# Everything* you wanted to know about revision control but never dared to ask

*OK, most things I think are useful and that I can (try to) fit into a 20 minute talk

David Cottingham
david.cottingham@cl.cam.ac.uk

# What is Revision Control?

- Poor man's version:
  - Keep multiple copies of the same file, one for each change ever made, and some notes
- Keep track of changes to a file
- Roll back (undo) changes if needed
- Merge multiple changes together to create the next revision

# Where to Use Revision Control?

- Your own software projects (return to a stable version when necessary)
- Documents (particularly papers, where you are likely to re-write sections)
- Working collaboratively on either of these (can merge in changes from multiple reviewers, keep track of original version)

# Concurrent vs. Strict Locking

- Version control exists in many applications
  - Simplest form is an undo log!
  - MS Word has “Track Changes” feature
- These are great for keeping track of changes to a document that is passed around between users (i.e. only one person edits at once)
- Revision Control Systems (a.k.a. Software Configuration Management [SCM] systems) allow concurrent editing of files by multiple users

# Contents & “Non-Contents”

- We **will** cover:
  - Basic revision control (import, check out, commit, update, diff, merge, revert/rollback)
  - Advanced revision control (branching)
  - Different revision control paradigms (centralised vs. distributed)
- We will **not** cover:
  - In depth command line options of any one system (though there are some reference slides)

# Quick Comparison I

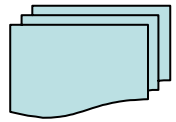
- RCS: not network-aware, only files (not projects), strict locking rather than optimistic concurrency, useful for single config files
- CVS: built on RCS, but fixes above, widely used, non-atomic commits, no directory versioning, no binary files
- SVN: fixes above, but uses Berkeley DB, so you can't view metadata with a text editor. Allows property sets (text, binary files…) for all objects in repository

# Quick Comparison II

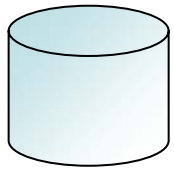
- Mercurial: great for giving each developer their own repository, slightly confusing first time round, lots of hooks, bisect to find bugs, patch queues, various extensions
- Git (not mentioned here): also distributed, used for Linux kernel source management
- Many others…

# (Very Basic) Terminology

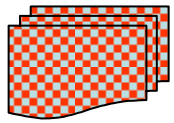
- Revision: a set of changes to a project. A revision is a snapshot of a project



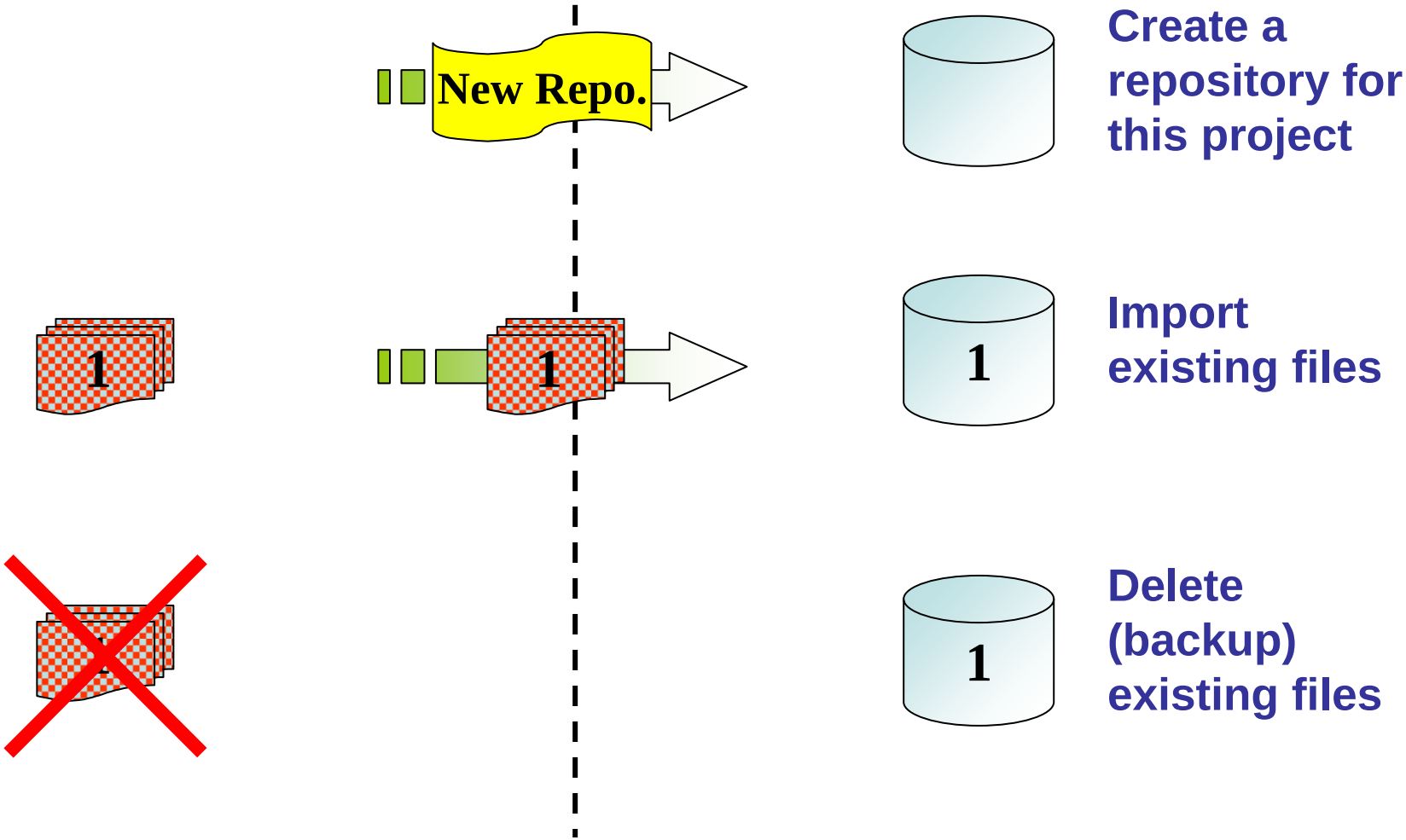
- Repository: master copy of a project's revision history (on a "server")



