Distributed Version Control Systems

Matthieu Moy

Computer Science and Automation
Indian Institute of Science
Bangalore

October 2006

Outline

Outline

Backups: The Old Good Time

- Basic problems:
  - "Oh, my disk crashed." / "Someone has stolen my laptop!"
  - "@#%!!, I've just deleted this important file!"
  - "Oops, I introduced a bug a long time ago in my code, how can I see how it was before?"

Backups: The Old Good Time

- Basic problems:
  - "Oh, my disk crashed." / "Someone has stolen my laptop!"
  - "@#%!!, I've just deleted this important file!"
  - "Oops, I introduced a bug a long time ago in my code, how can I see how it was before?"
- Historical solutions:

Backups: The Old Good Time

- Basic problems:
  - “Oh, my disk crashed.” / “Someone has stolen my laptop!”
  - “@#%!!, I’ve just deleted this important file!”
  - “Oops, I introduced a bug a long time ago in my code, how can I see how it was before?”
- Historical solutions:
  - Replicate:

```
$ cp -r ~/project/ ~/backup/
```

Backups: The Old Good Time

- Basic problems:
  - "Oh, my disk crashed." / "Someone has stolen my laptop!"
  - "@#%!!, I've just deleted this important file!"
  - "Oops, I introduced a bug a long time ago in my code, how can I see how it was before?"
- Historical solutions:
  - Replicate:
    ```
    $ cp -r ~/project/ ~/backup/
    ```
  - Keep history:
    ```
    $ cp -r ~/project/ ~/backup/project-2006-10-4
    ```

Backups: The Old Good Time

- Basic problems:
  - "Oh, my disk crashed." / "Someone has stolen my laptop!"
  - "@#%!!, I've just deleted this important file!"
  - "Oops, I introduced a bug a long time ago in my code, how can I see how it was before?"
- Historical solutions:
  - Replicate:
    ```
    $ cp -r ~/project/ ~/backup/
    ```
  - Keep history:
    ```
    $ cp -r ~/project/ ~/backup/project-2006-10-4
    ```
  - Keep a description of history:
    ```
    $ echo "Description of current state" > \
        ~/backup/project-2006-10-4/README.txt
    ```

Backups: Improved Solutions

- Replicate over multiple machines
- Incremental backups: Store only the changes compared to previous revision
  - With file granularity
  - With finer-grained (diff)
- Many tools available:
  - Standalone tools: `rsync`, `rdiff-backup`, ...
  - Versionned filesystems: VMS, Windows 2003+, cvsfs, ...

Collaborative Development: The Old Good Time

- Basic problems: Several persons working on the same set of files
  1. “Hey, you’ve modified the same file as me, how do we merge?”,
  2. “Your modifications are broken, your code doesn’t even compile. Fix your changes before sending it to me!”,
  3. “Your bug fix here seems interesting, but I don’t want your other changes”.

Collaborative Development: The Old Good Time

- Basic problems: Several persons working on the same set of files
    1. "Hey, you've modified the same file as me, how do we merge?",
    2. "Your modifications are broken, your code doesn't even compile. Fix your changes before sending it to me!",
    3. "Your bug fix here seems interesting, but I don't want your other changes".
- Historical solutions:

Collaborative Development: The Old Good Time

- Basic problems: Several persons working on the same set of files
  1. "Hey, you've modified the same file as me, how do we merge?",
  2. "Your modifications are broken, your code doesn't even compile. Fix your changes before sending it to me!",
  3. "Your bug fix here seems interesting, but I don't want your other changes".
- Historical solutions:
  - Never two person work at the same time. When one person stops working, (s)he sends his/her work to the others.
    $\Rightarrow$ Doesn't scale up! Unsafe.

Collaborative Development: The Old Good Time

- Basic problems: Several persons working on the same set of files
  1. "Hey, you've modified the same file as me, how do we merge?",
  2. "Your modifications are broken, your code doesn't even compile. Fix your changes before sending it to me!",
  3. "Your bug fix here seems interesting, but I don't want your other changes".
- Historical solutions:
  - Never two person work at the same time. When one person stops working, (s)he sends his/her work to the others.
    $\Rightarrow$ Doesn't scale up! Unsafe.
  - People work on the same directory (same machine, NFS, ...)
    $\Rightarrow$ Painful because of (2) above.

Collaborative Development: The Old Good Time

- Basic problems: Several persons working on the same set of files
  1. "Hey, you've modified the same file as me, how do we merge?",
  2. "Your modifications are broken, your code doesn't even compile. Fix your changes before sending it to me!",
  3. "Your bug fix here seems interesting, but I don't want your other changes".
- Historical solutions:
  - Never two person work at the same time. When one person stops working, (s)he sends his/her work to the others.
    $\Rightarrow$ Doesn't scale up! Unsafe.
  - People work on the same directory (same machine, NFS, . . .)
    $\Rightarrow$ Painful because of (2) above.
  - People lock the file when working on it.
    $\Rightarrow$ Doesn't scale up!

Collaborative Development: The Old Good Time

- Basic problems: Several persons working on the same set of files
  1. "Hey, you've modified the same file as me, how do we merge?",
  2. "Your modifications are broken, your code doesn't even compile. Fix your changes before sending it to me!",
  3. "Your bug fix here seems interesting, but I don't want your other changes".
- Historical solutions:
  - Never two person work at the same time. When one person stops working, (s)he sends his/her work to the others.
    ⇒ Doesn't scale up! Unsafe.
  - People work on the same directory (same machine, NFS, . . .)
    ⇒ Painful because of (2) above.
  - People lock the file when working on it.
    ⇒ Doesn't scale up!
  - People work trying to avoid conflicts, and merge later.

