Construction: version control and system building

Paul Jackson

School of Informatics
University of Edinburgh

The problem of systems changing

- Systems are constantly changing through development and use
  - Requirements change and systems evolve to match
  - bugs found and fixed
  - new hardware and software environments are targeted
- Multiple versions might have to be maintained at each point in time
- Easy to lose track of which changes realised in which version
- Help is needed in managing versions and the processes that produce them.

Software Configuration Management to the rescue

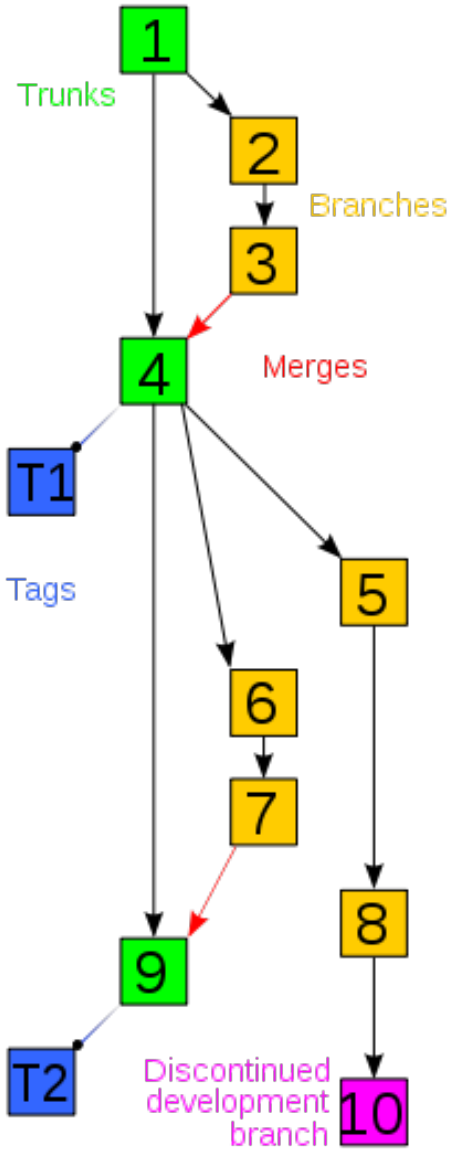
CM is all about providing such help.

Common CM activities:

- Version control
  - tracking multiple versions,
  - ensuring changes by multiple developers don't interfere
- System building
  - assembling program components, data and libraries,
  - compiling and linking to create executables
- Change management
  - tracking change requests,
  - estimating change difficulty and priority
  - scheduling changes
- Release management
  - preparing software for external release
  - tracking released versions

