: CONFLICT add/add

- Git says either:

- Added file3 in both, but differently. `ERROR: Merge conflict in file3`

- `CONFLICT (add/add):`

- File `arch/somearch/platforms/86xx/mpc86xx_hpcn.c`
added non-identically in both branches.

- Adding as `arch/somearch/platforms/86xx/mpc86xx_hpcn.c~HEAD`
and `arch/somearch/platforms/86xx/mpc86xx_hpcn.c~501b7c` instead.

- Two branches added the same file, but differently.

- The version from my `master HEAD` is left with `~HEAD` suffix.

- The version from incoming origin is left with `~501b7c` suffix.

- Resolution:

- Pick the right one or merge them

- Use `git add` to add it back with the right name (drop `~suffix`)

- Remove the other

Failed Merge: CONFLICT rename/add

- Git says:

```
CONFLICT (rename/add): Rename
arch/{mips/configs/ddb5476_defconfig =>
  somearch/configs/mpc8641_hpcn_defconfig} in 501b7c...
arch/somearch/configs/mpc8641_hpcn_defconfig added in HEAD
```

- Maybe it lies!

- Rename detection isn't 100% precise. It is a heuristic!
 - ddb5476_defconfig just happens to be removed
 - mpc8641_hpcn_defconfig really added!

- Resolution:

- Add it back! Holy cow!
- Make sure index has correct, added file and either:

```
git update-index arch/somearch/configs/mpc8641_hpcn_defconfig
git add arch/somearch/configs/mpc8641_hpcn_defconfig
```

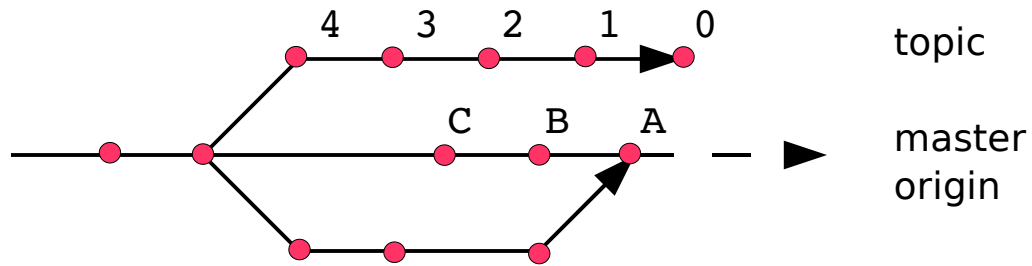
Rebase Local Changes Example

- Situation:
 - You have done development work on your topic branch
 - You pull in upstream `origin` and merge it to `master`
 - You don't want to merge `master` into your topic branch
 - You don't want to merge your topic into the `master` branch
 - No one else has cloned your topic branch!
 - ◆ Why? Don't rewrite history
 - But you want to send your topic branch patches upstream!
- This is a job for `git rebase`