Merging: Problem and Solution

- My version

```
#include <stdio.h>

int main () {
  printf("Hello");

  return EXIT_SUCCESS;
}
```

- Your version

```
#include <stdio.h>

int main () {
  printf("Hello!\n");

  return 0;
}
```

Merging: Problem and Solution

- My version

```
#include <stdio.h>

int main () {
  printf("Hello");

  return EXIT_SUCCESS;
}
```

- Your version

```
#include <stdio.h>

int main () {
  printf("Hello!\n");

  return 0;
}
```

- Common ancestor

```
#include <stdio.h>

int main () {
  printf("Hello");

  return 0;
}
```

Merging: Problem and Solution

- My version

```
#include <stdio.h>

int main () {
  printf("Hello");

  return EXIT_SUCCESS;
}
```

- Your version

```
#include <stdio.h>

int main () {
  printf("Hello!\n");

  return 0;
}
```

- Common ancestor

```
#include <stdio.h>

int main () {
  printf("Hello");

  return 0;
}
```

Tools like `diff3` can solve this

Merging relies on history!

Merging: Problem and Solution

- My version

```
#include <stdio.h>

int main () {
  printf("Hello");

  return EXIT_SUCCESS;
}
```

- Your version

```
#include <stdio.h>

int main () {
  printf("Hello!\n");

  return 0;
}
```

- Common ancestor

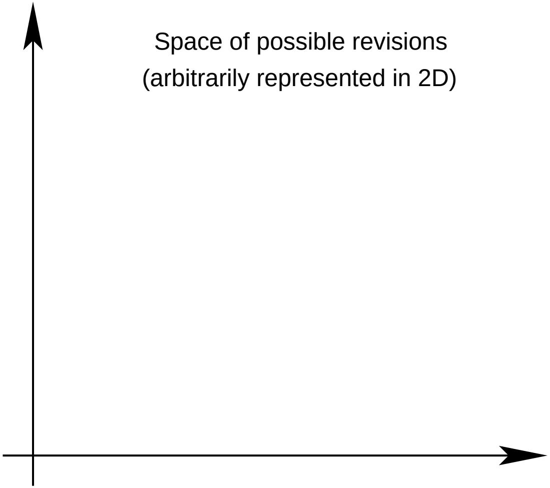
```
#include <stdio.h>

int main () {
  printf("Hello");

  return 0;
}
```

Tools like `diff3` can solve this

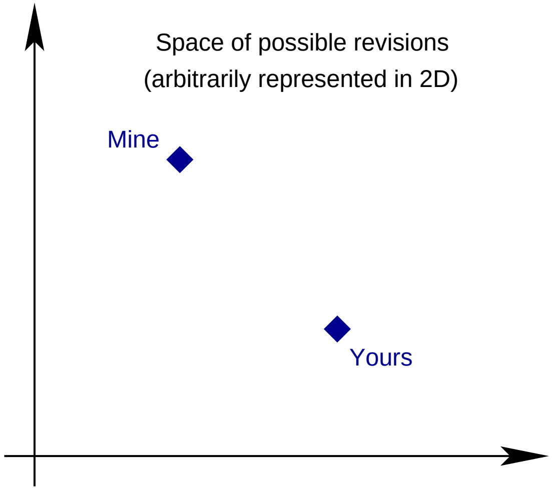
Merging relies on history!

Collaborative development linked to backups

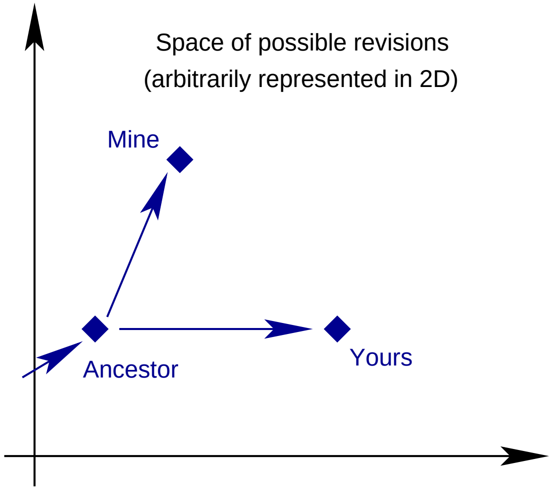
Merging


Space of possible revisions
(arbitrarily represented in 2D)

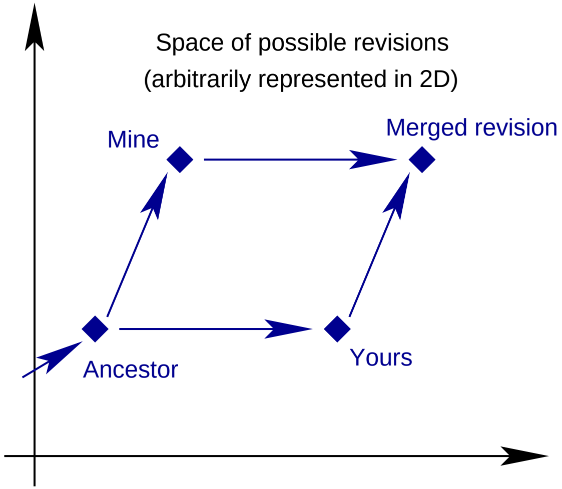
Merging


Space of possible revisions
(arbitrarily represented in 2D)
Mine
Yours

Merging


Space of possible revisions
(arbitrarily represented in 2D)
Mine
Ancestor
Yours

Merging


Space of possible revisions
(arbitrarily represented in 2D)
Mine
Merged revision
Ancestor
Yours

Revision Control System: Basic Idea

- Keep track of history:
  - User makes modification and use `commit` to keep a snapshot of the current state,
  - Meta-data (user's name, date, descriptive message,...) recorded together with the state of the project.
- Use it for merging/collaborative development.
  - Each user works on its own copy,
  - User explicitly "takes" modifications from others when (s)he wants.

Revision Control System: Basic Idea

- Keep track of history:
  - User makes modification and use `commit` to keep a snapshot of the current state,
  - Meta-data (user's name, date, descriptive message,. . .) recorded together with the state of the project.
- Use it for merging/collaborative development.
  - Each user works on its own copy,
  - User explicitly "takes" modifications from others when (s)he wants.
- Efficient storage ("delta-compression" ≈ incremental backups):
  - At least at file level (`git`)
  - Usually store a concatenation of diffs

Revision Control System: Basic Idea

- Keep track of history:
  - User makes modification and use `commit` to keep a snapshot of the current state,
  - Meta-data (user's name, date, descriptive message,...) recorded together with the state of the project.
- Use it for merging/collaborative development.
  - Each user works on its own copy,
  - User explicitly "takes" modifications from others when (s)he wants.
- Efficient storage ("delta-compression" $\approx$ incremental backups):
  - At least at file level (`git`)
  - Usually store a concatenation of diffs
- (Optional) notion of branch:
  - Set of revisions recorded, but not visible in mainline,
  - Can be merged into mainline when ready.