Focus on first two today

Version control



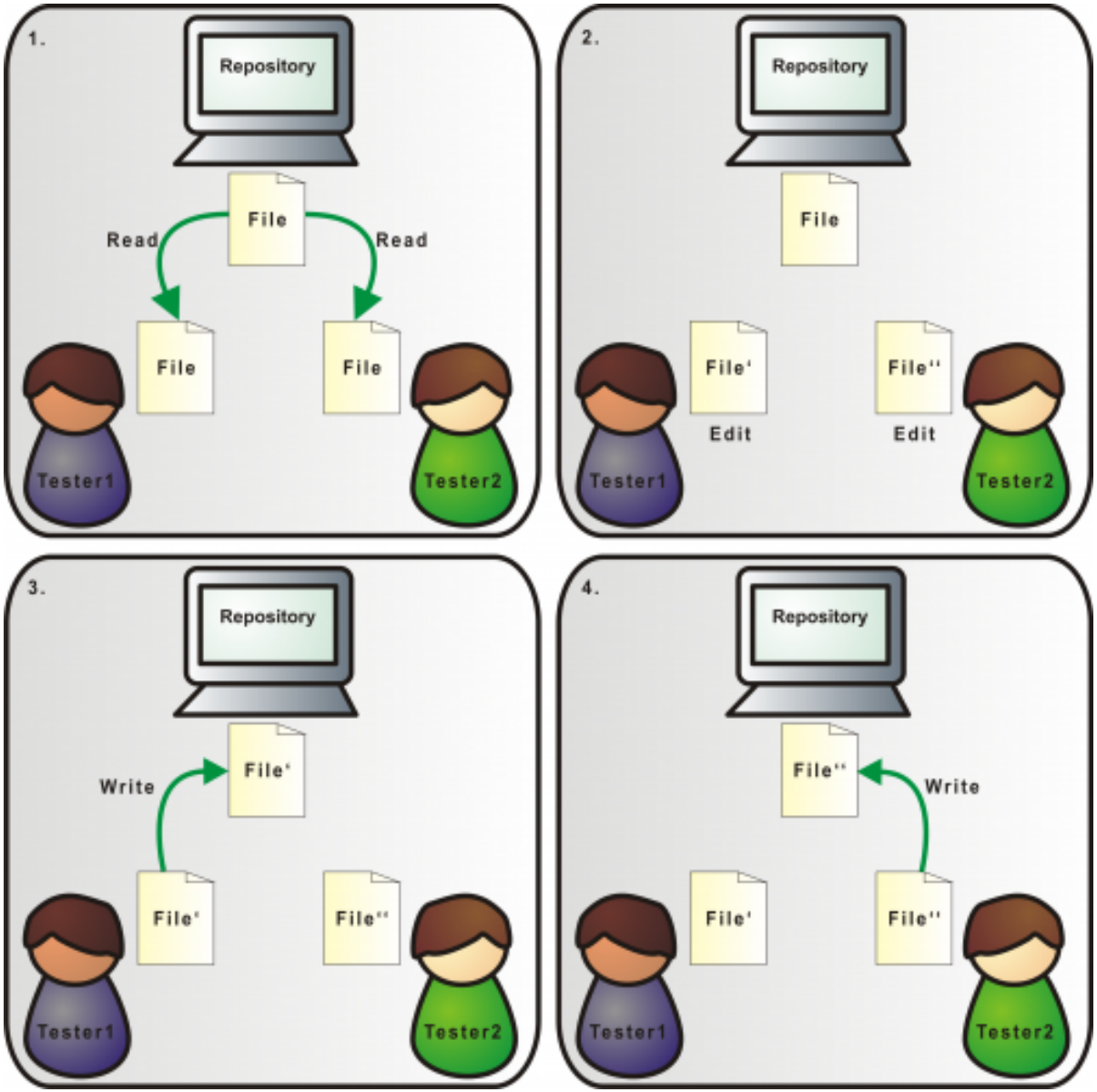
Version control

The core of configuration management.

The idea:

- keep copies of every version (every edit?) of files
- provide change logs
- somehow manage situation where several people want to edit the same file
- provide *diffs*/*deltas* between versions