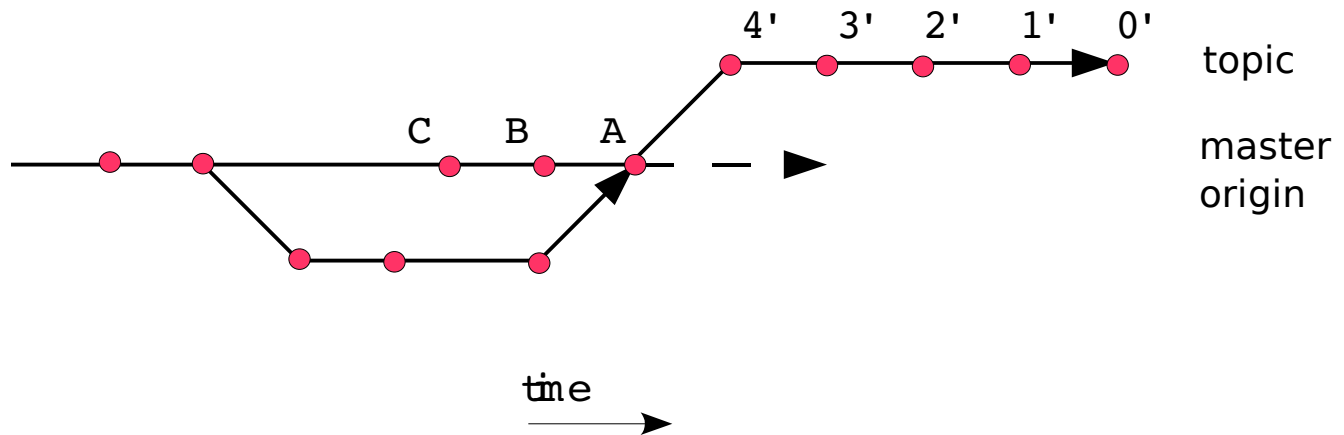
Rebase: Before and After

- Rebase commits 0 – 4 of `topic` onto `master` branch at A as 0' – 4'

Before



After



Rebase: Commands

- Rebase Commands

```
$ git checkout topic  
$ git rebase master
```

- Creates a series of patches from `topic` to be applied to `master`

- May have to resolve conflicts at each step due to merge operation

```
git rebase --continue  
git rebase --skip  
git rebase --abort
```

Git Resources

- Sources and Documentation:
 - Git sources, documentation and many repositories: kernel.org/git
 - The Git Wiki: git.or.cz/gitwiki
- Front-ends and Viewers:
 - Cogito: kernel.org/pub/software/scm/cogito
 - Stacked Git: www.procode.org/stgit
 - Patchy Git: www.spearce.org/category/projects/scm/pg
 - QT Gui viewer: sourceforge.net/project/qgit
- Mail List: [git @ vger.kernel.org](mailto:git@vger.kernel.org)
- IRC: [#git](http://freenode.net)

