Chapter 2: FOSS project management (2.1-2.3)

Josep M. Ribó

28 de febrer de 2010

Chapter 2: FOSS project management

2.1 Getting started

Chapter 1 has been devoted to explain and discuss the FOSS development process.

Now, it is important to move to practical issues:

- What practical issues must be undertaken in order to start a FOSS project
- What tools are useful in order to develop a FOSS project

Practical issues to start a project

Some practical issues that must be undertaken when starting a new FOSS project:

- **Choose a good name**
  - Self-explicative
  - Easy to remember
  - A new name!!!
- **Convey a clear objective in the front page of the project site**

  Example: First sentence in the Apache web server page:

  The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows NT. The goal of this project is to provide a secure, efficient and extensible server that provides HTTP services in sync with the current HTTP standards.
- **State that it is a FOSS project**
  - Again in the front page

  - Choose a license and show it

- **Show the feature and requirement list**

  - Feature list:
    It outlines the functionalities that the application has (or is planned to have).
    State which are implemented and which are just planned
  - Requirements:
    Software and hardware requirements necessary to run the application

- **Development status page**

  Explain in a page:

  - What functionalities have been implemented
  - Which is the scheduled development plan of the project
  - Which are the topics that need more help
  - How many core developers are involved in the project
  - How often it puts out new releases
  - ...

- **Downloads**

  - The application should be downloadable, from the beginning, with building instructions
  - The building process should be standars and as much automatized as possible

  - When the project is mature, the binaries are also encouraged

- **Version control and bug tracker access**

  - The source code should be maintained under version control and made read-accessible to everybody in an anonymous way
  - A bug database and a way to submit bugs to it are a sign of project maturity
    The higher the number of bugs in the database, the better the project looks (many people are using the application and contributing with the project)

- **Communication issues**

  At least two mailing lists should be created (and archives maintained):

  - User list
  - Development list

  However, it is strongly recommended that at the begining of the project, only one list is maintained

- **Documentation issues**

  Documentation is absolutely crucial in order to gain users and contributors

Documentation should contain, at least:

- Explain in detail how to set up the software
- Basic explanation of how the software works
- Label the parts that contain incomplete documentation (and ask for contributions)
- A FAQ

Documentation should be on-line and in the downloadable distribution

Better to use an easy-to-edit format (text, html, XML...)

- **Developer documentation**
  - *Developer documentation:*
    Documentation to understand the code (it includes APIs)
  - *Developer guidelines:*
    Guidelines on how the developer should write code, patches and submit them to the project

2.2 Tools to manage a FOSS project

The tools that are necessary to develop a FOSS project in community (using a Bazaar-like model) are:

- The project repository (the forge)
- A Version control system
- A bug/event tracker
- A documentation tool
- Mailing lists

These tools constitute the **technical infrastructure** of the project

2.2 Tools to manage a FOSS project. The forge

See chapter 4

2.2 Tools to manage a FOSS project. The version control system

A version control (revision control) system is an application that keeps track and controls the changes to the source and documentation files of a project
The core of version control is **change management**:
To identify each single change made to a file and to annotate it with some information such as *change date* and *change author*

Version control vocabulary:

- **Revision**

  A specific instantiation of a file or directory

  A change on *file f rev. 6* yields *file f rev. 7*

  Sometimes the word *version* is used for *revision*. For example: *version control* instead of *revision control*

- **Repository**

  Database that contains the project and keeps track of all the revisions generated on all the files

  It does not store all the versions of a file but just the differences between them

- **Checkout**

  The process of obtaining a local copy of the project from the repository

  ```
  $ svn checkout http://svn.ex.com/repos/main mycopy
  ```

  In a FOSS project everybody should have right to checkout the project files from the repository

- **Working copy**

  A local copy of the project files stored in a developer's work space

  Some metadata is also stored in the working copy

  A checkout generates a working copy of the project files

- **Commit**

  To make a change on a file/directory of the repository from a developer's working copy

  ```
  $ svn commit
  ```

  A change committed to the repository produces a revision of a file

  In a FOSS project, (in general) only the core developers have commit access to the repository

- **Update**

  To incorporate in the local copy of a file/directory the changes that other developers have made on the copy of that file/directory in the repository

  ```
  $ svn update
  ```

  An update may generate conflicts

- **Conflict**

  A conflict arises if two developers modify the same file in an overlapping way.