Outline

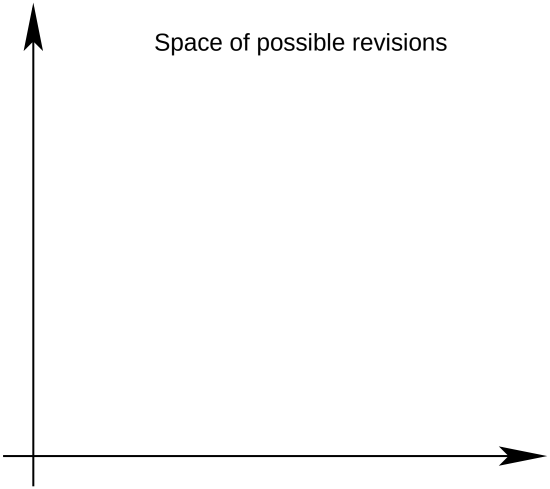
CVS: The Centralized Approach

- Configuration:
  - 1 repository (contains all about the history of the project)
  - 1 working copy per user (contains only the files of the project)
- Basic operations:
  - `checkout`: get a new working copy
  - `update`: update the working copy to include new revisions in the repository
  - `commit`: record a new revision in the repository

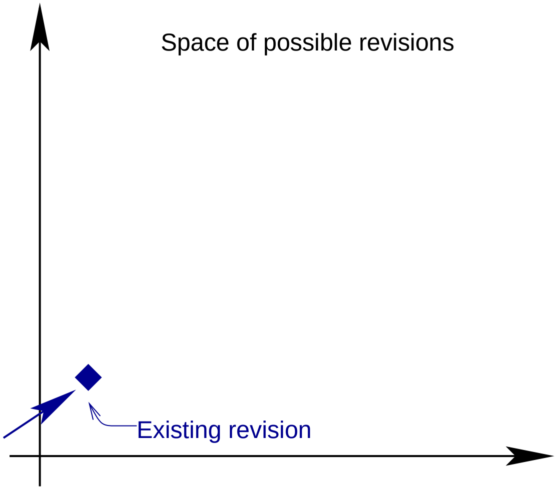
CVS: Example

- Start working on a project:
```
$ cvs checkout project
$ cd project
```
- Work on it:
```
$ vi foo.c      # or whatever
```
- See if other users did something, and if so, get their modifications:
```
$ cvs update
```
- Review local changes:
```
$ cvs diff
```
- Record local changes in the repository (make it visible to others):
```
$ cvs commit -m "Fixed incorrect Hello message"
```