- Working Copy: your own "workstation" copy of the project (you can mess this up as much as you like)



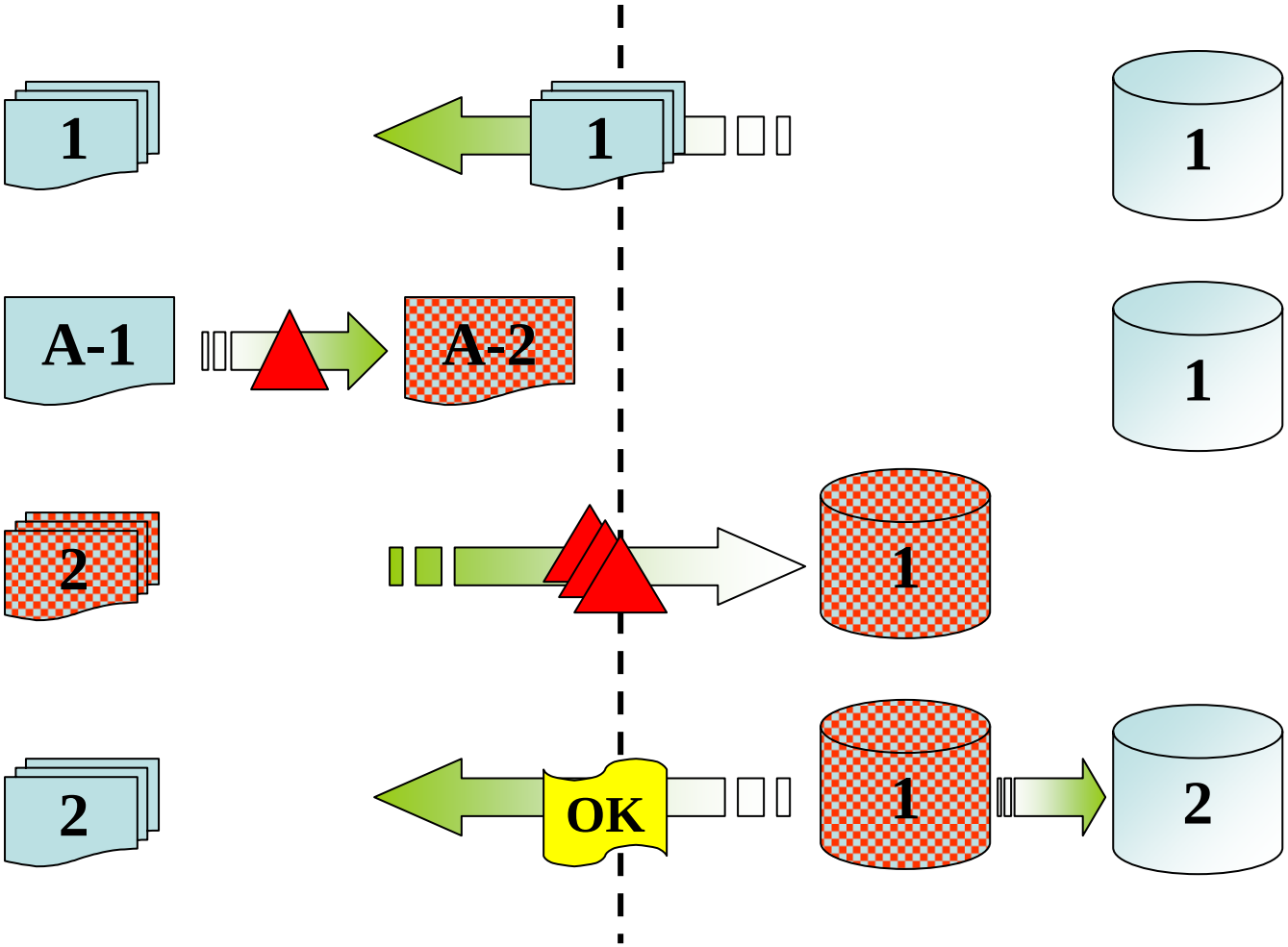
# Importing (a.k.a. Starting a New Repository)