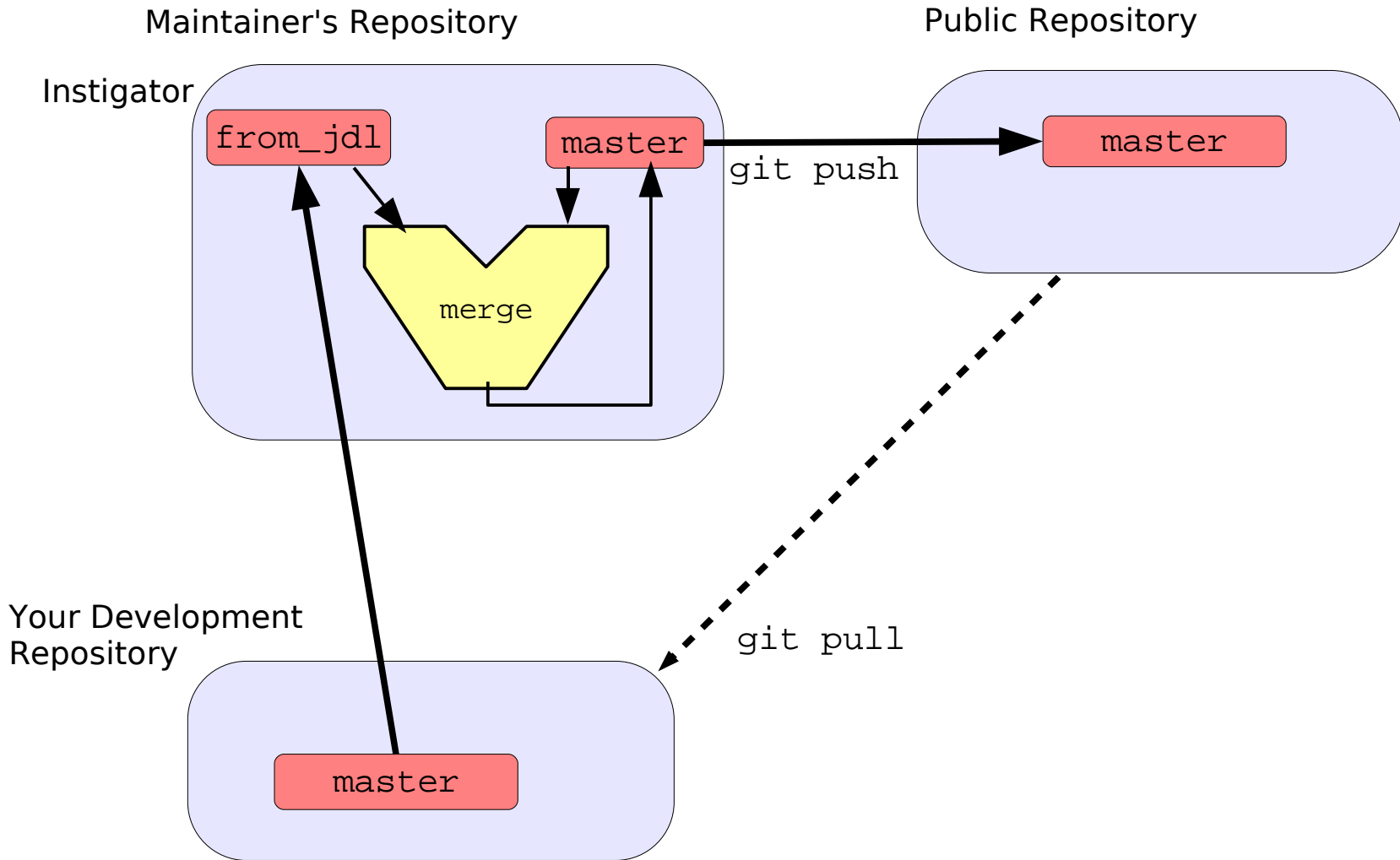
Outline: Supplemental

- Repository Management
- Local Use Cases
- Supplemental Slides
 - Sending Changes Upstream Using Pull and Push
 - Screen Capture of gitk

Send Changes Upstream Using git pull

- Upstream maintainer trusts you and your work
- You have the ability to publish a public repository
- You have a lot of changes, more than a few patches
- Advertise your repository and branch
- Wait for upstream maintainer to pull it
 - `$ git pull git://www.your-site.org/path/to/repo.git`
- Profit.

Using Pull: Data Flow Picture



Sending Changes Upstream Using git push

- Direct push into remote repository
 - Sends objects, packs, branches to remote repository
- Push to a repository from which you have fetched
 - Can only push to a branch that is a proper subset
 - Should result in a "fast-forward" on the remote end
 - Technically you can push elsewhere, but...
- Requires write access on the remote end
 - Likely via ssh

Using Push: Setup Remotes File

- Create a “remotes” file
 - State upstream repository URL
 - State branches to be pushed upstream
- Can push multiple branches, as needed
 - Just add a `Push: refspec` line for each branch
- Can push to different remote branch names

```
Push: master:incoming
```

```
$ cat .git/remotes/publish
URL: ssh://www.jdl.com/software/dtc.git
Push: master:master
Push: jdl:jdl
```

Using Push: git push

```
$ git push publish
```

```
Password:
```

```
updating 'refs/heads/master'
```

```
  from 38e8f8fd88dae07ef8ada9d6baa41b06a4d9ac9f
```

```
  to   a73b7d43d4f60e76d82018fb9a4d137b089a1325
```

```
Generating pack...
```

```
Done counting 11 objects.
```

```
Result has 8 objects.
```

```
Deltifying 8 objects.
```

```
 100% (8/8) done
```