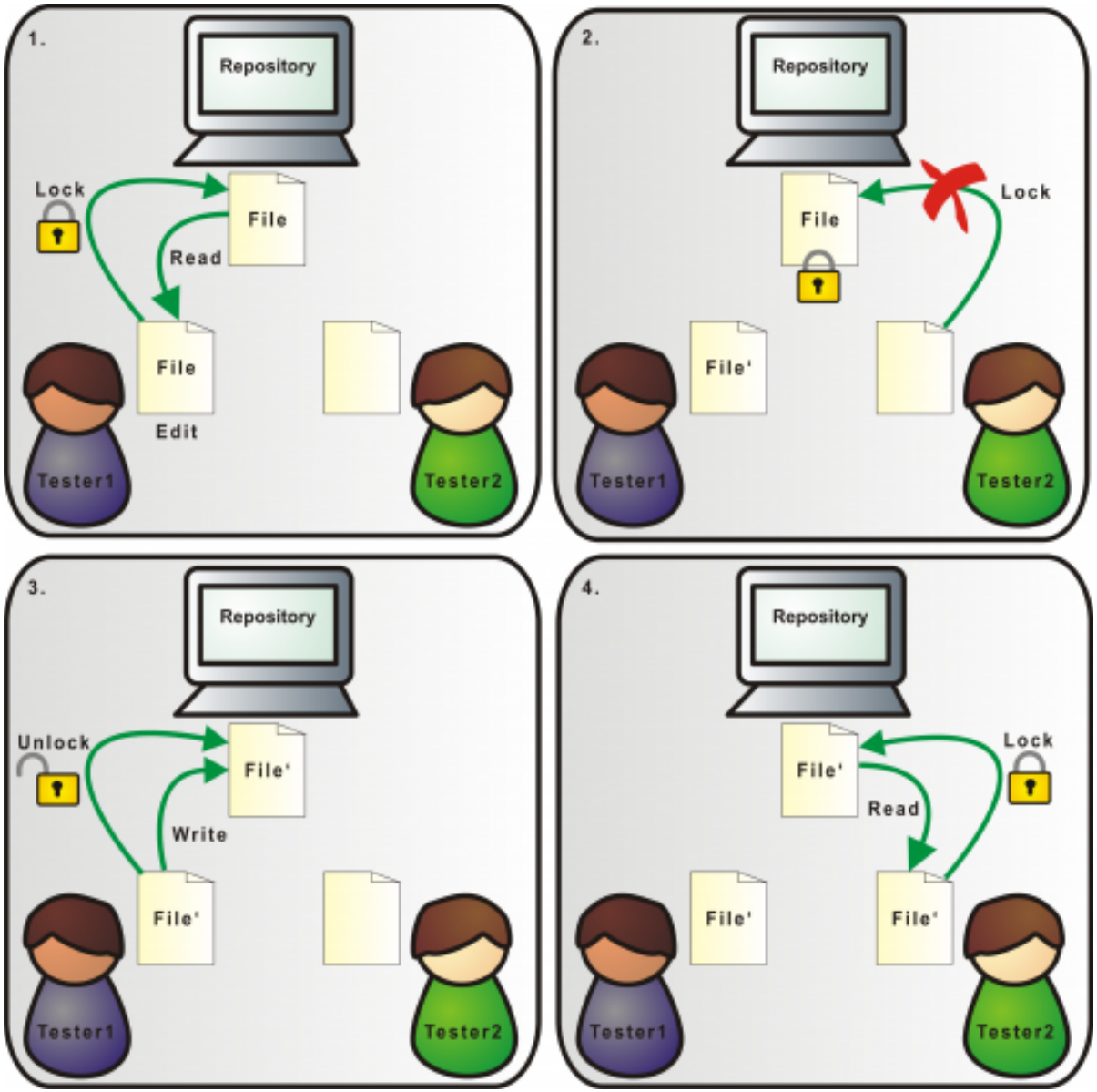
How file updates can be lost



Lock-Modify-Unlock Model

- Editor *checks-out* a file from a *repository*
  - Editor locks file
  - Others can check-out, but only for reading
- Editor makes changes
- Editor *checks-in* modified file to repository
  - Lock is released
  - Changes now viewable by others
  - Others now can make their own changes

Lock-Modify-Unlock Example



RCS

- Old, primitive VC system, much used on Unix.
- Uses Lock-Modify-Unlock model
- Keeps deltas between versions; can restore, compare, etc.
- Can manage multiple *branches* of development.
- Works on single files, not collection of files or directory hierarchies.
- Best suited for small projects, where only one person edits at a time.

CVS and SVN

CVS is a much richer system, (originally) based on RCS.
Subversion (SVN) is very similar, but newer.

Both handle entire directory hierarchies or projects – keep a single master *repository* for project.