  That is, if two developers update the same line of a specific file in two different ways

  Example:

  Consider the file (`m.txt`), at revision 1, which will ve updated by two users: `u1` and `u2`:

`m.txt` Revision 1	`m.txt` u1 revision	`m.txt` u2 revision
a	a	a
b	b	b
c	1	x
d	2	c
	d	d

A conflict arises **at line 3** of the original file since, at that line:

- User 1 has added:
  1
  2
- and has deleted `c`

...While user 2 has added `x` at the same line 3.

- **Diff/patch**
  - **Diff**
    A textual/graphic representation of the changes in revisions i of file `f` with respect of the revision j of the same file `f`
    Diff representations are usually written to files
  - **Patch**
    An application of the updates contained in a *diff file* into the file for which those updates were meant

Example: Consider the files m1.txt and m2.txt:

m1.txt	m2.txt
a	a
b	b
c	1
d	2
	d

`Diff` of `m2.txt` with respect to `m1.txt` is obtained by means of the following command:

```
$ diff -u m1.txt m2.txt >diff.txt
```

The file `diff.txt` contains:

```
--- m1.txt 2009-02-23 17:45:13.000000000 +0100
+++ m2.txt 2009-02-23 17:44:44.000000000 +0100
@@ -1,4 +1,5 @@
 a
 b
-c
+1
+2
 d
```

Which means that `m2.txt` has had 'c' ('-c') with respect to `m1.txt` and has had '1', '2' added ('+1', '+2')

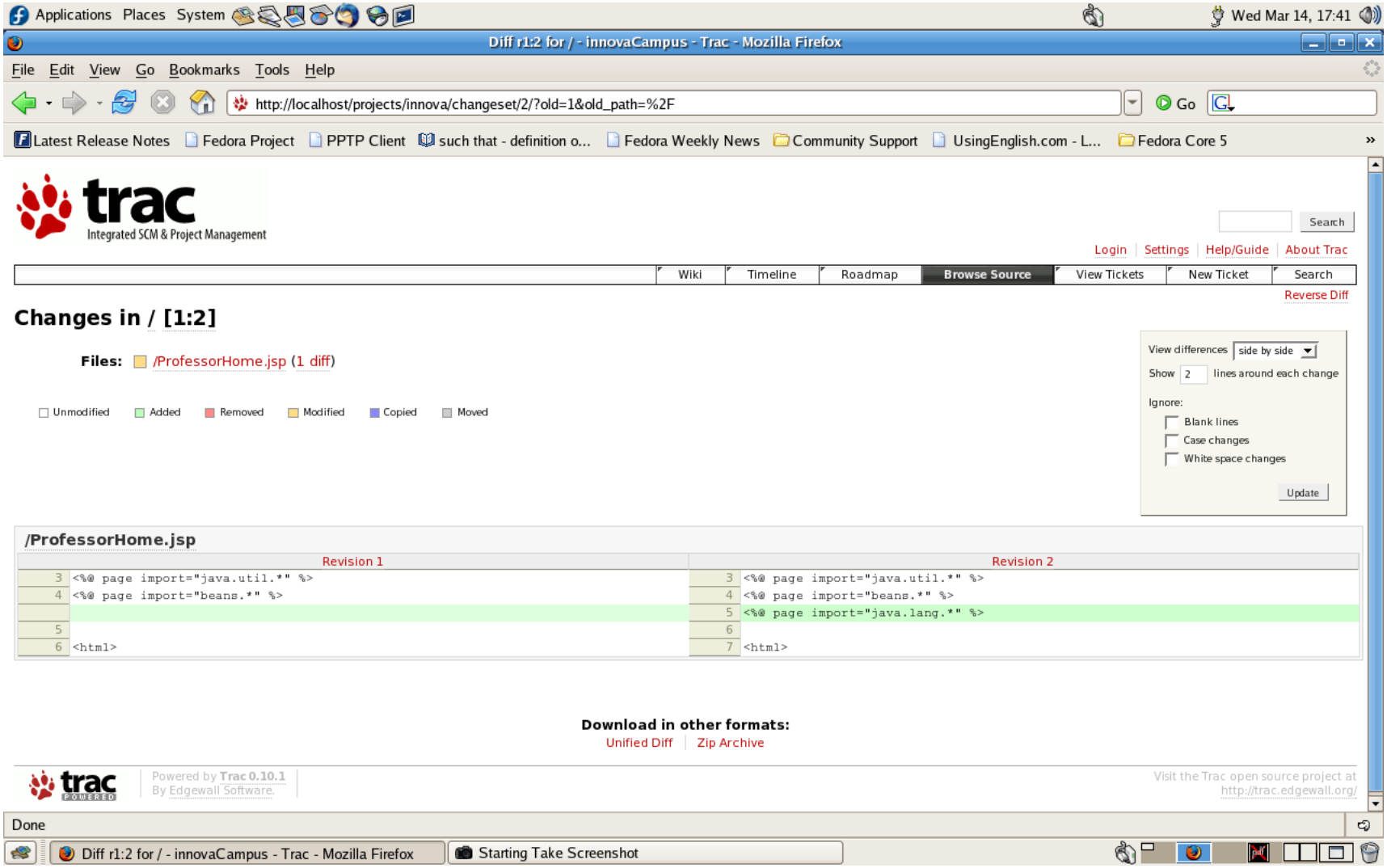
Now, it is possible to apply to m1.txt its differences with respect to m2.txt (i.e., those differences contained in diff.txt):

```
$ patch m1.txt diff.txt
```