# Creating a Repository & Importing in SVN

```
foss:~$ mkdir –p ~/svn/testRepo
foss:~$ svnadmin create –fs-type fsfs ~/svn/testRepo
foss:~$ ls ~/svn/testRepo
conf dav db format hooks locks README.txt
foss:~$ svn import ~/scripts \
  file:///filer/dnc25/unix_home/svn/testRepo/trunk/\
  -m “Project import.”
Adding scripts/colour.pl
Adding scripts/readme.txt

Committed revision 1.
foss:~$ mv ~/scripts ~/scripts-old
foss:~$ svn list file://~/svn/testRepo/trunk/
colour.pl
readme.txt
```

# Checking Out & Committing




**Check out working copy at revision 1**

**Do work on file A**

**Commit changes to repository**

**Resolve conflicts, store new revision**

# Checking Out in SVN

```
foss:~$ mkdir ~/workingCopy; cd ~/workingCopy
foss:workingCopy$ svn co
  file://~/svn/testRepo/trunk/
A trunk/colour.pl
A trunk/readme.txt
Checked out revision 1.
foss:workingCopy$ cd trunk; cat colour.pl
#! /bin/perl
print “The following colours are available:\n”;
my $colours = `showrgb`;
print $colours;
foss:trunk$ perl colour.pl
The following colours are available:
255 250 250          snow
248 248 255          ghost white
```

# Committing in SVN

```
foss:trunk$ perl colour.pl
The following colours are available:
255 250 250         snow
248 248 255         ghost white
foss:trunk$ vi colour.pl
foss:trunk$ perl colour.pl
The following colours are available:
snow
ghost white
foss:trunk$ svn commit –m “Made colour listing only\
  names.”
Sending   colour.pl
Transmitting file data .
Committed revision 2.
```

# What Changed? `diff` In Action

```
foss:trunk$ svn diff -r PREV
Index: colour.pl
===================================================================
--- colour.pl   (revision 1)
+++ colour.pl   (working copy)
@@ -1,4 +1,8 @@
 #! /bin/perl
 print "The following colours are available:\n";
 my $colours = `showrgb`;
-print $colours;
+while ($colours) {
+ if ($colours =~
   s/^\s*\d{1,3}\s*\d{1,3}\s*\d{1,3}\s*(.*?)\n//x) {
+    print "$1\n";
+ }
+}
```

# `diff` in Action Explained

```
foss:trunk$ svn diff -r PREV
Index: colour.pl
===================================================================
--- colour.pl   (revision 1)
+++ colour.pl   (working copy)
@@ -1,4 +1,8 @@
 #! /bin/perl
 print "The following colours are
 my $colours = `showrgb`;
-print $colours;
+while ($colours) {
+ if ($colours =~
   s/^\s*\d{1,3}\s*\d{1,3}\s*\d{1,3}\s*(.*?)\n//x) {
+    print "$1\n";
+ }
+}
```