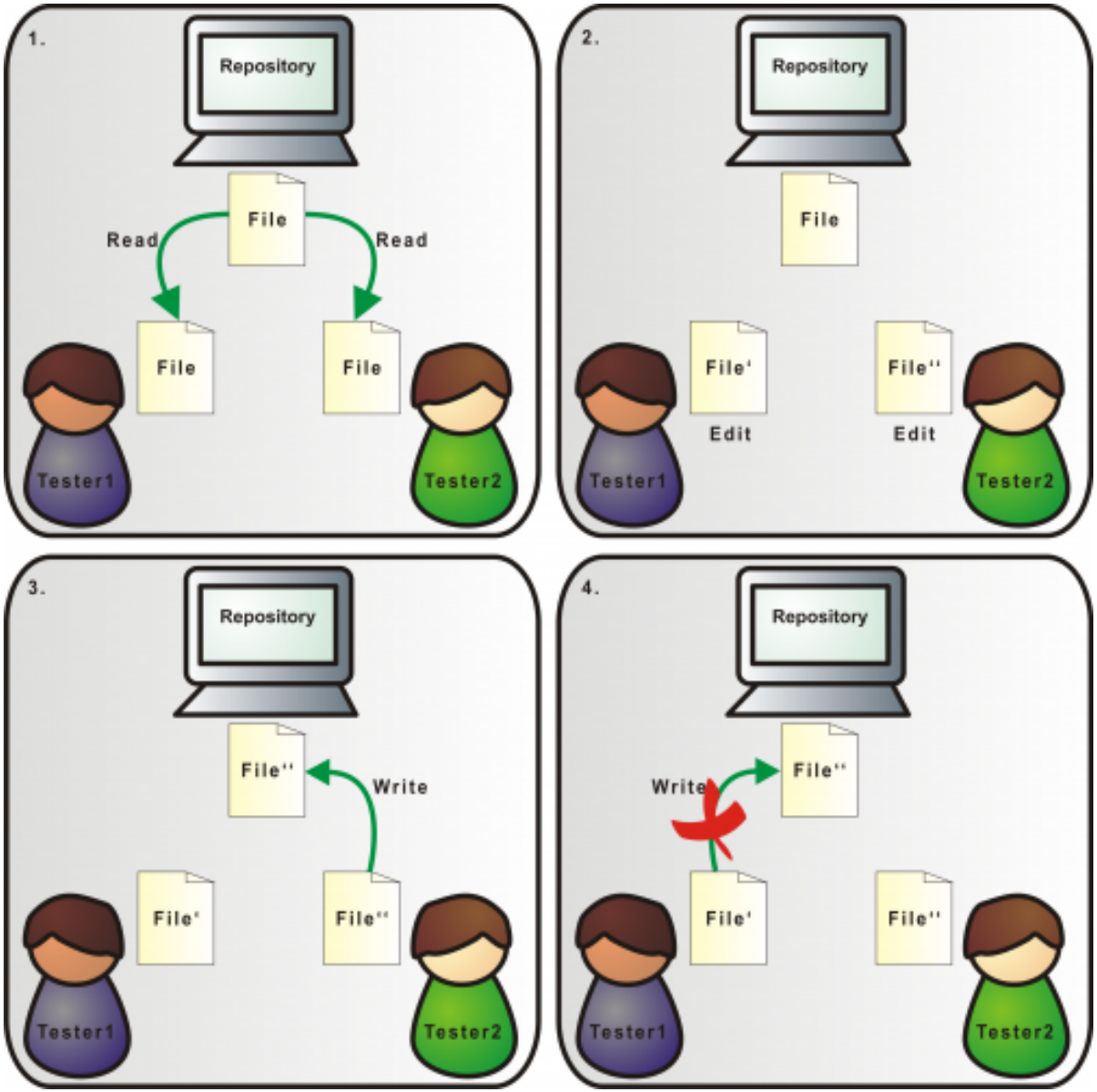
Designed for use by multiple developers working simultaneously – Copy-Modify-Merge model replaces Lock-Modify-Unlock.