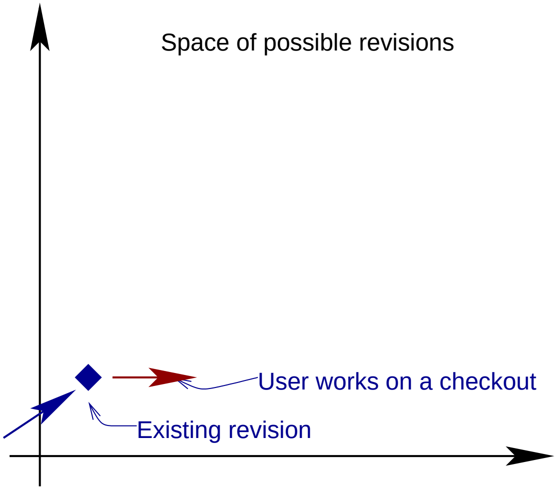
Commit/Update Approach



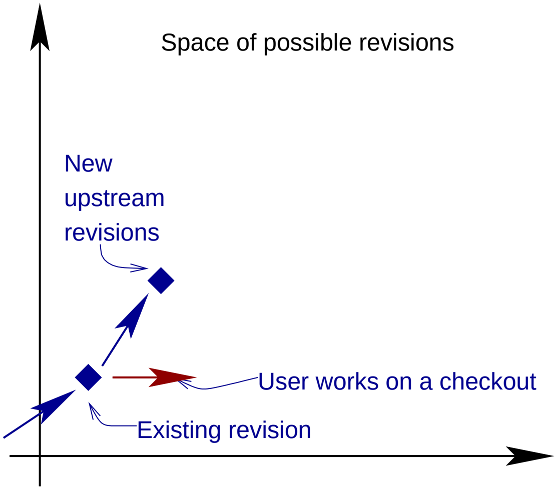
Commit/Update Approach



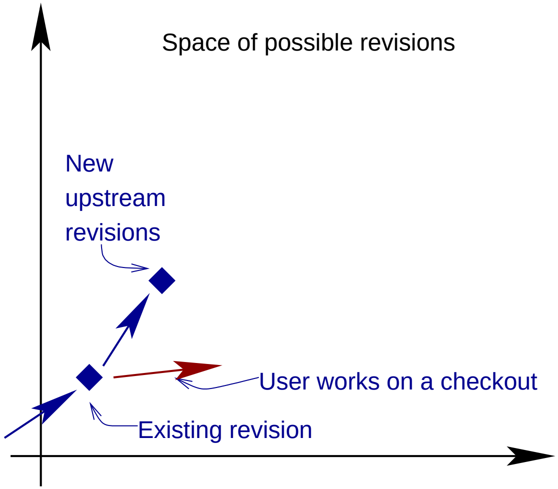
Commit/Update Approach



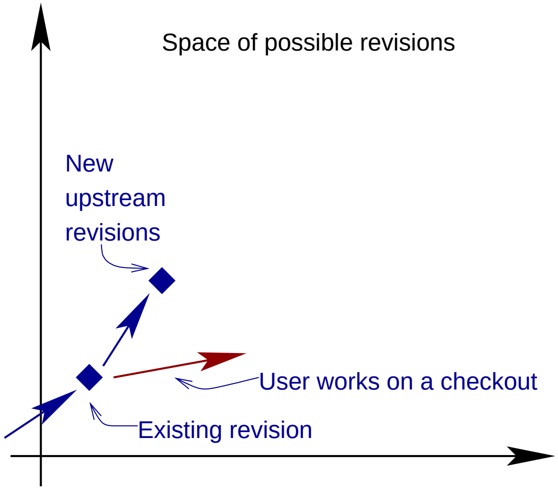
Commit/Update Approach



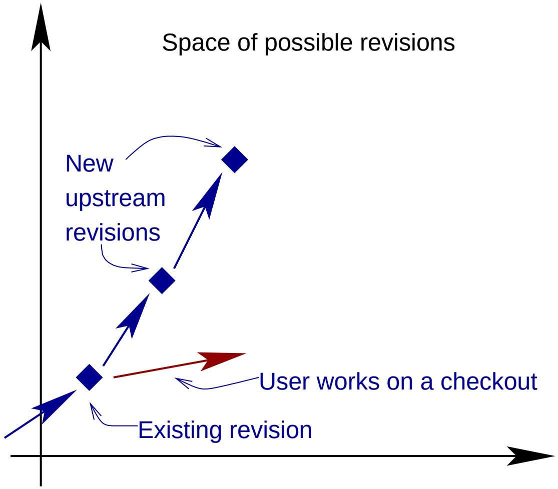
Commit/Update Approach



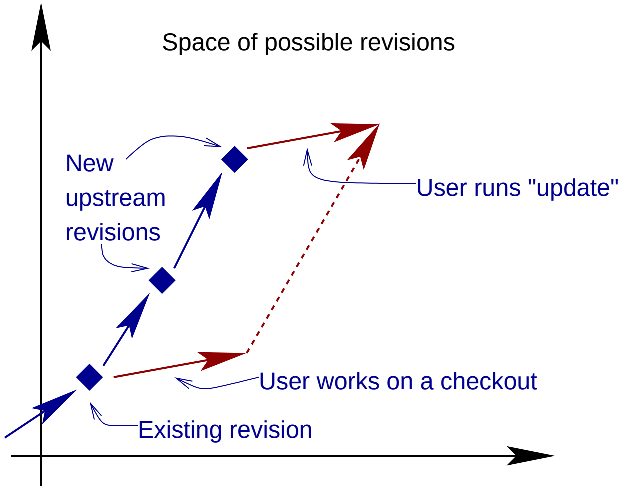
Commit/Update Approach



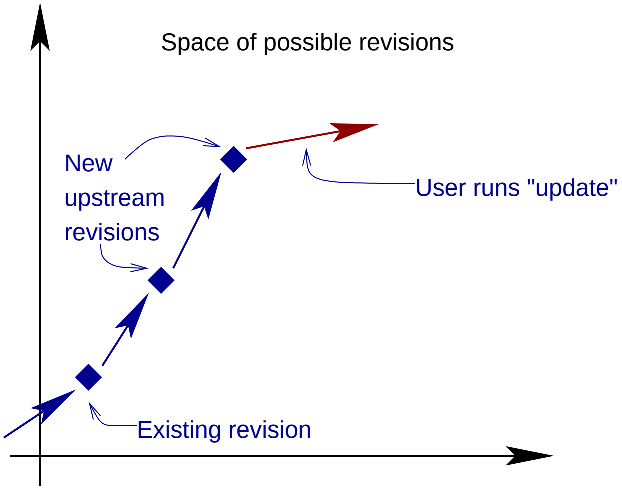
Commit/Update Approach



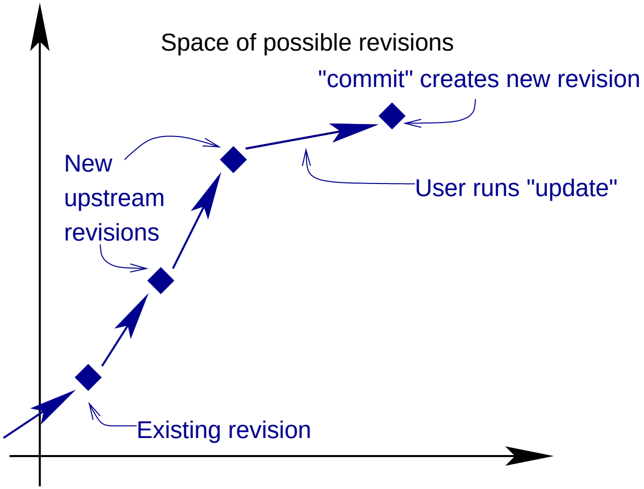
Commit/Update Approach



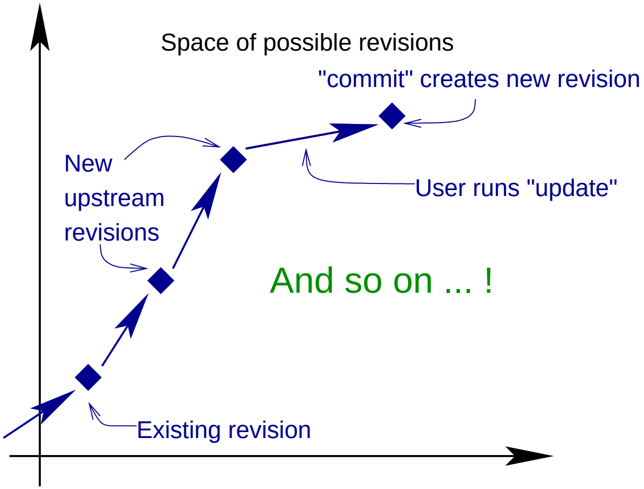
Commit/Update Approach



Commit/Update Approach



Commit/Update Approach



Conflicts

- When several users change the same line of code concurrently,
- Impossible for the tool to guess which version to take,
- $\Rightarrow$ CVS leaves both versions with explicit markers, user resolves manually.
- Merge tools (Emacs's `smerge-mode`, ...) can help.