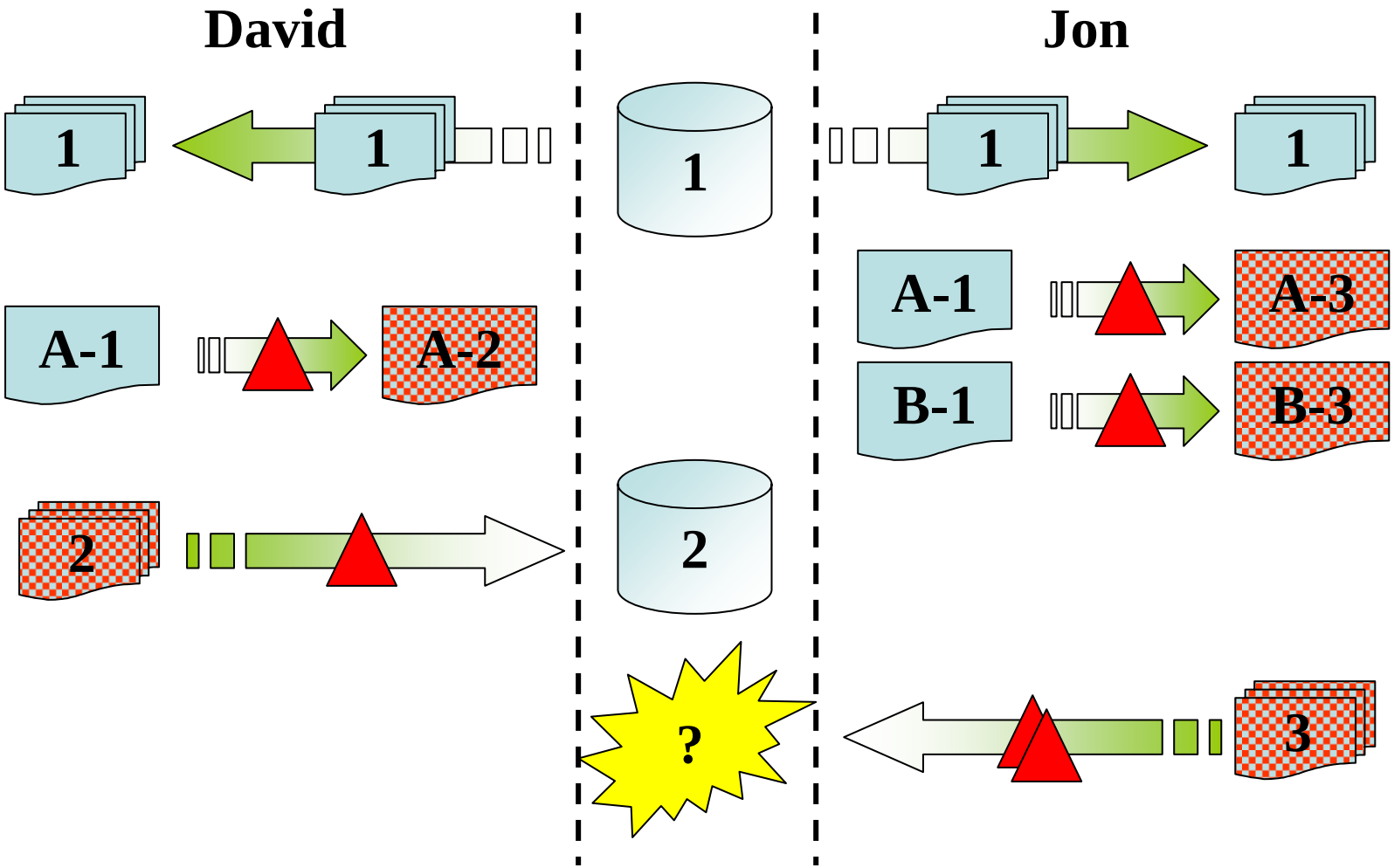
**Original lines 1-4**

**Showing new 1-8**

**Removed line**

**Inserted 5 lines**

# Collaboration & Conflict



# What a Conflict Looks Like

```
foss:trunk$ vi readme.txt
foss:trunk$ svn commit
Sending        readme.txt
svn svn: Commit failed (details follow):
svn: Out of date: '/trunk/readme.txt' in transaction
   '3-1'
svn: Your commit message was left in a temporary
   file:
svn: '/filer/dnc25/unix_home/workingCopy/trunk/svn-
   commit.tmp‘
foss:trunk$ svn update
C  readme.txt
Updated to revision 3.
```

# Update: What Changed?

- The `update` command will compare your working copy to the repository
- Reports which files have changed, and merges them if possible, but does nothing further (i.e. conflicts are your problem)
- (Note: `commit` only pushes your changes to repository; you probably then want to `update`)

# Merging Conflicts I

- The revision control system can work out simple “conflicts” itself (e.g. two separate portions of a file)
- A conflict occurs if there have been concurrent edits to the same line of a file
- If this happens, your commit will fail, and you will need to `update`, [rack your brains], [write upset e-mail to co-worker], sort it out manually
- Then tell SVN that the conflict is resolved, and commit the fix back to the repository

# Merging Conflicts II

```
foss:trunk$ cat readme.txt
<<<<<<< .mine
This is a readme file for colour.pl!
=======
This is a useless readme file for colour.pl.
>>>>>>> .r3
Don't ever modify this code!
```

# Merging Conflicts II Explained

```
foss:trunk$ cat readme.txt
<<<<<<< .mine
This is a readme file for colour.pl!
=======
This is a useless readme file for colour.pl.
>>>>>>> .r3
Don't ever modify this code!
```

**Working copy change**

**Repo. change**

**Repo. Revision #**

**Contextual line**

# Resolving the Conflict

```
foss:trunk$ vi readme.txt
foss:trunk$ cat readme.txt
This is a useless readme file for colour.pl!
Don't ever modify this code!
foss:trunk$ svn resolved readme.txt
Resolved conflicted state of 'readme.txt'
foss:trunk$ svn commit -m "Merged xyz12's change of
  first sentence with mine."
Sending        readme.txt
Transmitting file data .
Committed revision 4.
foss:trunk$ svn commit –m “Merged xyz12’s change of
  first sentence with mine.”
```