Pattern of use for Copy-Modify-Merge:

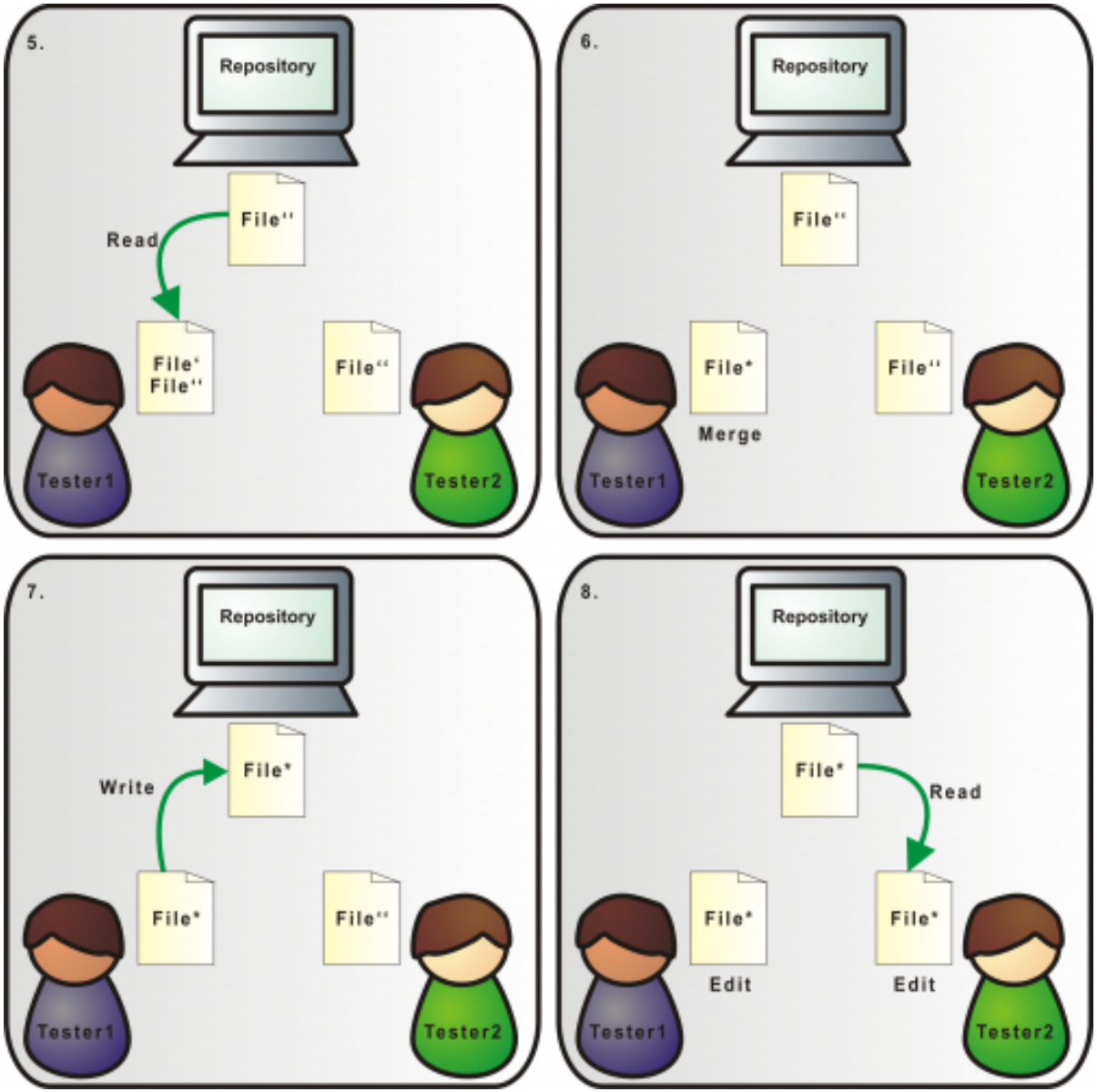
- *check out* entire project (or subdirectory) (not individual files).
- Edit files.
- Do *update* to get latest versions of everything from repository
  - system merges non-overlapping changes
  - user has to resolve overlapping changes - conflicts
- *check-in* version with merges and resolved conflicts

Central repository may be on local filesystem, or remote.