The updates contained in `diff.txt` are applied to `m1.txt` and, as a result, both `m1.txt` and `m2.txt` are identical and contain:

```
a
b
1
2
d
```

Diff. Graphical representation



- **Tag**

  A label for a set of file revisions

  Usually, the set of file revisions that are tagged are those with a special relevance in the project (e.g., the files that constitute a specific release).

  Example of a tag: `Release_1.0`

  ```
  $ svn checkout http://svn.ex.com/repos/Release_1.0
  ```

- **Branch**

  An isolated copy of the project

  Changes made to that branch will not affect the rest of the project

- **Merge**

  To move a change from one branch to another

  Example: to move all the changes in one branch to the main trunk of the project

  A merge may generate conflicts if overlapping changes have been made to the files

A usual work session with a version control system

The basic work session with a version control system consists of the following steps:

1. **Download (check out) the files of the project repository** to get a local copy of them

2. **Update that local copy with local changes**

3. **Update that local copy with community changes**

   While a user was editing his/her local files, other users can have committed changes to the repository, so that, the local copy of the repository gets out-of-date

4. **Commit the updated local copy to the repository**

   That is, the local changes are made "official". They are made part of the repository and when some other user does a *checkout* of the repository he/she will find them.

In the next few slides we will show an example of how this cycle works

We use the specific version control system called `subversion` (****) to illustrate a typical work session

1. Getting the project source code

```
$ svn checkout repositoryURL
```

```
$ svn checkout http://sedna.udl.cat/innova
```

A local copy of the repository is obtained. Along with the copy, each directory contains version information in a subdirectory called `.svn`. This information is used by `subversion` to manage version control.

2. Updating the local copy with local changes

These local changes **are not uploaded/committed to the repository**. They are just local changes.