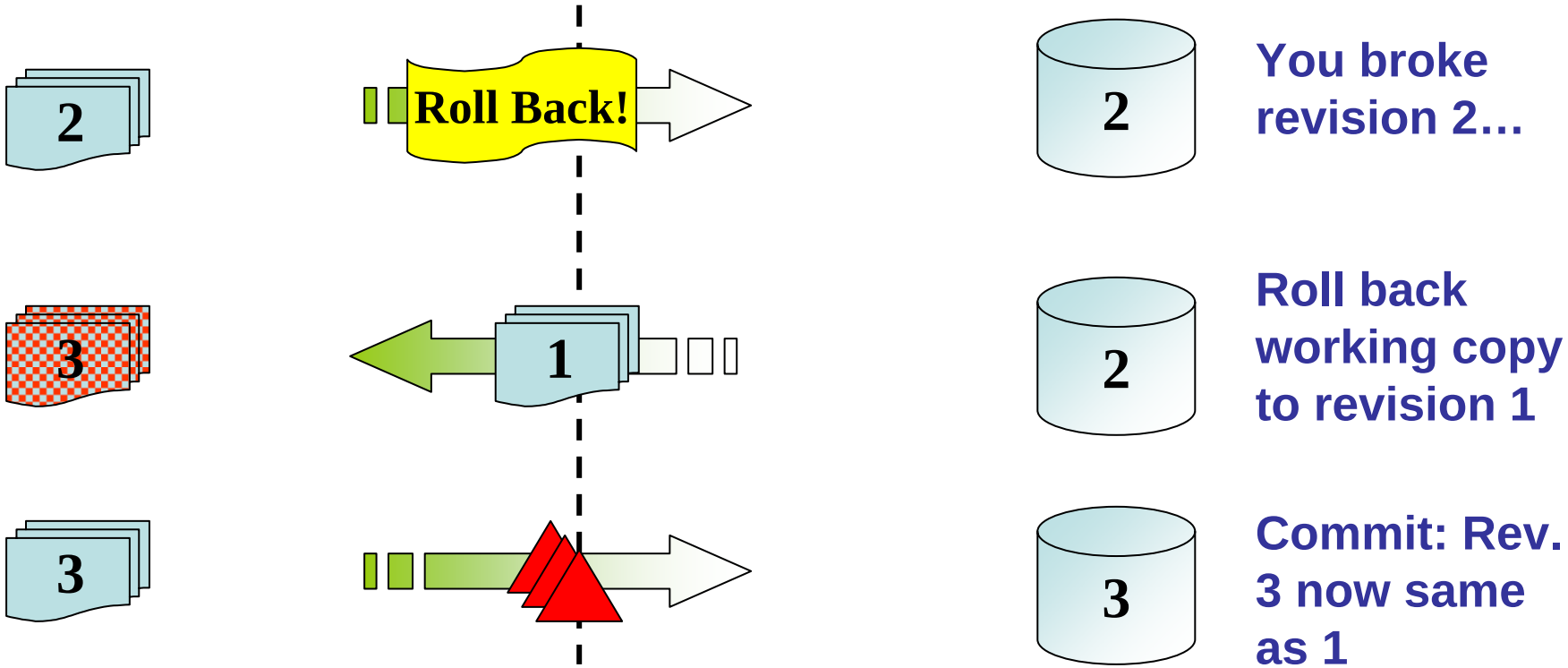
# Revision/Version Numbers & Change Log I

- SVN has a single revision number for the entire repository
- CVS keeps version numbers on a per-file basis
- Each commit increments the revision/version number
- Every commit should have a comment from the committer explaining what the change is; the change log contains all of these

# Revision/Version Numbers & Change Log II

```
foss:trunk$ svn log readme.txt
------------------------------------------------------------------------
r4 | dnc25 | 2008-02-01 14:23:45 +0000 (Fri, 01 Feb 2008) | 1
   line

Merged xyz12's change of first sentence with mine.
------------------------------------------------------------------------
r3 | xyz12 | 2008-02-01 13:59:58 +0000 (Fri, 01 Feb 2008) | 2
   lines

Told everyone that the readme file was useless

------------------------------------------------------------------------
r1 | dnc25 | 2008-02-01 13:24:52 +0000 (Fri, 01 Feb 2008) | 1
   line

Project import.
------------------------------------------------------------------------
```

# Rolling Back (Reverting)




- Get earlier revision, then commit it as a change
- Roll back as far as you wish
- CVS Works on individual files, entire repository more complex

# Reverting to a Previous Revision

```
foss:trunk$ foss:trunk$ svn diff -r 2
Index: readme.txt
===================================================================
--- readme.txt  (revision 2)
+++ readme.txt  (working copy)
@@ -1,2 +1,2 @@
-This is a readme file for colour.pl.
+This is a useless readme file for colour.pl!
 Don't ever modify this code!
foss:trunk$ svn cat -r 2
   file://~/svn/testRepo/trunk/readme.txt > readme.txt
foss:trunk$ svn commit -m "Reverted readme text to before
   'useless' was added."
Sending        readme.txt
Transmitting file data .
Committed revision 5.
```