Copy-Modify-Merge



Copy-Modify-Merge



Distributed version control

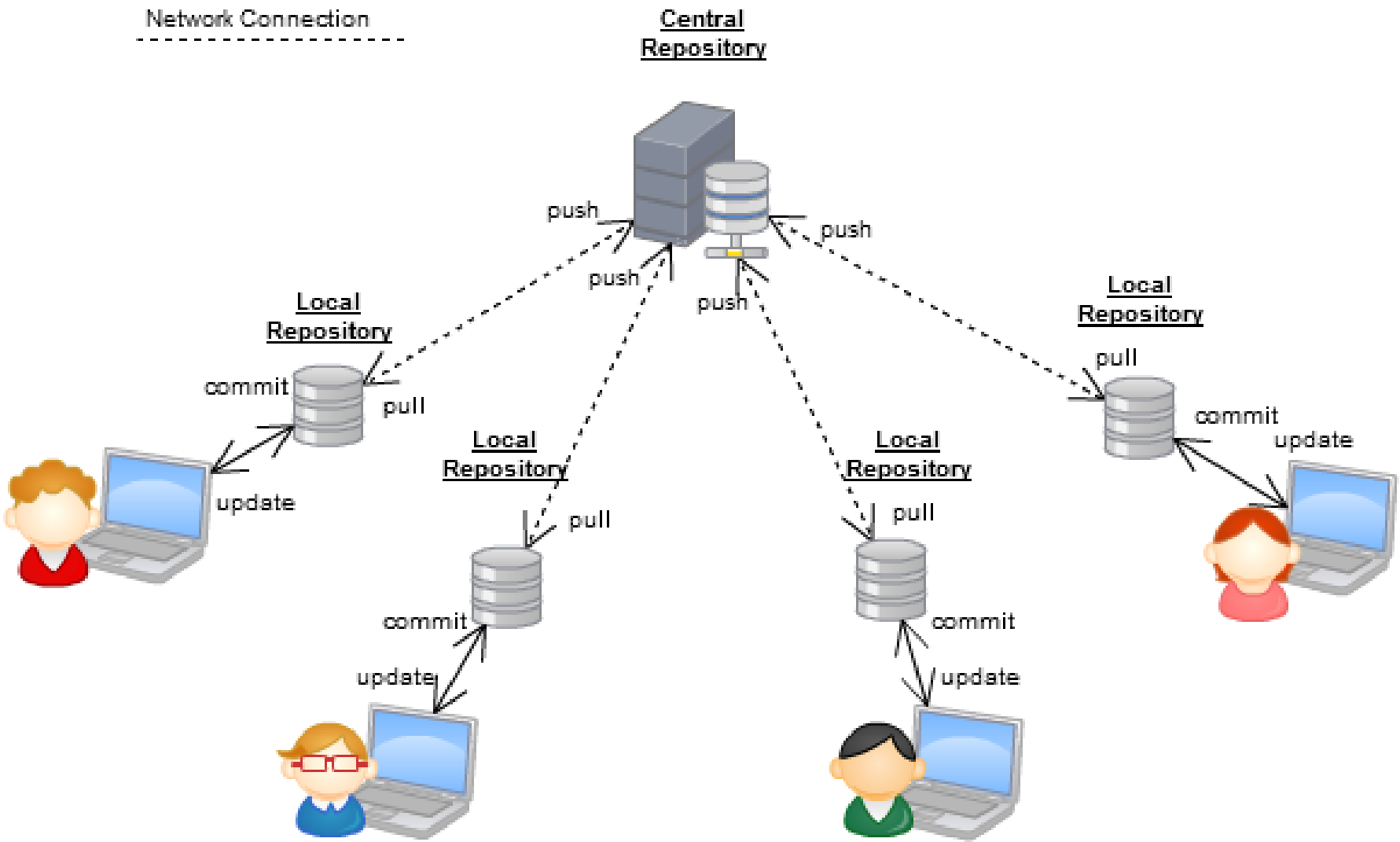
E.g. Git, Mercurial, Bazaar.