Conflicts: an Example

- Someone added `\n`, someone else added `!`:

```
#include <stdio.h>

int main () {
<<<<<<< hello.c
  printf("Hello\n");
=======
  printf("Hello!");
>>>>>>> 1.6

  return EXIT_SUCCESS;
}
```

CVS: Obvious Limitations

- File-based system. No easy way to get back to a consistant old revision.
- No management of rename (`remove` + `add`)
- Bad performances

Subversion: A Replacement for CVS

- Idea of subversion: drop-in replacement for CVS (could have been "CVS, version 2", fix the obvious limitation, but no major change/innovation:
  - Atomic, tree-wide commits (commit is either successful or unsuccessful, but not *half*),
  - Rename management,
  - Optimized performances, some operations available offline.

Remaining Limitations

- Weak support for branching,
- Most operations can not be performed offline,
- Permission management:
    - Allowing anyone on earth to commit compromises the security,
    - Denying someone permission to commit means this user can not use most of the features
    - Constraint acceptable for private project, but painful for Free Software in particular.

Decentralized Revision Control Systems

- Idea: not just 1 central repository. Each user has his own repository.
- By default, operations (including `commit` are done on the user's private branch)
- Users publish their repository, and request a merge.

Outline

Linux: A Project With Huge Needs in Version Control

- Not the biggest Open-Source project, but probably the most active,
- $\approx$ 10Mb of patch per month,
- $\approx$ 20,000 files, 280Mb of sources.
- Many branches:
  - Short life: work on a feature in a branch, request merge when ready.
  - Long life: things that are unlikely to get into the official kernel before some time (grsecurity, reiserfs4, SELinux in the past, ...)
  - Test, debug: a modification goes through several branches, is tested there, before getting into mainline
  - Distributor: Most distributions maintain a modified version of Linux

$\Rightarrow$ Centralized revision control is not manageable.

A bit of history

1991:	Linus Torvalds starts writing Linux, using CVS,
2002:	Linux adopts BitKeeper, a proprietary decentralized version control system (available free of cost for Linux),
2002-2005:	Flamewars against BitKeeper, some Free Software alternatives appear (GNU Arch, Darcs, Monotone). None are good enough technically.

A bit of history

1991:	Linus Torvalds starts writing Linux, using CVS,
2002:	Linux adopts BitKeeper, a proprietary decentralized version control system (available free of cost for Linux),
2002-2005:	Flamewars against BitKeeper, some Free Software alternatives appear (GNU Arch, Darcs, Monotone). None are good enough technically.
2005:	BitKeeper's free of cost license revoked. Linux has to migrate.
2005:	Unsatisfied by the alternatives, Linus decides to start his own project, git.

A bit of history

1991:	Linus Torvalds starts writing Linux, using CVS,
2002:	Linux adopts BitKeeper, a proprietary decentralized version control system (available free of cost for Linux),
2002-2005:	Flamewars against BitKeeper, some Free Software alternatives appear (GNU Arch, Darcs, Monotone). None are good enough technically.
2005:	BitKeeper's free of cost license revoked. Linux has to migrate.
2005:	Unsatisfied by the alternatives, Linus decides to start his own project, git.
2006:	Many young, but good projects for decentralized revision control: Darcs, git, Monotone, Mercurial, Bazaar, ...
200?:	Most likely, several projects will continue to compete, but I guess only 2 or 3 of the best will be widely adopted.

Outline

History of Bazaar

GNU Arch:	First Free Software Decentralized Revision Control. Extremely complex for what it does, very slow,
Baz:	Fork of GNU Arch. Unmaintained as of now,
Bazaar:	Complete rewrite of Baz, with different concepts and user interface. “Bazaar” is the name of the project, “`bzr`” is the unix command.

http://bazaar-vcs.org/

Bazaar Concepts

Revision:	State of a project at a point in time, with meta-information,
Repository:	Set of revisions, with ancestry information,
Branch:	Totally ordered (and numbered) set of revisions,
Working tree (aka Checkout):	The project itself (set of files, directories. . .).

Starting a Project

- Create a new, empty project:
```
$ bzr init project
$ cd project
```
- Alternatively, create a project in an existing directory:
```
$ cd existing-project
$ bzr init
```
- This creates a repository, a branch, and a working tree in the same place. Try "`ls .bzr/`" to understand what happened.

Create the First Revision

- Add files (bzr won't touch the files unless you explicitly add them):
  ```
  $ bzr add
  ```
  or individually
  ```
  $ bzr add file1; bzr add file2
  ```
- Commit (record new revision):
  ```
  $ bzr commit -m "descriptive message"
  ```
  (if you don't provide `-m`, an editor will be opened to let you type your message)

Look at Your Own Changes

- Short summary:

```
$ bzr status
added:
  foo.c
modified:
  bar.c
```

- Complete diff:

```
$ bzr diff
=== modified file 'foo.c'
--- foo.c       2006-10-04 18:17:30 +0000
+++ foo.c       2006-10-04 18:17:35 +0000
@@ -1,5 +1,5 @@
 #include <stdio.h>

 int main () {
-  printf("Hello");
+  printf("Hello\n");
 }
```

Look at the History

- See the past revisions:

```
$ bzr log
------------------------------------------------------------
revno: 2
committer: Matthieu Moy <Matthieu.Moy@imag.fr>
branch nick: foo
timestamp: Wed 2006-10-04 23:55:49 +0530
message:
  fixed a bug
------------------------------------------------------------
revno: 1
committer: Matthieu Moy <Matthieu.Moy@imag.fr>
branch nick: foo
timestamp: Wed 2006-10-04 23:47:30 +0530
message:
  initial revision
```

Publish your branch

- Up to now, your branch is just on your hard disk, no one else sees it,
- Publish you branch:
  ```
  $ bzr push sftp://some-host.com/project-upstream
  ```
- Other people can now get their own copy:
  ```
  $ bzr get http://some-host.com/project-upstream
  ```
  (assuming the sftp location and http location are the same on `some-host.com`).

Working on an Existing Project

- Get your own branch:
  ```
  $ bzr branch http://some-host.com/project
  $ cd project
  ```
  (note: `get` is indeed an alias for `branch`).