# Reverting an Entire Repository

- For **SVN** this is the same as for a single file, as a revision number is for the **entire repository**
- Remember: **in CVS** every file has its **own** revision number
- Hence, when you release version 1.0 of your programme, you may well have some files that are at revision 2.5, 1.4, 0.9…
- To revert a repository, can do so by date (latest revision of all files up to that date)
- Or: tag (give a useful name to) a set containing one revision of each file. Revert to a tagged set
- (SVN also supports tag and date reversion)

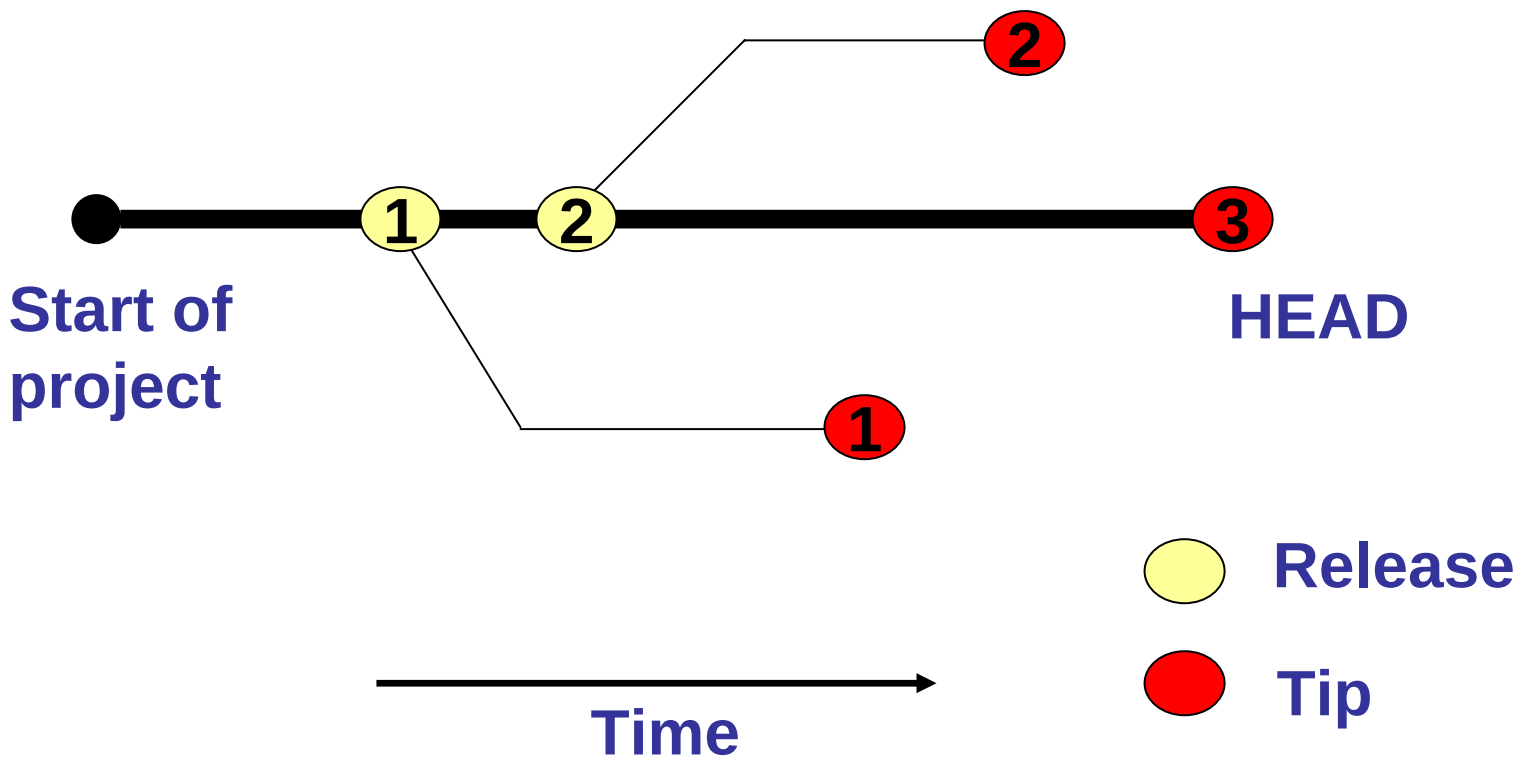
# Branching Timelines

- Consider a time machine… (Think of Doc. Brown from “Back to the Future” here)
- A timeline can split into two (or more) versions at a particular event (the root)
- For example, you could choose to listen or not listen to this talk
- In one timeline, I would get upset and walk out, in the other I will be generally happy