All the version control tools we've talked about so far use a single central repository: so, e.g., you cannot check changes in unless you can connect to its host, and have permission to check in.

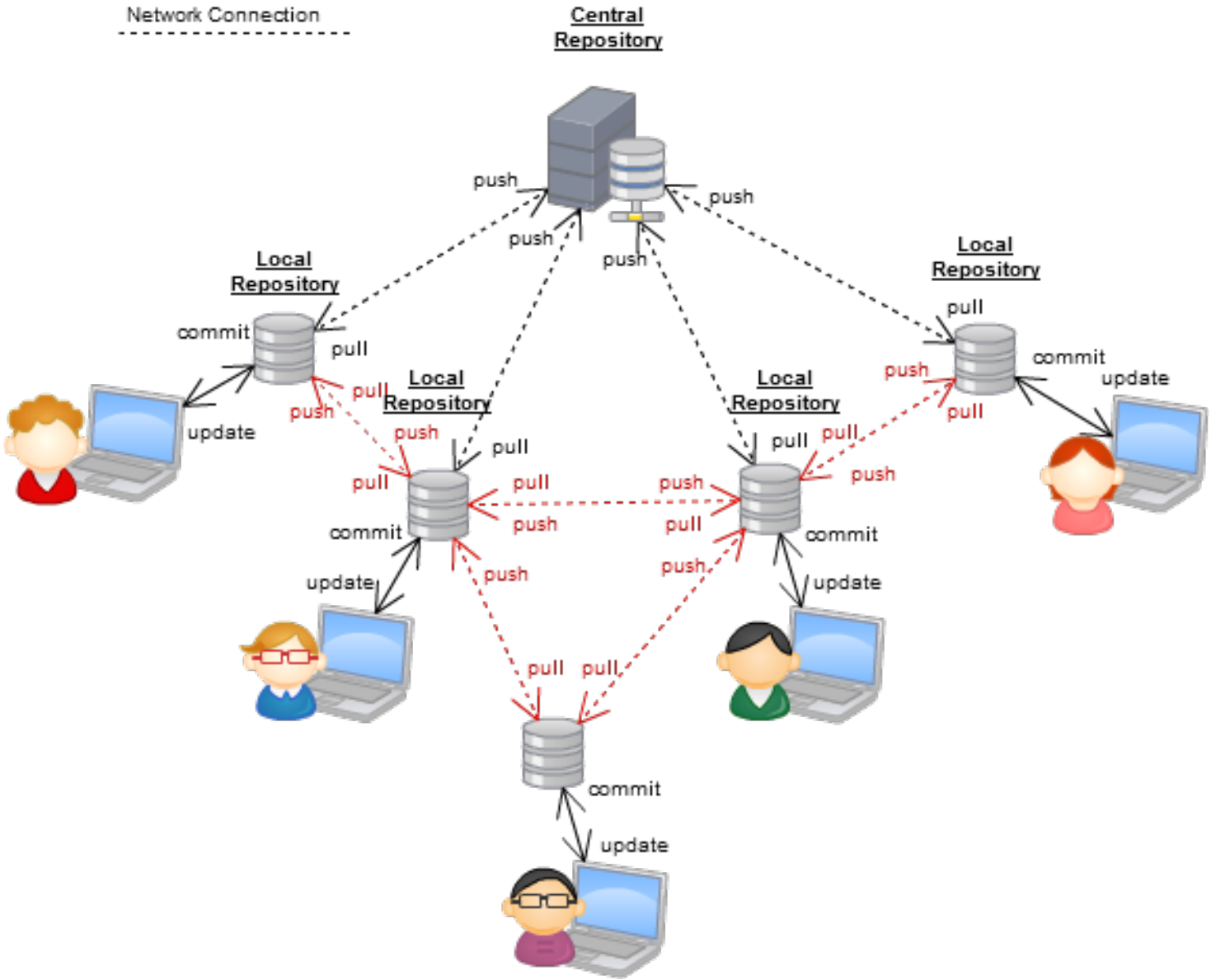
Distributed version control systems (dVCS) allow many repositories of the same software to be managed, merged, etc.