Working on an Existing Project

- Get your own branch:

  ```
  $ bzr branch http://some-host.com/project
  $ cd project
  ```

  (note: `get` is indeed an alias for `branch`).
- Work on it!
- Commit your changes:

  ```
  $ bzr commit -m "implemented something awesome"
  ```

Working on an Existing Project

- Get your own branch:
```
$ bzr branch http://some-host.com/project
$ cd project
```
(note: `get` is indeed an alias for `branch`).
- Work on it!
- Commit your changes:
```
$ bzr commit -m "implemented something awesome"
```
- Publish it and request a merge:
```
$ bzr push sftp://my.isp.com/project-contrib/
$ mail -s "please, merge ..."
```

Merging

- Two use cases:
  - A contributor started working on a feature in your own branch, but you want to follow upstream development.
  - The contributor's feature is completed, upstream wants to merge it.
- Symetry in both use-cases,
- Successive merge possible,
- Bazaar keeps track of merge history. It knows what you miss, and what has already been merged.

Merging

- Merge the changes into the working tree:

```
$ bzr merge ../bar/
All changes applied successfully.
```

Merging

- Merge the changes into the working tree:

```
$ bzr merge ../bar/
All changes applied successfully.
```

- Check what happened:

```
$ bzr status
modified:
  foo.c
pending merges:
  Matthieu Moy 2006-10-05 implemented something awesome
```

Merging

- Merge the changes into the working tree:

```
$ bzr merge ../bar/
All changes applied successfully.
```

- Check what happened:

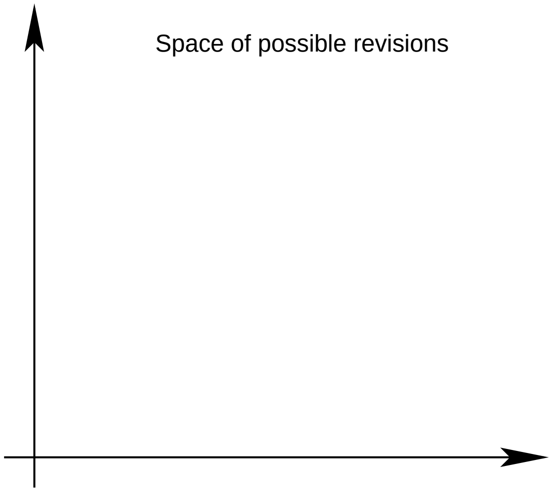
```
$ bzr status
modified:
  foo.c
pending merges:
  Matthieu Moy 2006-10-05 implemented something awesome
```

- Commit:

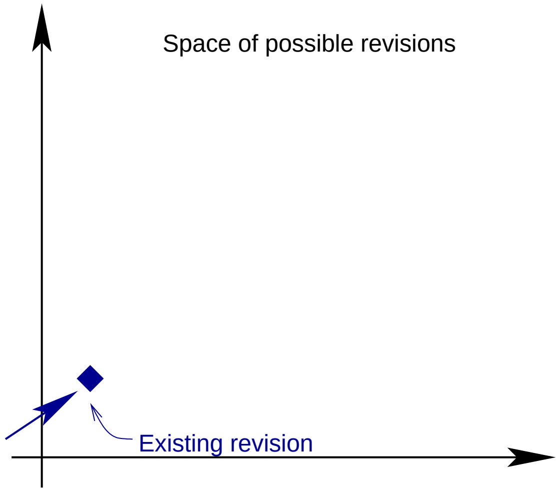
```
$ bzr commit -m "merged awesome feature from X"
Committed revision 3.
```

When commiting, bzr records both the previous revision and the merged revision as ancestor.