# Branching Repositories

- When you release a version of your software project, you may still have to supply updates for it, whilst developing a completely new (currently unstable) version
- So: when beginning on the new version, make a copy of the repository's contents: a **branch**
- Leave this branch intact, so you can work on it separately if needed, whilst you continue to work on the "main" repository
- The root of the branch is the release event, which is contained within the main development tree, the **trunk**

# Trunk, Branches, Tips I



# Trunk, Branches, Tips II

- Generally, your repository will be tree-like in structure
- Your **trunk** will be the new release of the software you're working on
- Older releases will each have **branches**, rooted at particular points on the trunk
- Each branch will have a **tip**: the latest commit made to that branch (confusingly, git calls these heads)
- Example: bug fix needed for release 1.0 of the software, whilst you're working on release 3.0
- The trunk has a tip (called the **HEAD** in CVS and SVN): the most recent commit to release 3.0
- In SVN, the trunk is no different from any other branch. In CVS the trunk is special

# Merging From a Branch Into the Trunk

- It may be that you need to apply all the bug fixes that you made to the 1.0 branch to release 3.0 (the trunk) as well
- When merging a branch into the trunk, the system calculates the diff between the root of **that** branch, and its tip
- That diff is then applied to the **trunk's** tip
- (Note: this is **not** the same as reverting the entire trunk to the previous release!)
- You can also merge from the trunk into a branch (e.g. `cvs update -j HEAD`)

# Caution: Multiple Merges => Conflict

- If you merge a branch into the trunk, then later do further work on the branch, (i.e. create a new tip), then re-merge, this generates a conflict
- This is because the diff (that is attempted) to be applied to the trunk is of the branch root and its tip (i.e. including the contents of your first merge)
- Fix: tag the branch before every merge. Then on merging, elect to merge in the changes that were committed to the branch *after* the last tag. Or use `svn merge` which is clever.
- (If you don't tag, you can use dates instead)
- Caution: if you merge in someone else's branch, make sure the tip of that branch is then tagged!

# Branching & Merging Example (SVN) I

- Do some work on an existing trunk
- Commit your changes (becomes revision 13)
- Do some more, commit (revision 14)
- Oh… Actually:
  - Revision 14 was bug fixes to trunk
  - Revision 13 should have been a branch
- Problems:
  - How to make a branch containing changes from 13, and the bug fixes?
  - How to get trunk back to revision 12 + the bug fixes from 14?

# Branching & Merging Example (SVN) II

- How to make a branch containing changes from 13, and the bug fixes?
  - Create a branch at revision 12 (i.e. the point before you committed the changes that should be on the branch)
  - Switch your working copy to point to the branch
  - Merge changes from revisions 13 to 14 of the trunk into your working copy
  - Commit your working copy to the branch
- Notes:
  - If you have no working copy, just check out the branch
  - Switching is preferable because it doesn't download the entire repository (just the differences)

# Branching & Merging Example (SVN) III

- How to get trunk back to revision 12 + the bug fixes from 14?
  - EITHER: Check out revision 12 of the trunk to a scratch space ("working copy")
  - Merge in (only) changeset 14 (the bug fixes) to your working copy
  - OR: use merge to reverse changeset 13 only (rather than revert)
  - Commit your working copy to the trunk (revision 15)
- Notes:
  - You never actually *delete* anything from the history: revision 14 stays as 13 + the bug fixes.
  - You can specify a range of changesets (rather than just a single one) that you want to apply to your working copy

# CVS: Watchers & Notifications

- Useful if you want to know by e-mail/SMS when someone begins to edit (or commits) on a file you're interested in
- For edit notification, requires you to tell the system that you are about to edit (e.g. `cvs edit` *`fileName`*)
- Very cumbersome. SVN and Mercurial provide hooks which require much less effort to use

# CVS: Branching Commands

- Create a tagged branch (drop -b to tag without branching)
  `cvs tag –b branch`*`Name`*
- Checkout the tip of a branch
  `cvs checkout -r branch`*`Name`*
- Update working copy to trunk (remove stickiness)
  `cvs update -A`
- Update working copy with diffs from a specified tagged branch
  `cvs update -j `*`tagName`*
- Merge a branch's changes after a specified tagged commit to that branch
  `cvs update -j `*`tagName`*` -j `*`branchName`*

# SVN: Properties

- Every object in the repository can have metadata associated with it
- This can be textual (e.g. copyright notice, bug ID) or binary (e.g. a thumbnail image)
- Use `svn propset` to set, then `svn proplist` to get the property names, `svn propget` to obtain the actual property value
- Properties can be versioned or unversioned
- SVN provides language bindings to easily access properties