- reduces dependence on single physical node
- allows people to work (including check in, with log comments etc.) while disconnected
- much faster VC operations
- But... much more complicated and harder to understand

Distributed VCS



Distributed VCS



Branches

Simplest use of a VCS gives you a single linear sequence of versions of the software.

Sometimes it's essential to have, and modify, two versions of the same item and hence of the software: e.g., both

- the version customers are using, which needs bugfixes, and
- a new version which is under development

As the name suggests, branching supports this: you end up with a tree of versions.

What about merging branches, e.g., to roll bugfixes in with new development?

- In general need a *3-way-merge* between ends of two branches and a common ancestor
- Merge support good in Git, Mercurial and recent SVN versions
  - Developers with these VCSs use branches a lot more.

Build tools

Given a large program in many different files, classes, etc., how do you ensure that you recompile one piece of code when another than it depends on changes?

On Unix (and many other systems) the `make` command handles this, driven by a *Makefile*. Used for C, C++ and other 'traditional' languages (but not language dependent).

part of a Makefile for a C program

```
OBJS = ppmtomd.o mddata.o photocolcor.o vphotocolcor.o dyesubcolcor.o
ppmtomd: $(OBJS)
        $(CC) -o ppmtomd $(OBJS) $(LDLIBS) -lpnm -lppm -lpgm -lpbm -lm

ppmtomd.o: ppmtomd.c mddata.h
        $(CC) $(CDEBUGFLAGS) -W  -c ppmtomd.c

mddata.o: mddata.c mddata.h
```

Makefile rule structure:

target : *dependencies*
    command1
    command2
    ...

Running `make` *target* uses *commands* to

- create *target* from *dependencies* if it does not exist
- rebuild *target* when any of *dependencies* are newer.

Before creating / rebuilding *target*, `make` recursively considers whether any *dependencies* need creating or rebuilding.

Ant

`make` can be used for Java.

However, there is a pure Java build tool called Ant.

Ant *Buildfiles* (typically `build.xml`) are XML files, specifying the same kind of information as `make`.

part of an Ant buildfile for a Java program

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name ="Dizzy" default = "run" basedir=".">
<description>
    This is an Ant build script for the Dizzy chemical simulator. [...]
</description>
<!-- Main directories -->
<property name = "source"      location = "${basedir}/src"/> [...]
<!--General classpath for compilation and execution-->
<path id="base.classpath">
  <pathelement location = "${lib}/SBWCore.jar"/> [...]
</path> [...]
<target name = "run" description = "runs Dizzy"
                     depends =" compile, jar">
  <java classname="org.systemsbiology.chem.app.MainApp" fork="true">
    <classpath refid="run.classpath" />
    <arg value="." />
  </java>
</target> [...]
</project>
```

Maven

For making the building of Java projects simpler and more uniform.

Defines default support for

- compiling
- testing (using unit tests)
- packaging (e.g. as jar file)
- integration testing
- installing (e.g. into local repository)
- deploying (e.g. into release environment for sharing with other project)
- generating documentation

Per-platform code configuration

Different operating systems and different computers require code to be written differently. (Incompatible APIs. . .). Writing portable code in C (etc.) is hard.

Tools such as GNU Autoconf provide ways to automatically extract information about a system. The developer writes a (possibly complex) configuration file; the user just runs a shell script produced by autoconf.

(Canonical way to install Unix software from source: `./configure; make; make install`.)

A newer tool is CMake.

Problem is less severe with Java. (Why?) But still tricky to write code working with all Java dialects.

Reading

Required: Ch 1 of the SVN book
`http://svnbook.red-bean.com/`

Suggested: Mercurial tutorial `http://hginit.com/`

Suggested: Chs 1 & 2 of Pro Git book:
`https://git-scm.com/book/en/v2`