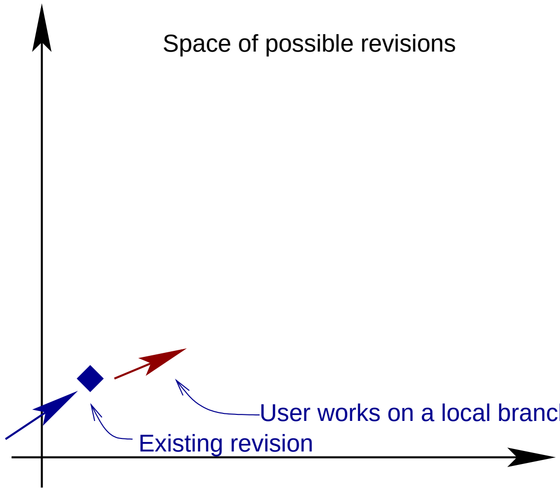
Merging



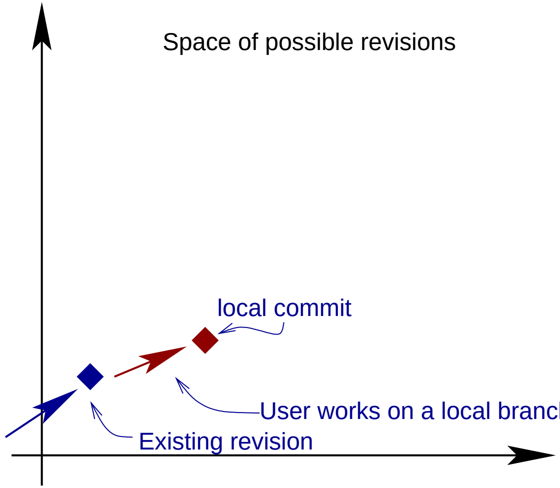
Merging



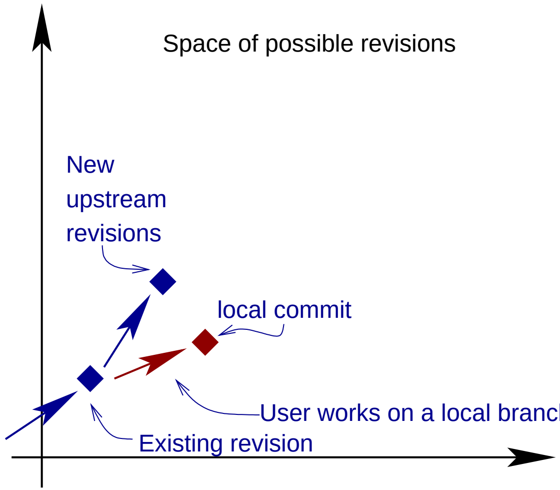
Merging


Space of possible revisions
User works on a local branch
Existing revision

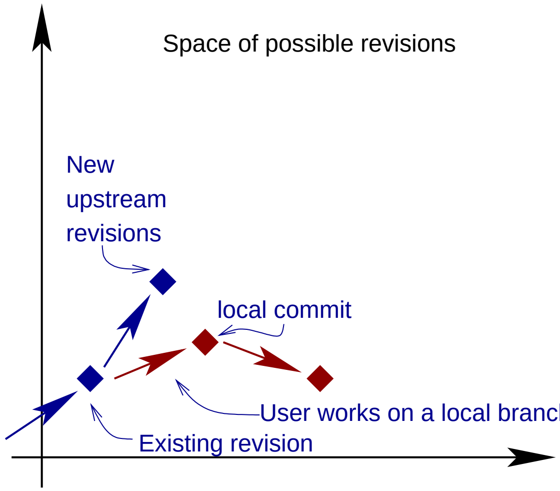
Merging


Space of possible revisions
local commit
User works on a local branch
Existing revision

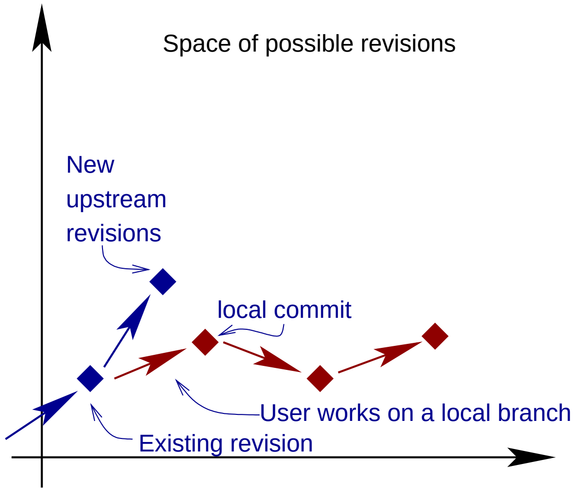
Merging


Space of possible revisions
New upstream revisions
local commit
User works on a local branch
Existing revision

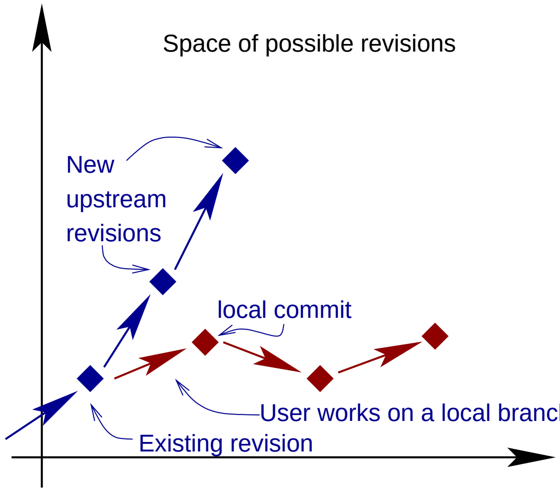
Merging


Space of possible revisions
New upstream revisions
local commit
User works on a local branch
Existing revision

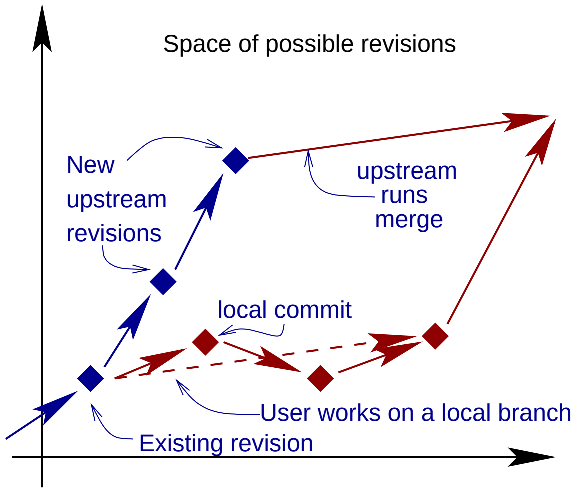
Merging


Space of possible revisions
New upstream revisions
local commit
User works on a local branch
Existing revision

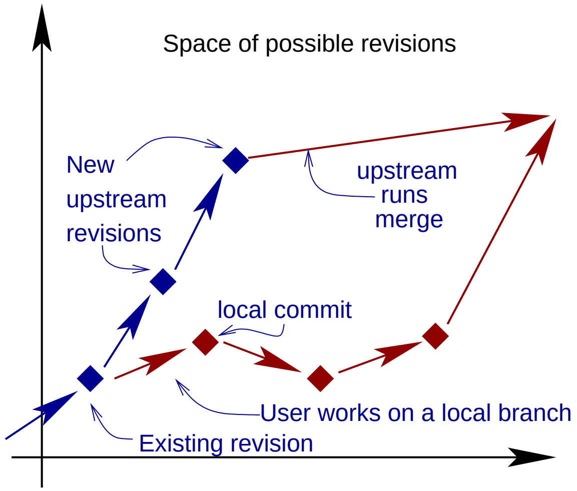
Merging


Space of possible revisions
New
upstream
revisions
local commit
User works on a local branch
Existing revision

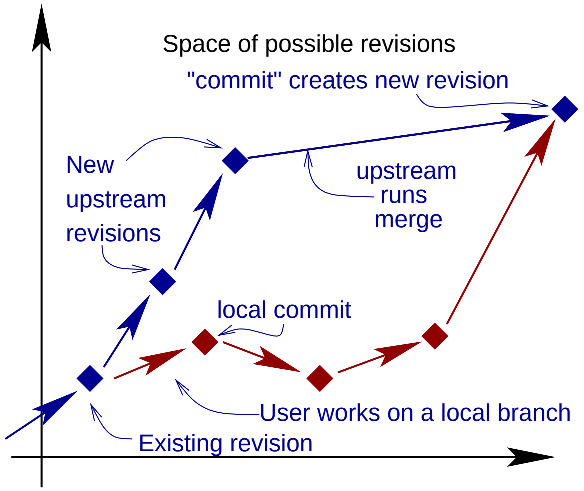
Merging


Space of possible revisions
New
upstream
revisions
upstream
runs
merge
local commit
User works on a local branch
Existing revision