# SVN: Unique Commands

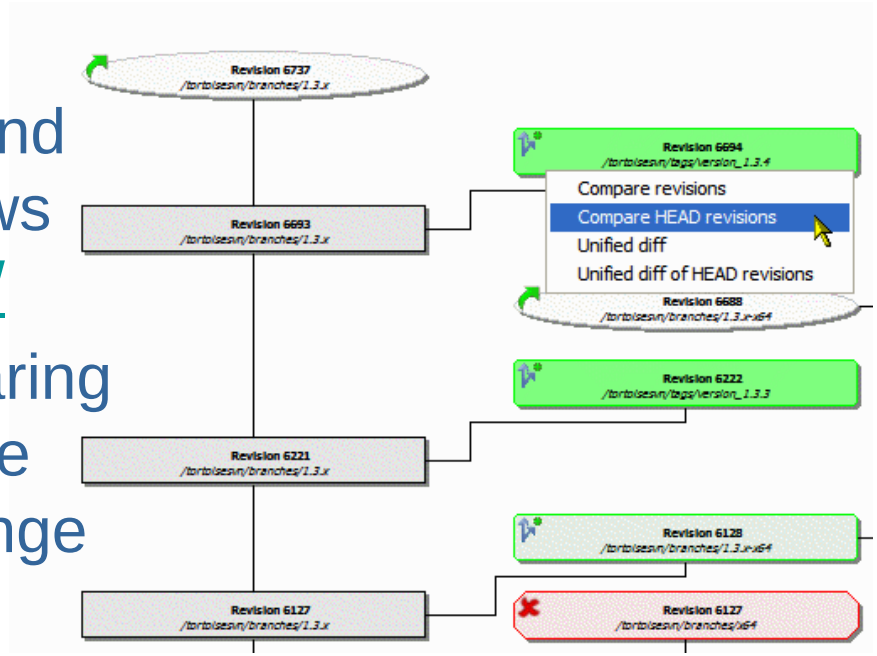
- View log of changes to a file or directory
  `svn log [fileName] [-r revisionNumber]`
- Revert to the last repository copy of a file
  `svn revert fileName`
- See a particular revision of a given file
  `svn cat -r revisionNumber fileName`
- See files in a given directory without downloading
  `svn list [-v] URL`
- Use journal to restore working copy to consistent state
  `svn cleanup`
- To copy, rename, or move files, use
  `svn copy`, `svn rename`, `svn move`

# SVN Branching Commands

- Create a branch (just copy to elsewhere in the repo.)
  `svn copy trunkURL branchURL`
- Download another branch into your working copy
  `svn switch [-r revisionNo] branchURL`
- Merge changes from trunk into branch (as many times as you like), optionally specify trunk revision number
  `svn merge [-c revisionNo] trunkURL`
- Track what changes from trunk have been merged into your current branch (using properties, only in 1.5)
  `svn mergeinfo .`
- Merge branch into a working copy of trunk
  `svn merge --reintegrate branchURL`

# Making Life Easier: Utilities

- Integrate SVN into Eclipse for DTG use: http://www.cl.cam.ac.uk/research/dtg/research/wiki/local/SharingCodeInDTG
- Tortoise: Graphical front end for SVN under MS Windows http://tortoisesvn.tigris.org/
- Kidff3 is brilliant for comparing files in a graphical interface and working out what change to keep http://kdiff3.sourceforge.net/



# Distributed Version Control

- RCS, CVS and SVN are all centralised, i.e. there is one “most up-to-date” repository that everyone commits to
- Mercurial and Git have the idea that all repositories are equal, and contain the entire change history
- Note that you’ll still probably **want** one central backup and up-to-date repository anyway, but you don’t **have** to.

# Mercurial (hg)

- To “checkout” a project, you `hg clone` the entire repository
- You then have **your own** local repository (not just a working copy!)
- Can `hg commit` to your repository
- Get changes from other repos using `hg pull`
- Merge your repo. with the result of the pull using `hg update` plus `hg merge`
- Push the result to other repos using `hg push`

# Why Bother With Mercurial?

- Having your own repository allows you to commit intermediate code states regularly, but not publish them to anyone else
- Bisect extension: allows you to mark earliest changeset you know of where a bug is present, and the latest where it is not. Then bisect iterates through changesets, updating your working copy. For each, you inform it whether your binary bug test fails. This then enables you to find the exact changeset that introduced the bug
- Lots of hooks (even more than SVN) that allow actions to be run in response to repository events
- Patch queues

# Links for Reference

- CVS Book: http://cvsbook.red-bean.com/OSDevWithCVS_3E.pdf
- SVN Book: http://svnbook.red-bean.com/nightly/en/svn-book.pdf
- Mercurial Book: http://hgbook.red-bean.com/hgbook.pdf
- Git User Manual: http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

# Conclusions

- There are a huge number of revision control systems out there
- We have examined CVS and SVN in detail (centralised approach)
- Briefly looked at Mercurial (distributed)
- Personally: CVS is OK, SVN far better but database locks up (use fsfs instead). Mercurial (thus far) has been great

# Everything* you wanted to know about revision control but never dared to ask

Thanks for listening!

David Cottingham
david.cottingham@cl.cam.ac.uk