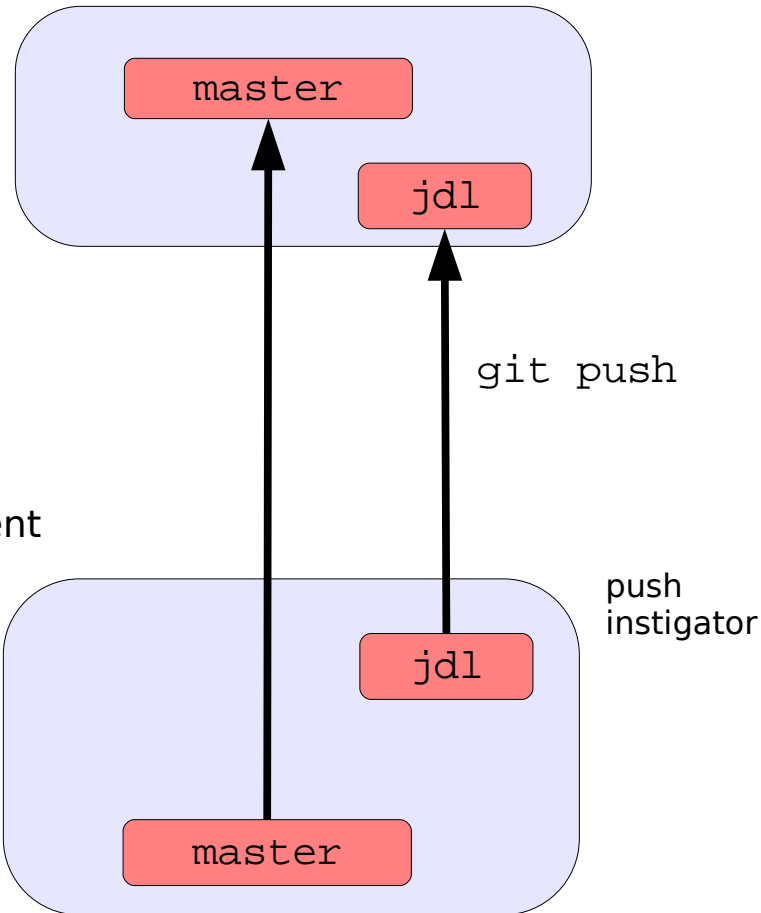
```
Total 8, written 8 (delta 5), reused 0 (delta 0)
```

```
Unpacking 8 objects
```

```
refs/heads/master: 38e8f8 -> a73b7d
```

Using Push: Data Flow Picture

Public
Repository



Your Development
Repository

gitk Example

The screenshot shows the gitk graphical user interface. At the top, there's a menu bar with 'File', 'Edit', 'View', and 'Help'. Below the menu bar is a commit history graph on the left, showing a 'master' branch and a 'pu' branch. The graph is color-coded and shows the flow of commits. To the right of the graph is a list of commits, each with a date and time. The selected commit is highlighted in green. Below the graph and list is a search bar with the SHA1 ID '5faf64cd28bf79f0c2939717d2ba117498717059' and a 'Find' button. Below the search bar is a 'Highlight: Commits' section with a 'touching paths:' dropdown. Below that is a 'Patch' section with a 'Tree' button. The main area shows the commit details for the selected commit, including the author, committer, parent, child, branch, and follows. The commit message is 'Remove awkward compatibility warts' and it is signed-off by Timo Hirvonen and Junio C Hamano. The commit details are shown in a patch view.

SHA1 ID:

Highlight: Commits

◆ Patch ◇ Tree

Comments
builtin-diff-files.c
builtin-diff.c

```
Author: Timo Hirvonen <tihirvon@gmail.com> 2006-06-24 12:28:42
Committer: Junio C Hamano <junkio@cox.net> 2006-07-02 00:26:00
Parent: d7de00f7e0d2374cb7933d2ee1ebe5273a8acf53 (--name-only, --name-status, --
Child: d16867cc9a5bc8f62680c794b58608953b60edbf (Merge branches 'jc/upload-pack
Branch: pu
Follows: v1.4.1-rc1
Precedes:

Remove awkward compatibility warts

Signed-off-by: Timo Hirvonen <tihirvon@gmail.com>
Signed-off-by: Junio C Hamano <junkio@cox.net>

----- builtin-diff-files.c -----
index a655eea..3361898 100644
@@ -47,12 +47,5 @@ int cmd_diff_files(int argc, const char
     if (rev.pending.nr ||
         rev.min_age != -1 || rev.max_age != -1)
         usage(diff_files usage);
```