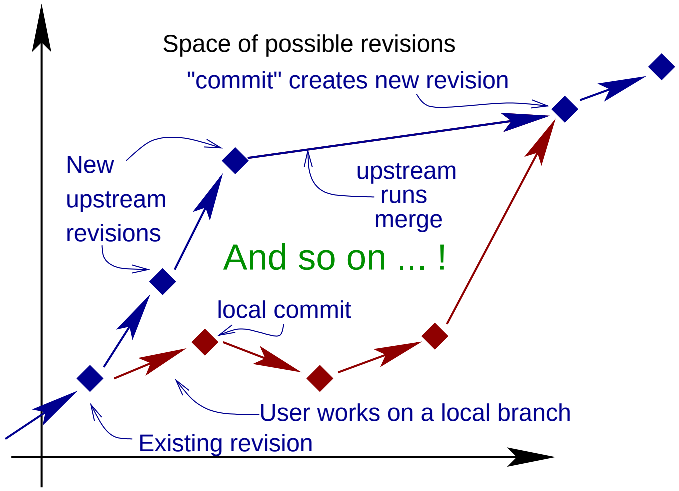
Merging


Space of possible revisions
New
upstream
revisions
upstream
runs
merge
local commit
User works on a local branch
Existing revision

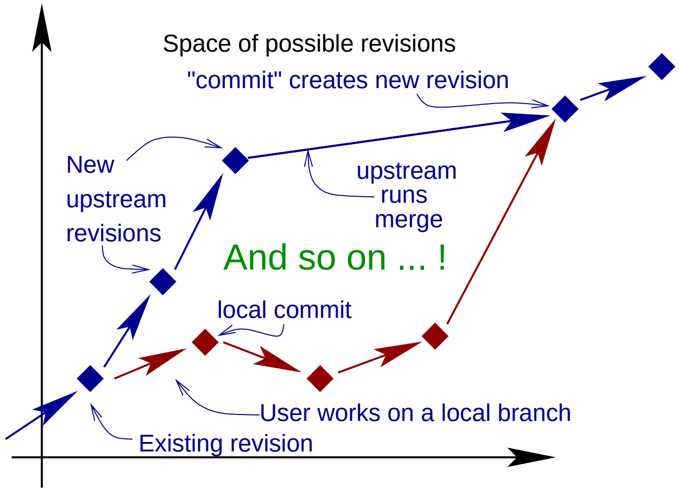
Merging


Space of possible revisions
"commit" creates new revision
New
upstream
revisions
upstream
runs
merge
local commit
User works on a local branch
Existing revision

Merging


Space of possible revisions
"commit" creates new revision
New upstream revisions
upstream runs merge
And so on ... !
local commit
User works on a local branch
Existing revision

Merging



Resulting revision history is a DAG

Other Features of Interest

Light Checkout: A working tree pointing to an branch located somewhere else (a la CVS). `bzr update` to get changes from the branch into the working tree,

Heavy Checkout: A working tree plus a duplicate of the branch used as a cache. Allows local commits (`bzr commit --local`),

Shared repository: Multiple branches sharing the common revisions for storage,

Revision Bundle: Pack a set of revisions in a single file (to be sent by email and merged in another branch for example), together with a human-readable diff,

Plugins: Extensibility via a plugin system in Python,

Foreign Branches: Experimental plugins to access a Subversion branch directly with `bzr`.

Outline

Benefit of Version Control

- Working alone:
  - Possibility to revert to a previous revision,
  - Makes it easy to review your own code (before committing),
  - Synchronization of multiple machines.
- Collaborative development:
  - One can work without disturbing others,
  - Merge is automated.

Benefit of Version Control

- Working alone:
  - Possibility to revert to a previous revision,
  - Makes it easy to review your own code (before committing),
  - Synchronization of multiple machines.
- Collaborative development:
  - One can work without disturbing others,
  - Merge is automated.

Text editing without version control is like
sky diving without a parachute!

Benefit of *Decentralized* Version Control

- Easy branch/merge,
- Simplifies permission management
  (no need to give any permission to other users),
- Disconnected operation
  (useful for laptop users in particular).

Other Decentralized Version Control Systems

Monotone:	A clever system based on hashes (SHA1). Inspired git a lot. http://venge.net/monotone/
git:	Designed for speed. Used by the Linux kernel, http://git.or.cz/
Mercurial:	Close in concepts and performance to git. Written in python, with a plugin system. http://www.selenic.com/mercurial/
Darcs:	Based on a powerful patch theory. Was the first system to have a really simple user-interface. http://abridgegame.org/darcs/
SVK:	Distributed Version Control built on top of Subversion. http://svk.bestpractical.com/

Emacs Users

[Warning: Self advertisement]

- Most version control systems have an Emacs integration.
- Check out DVC: `http://download.gna.org/dvc/`

Version Control and Backups

- Version Control is a good complement for backups
- But your repository should be backed-up/replicated !
  (many users lost their data and their revision history at the same time with a disk crash)
- Usually:
  - Version Control = User side (manual creation of project, manual add of source files, manual commits, . . .)
  - Backup = System Administrator side (cron job, backing up everything)

Last Word on Backups

- Don't trust your hard disk,
- Don't trust a CD (too short life),
- Don't trust yourself,
- Don't trust Anything!
- REPLICATE!!!
    - Multiple machines for normal work
    - Multiple sites for important work (are you ready to loose you thesis if your house or lab burns?)

Learn More

Bazaar: http://bazaar-vcs.org/

Bazaar Docs: http://doc.bazaar-vcs.org/

Version Control: http://en.wikipedia.org/wiki/Revision_control

This presentation:

http://www-verimag.imag.fr/~moy/slides/bzr/