These local changes can be carried out in different ways:

- *Updating some local files with a text editor*
- *Deleting some files from the local copy*
  Existing files can be deleted:

```
$ svn delete fileName          (1)

$ svn delete DirectoryName     (2)
```

  - Command (1) is applied to file `fileName`
  - Command (2) is applied to the directory `DirectoryName` and, recursively, to all its contents

> Those files to which the command `svn delete` is applied and that were already committed in the repository are marked for deletion upon the next commit (see below) **but are not deleted from the repository yet!!!**.
> Files that were not committed in the repository are deleted immediately from the local copy.

- *Adding some files to the local copy*
  New files can be created (e.g., with the text editor)
  `$ svn add fileName`

  `$ svn add DirectoryName`

  When this command is issued those files to which the command `svn add` is applied and which have not been committed to the repository (see below) are marked for addition upon the next *commit* **but are not added to the repository yet!!!**

- *Copying some files.*
  `$ svn copy src dst`
  If `src` and `dst` are local paths, the files are **not** copied into the repository. The changes are scheduled to be made in the next commit

3. Getting information about the local changes

```
$ svn status
```

`svn status` informs on the changes made on the local copy of the repository:

- `M m.txt`: The file `m.txt` has been updated locally (since the last check-out from the repository
- `D m.txt`: The file `m.txt` has been scheduled for deletion from the repository
- `A m.txt`: The file `m.txt` has been scheduled for addition into the repository
- ...

Notice that the usual form of this command **does not contact the repository**

```
$ svn status --show-updates
```

Contacts the repository and shows which files have changed on the repository since the last check-out and, hence, are out-of-date in the local copy.

Example:

```
* m.txt
M    *   y.txt
```

- The file `m.txt` has changed in the repository
- The file `y.txt` has changed in the repository and has been modified locally

Clearly, the file `y.txt` is a potential source of conflicts

Those conflicts will occur if the local updates of `y.txt` overlap with the updates already committed to the repository

We will only be aware of those possible conflicts when a `svn update` command is issued (see next)

4. Updating the local copy with community changes

While a user `A` has been editing his/her local files, other users can have committed changes to the repository, so that, the `A`'s local copy of the repository gets out-of-date

```
$ svn update
```

> Brings the local copy of the repository up to date with respect to the changes that other users have been made to the repository

This means:

- Adding the **new files** that have been committed into the repository by other users after the checkout of the local copy

  `svn update` shows the added files preceded by an `A`

- Deleting the files that have been deleted in the repository since the checkout of the local copy

  `svn update` shows the deleted files preceded by a `D`

- Updating the files that have been updated into the repository

  `svn update` shows the updated files preceded by a `U`

However, in the process of:

- Deleting a local file that has been deleted in the repository or
- Updating a local file that has been updated in the repository

a **merging** or a **conflict** may come up:

- **A merging:**
  A file `a.cc` has been updated both in the local copy and in the repository. But both updates do not overlap. In this case svn update automatically merges both files in the local copy (of course, the copy committed in the repository is not altered).

  `svn update` shows the merged files preceded by a `G`

- **A conflict**
  A file `a.cc` has been updated both in the local copy and in the repository. Both updates DO overlap.

  `svn update` shows the conflicting files preceded by a `C`
  If this is the case, `svn update` can insert into the local file to be updated the `diff` betwen the local copy and the repository one

In the example shown in the **Conflicts** section, `svn update` would insert the following information into `m.txt`:

```
--- .svn/text-base/m.txt.svn-base Mon Feb 16 11:35:22 2009
+++ .svn/tmp/tempfile.4.tmp Mon Feb 16 11:36:19 2009
@@ -1,4 +1,10 @@
 a
 b
+<<<<<<< .mine
+x
 c
+=======
+1
+2
+>>>>>>> .r19
 d
```

This information shows the differences between user's local copy of m.txt (.mine) and the revision 19 of this file kept in the repository (.r19)

Files with a conflict status cannot be committed. Therefore, the user should indicate `subversion` that he/she has resolved the conflicts in a file before committing it. This is done with svn resolved:

```
$ svn resolved m.txt
```

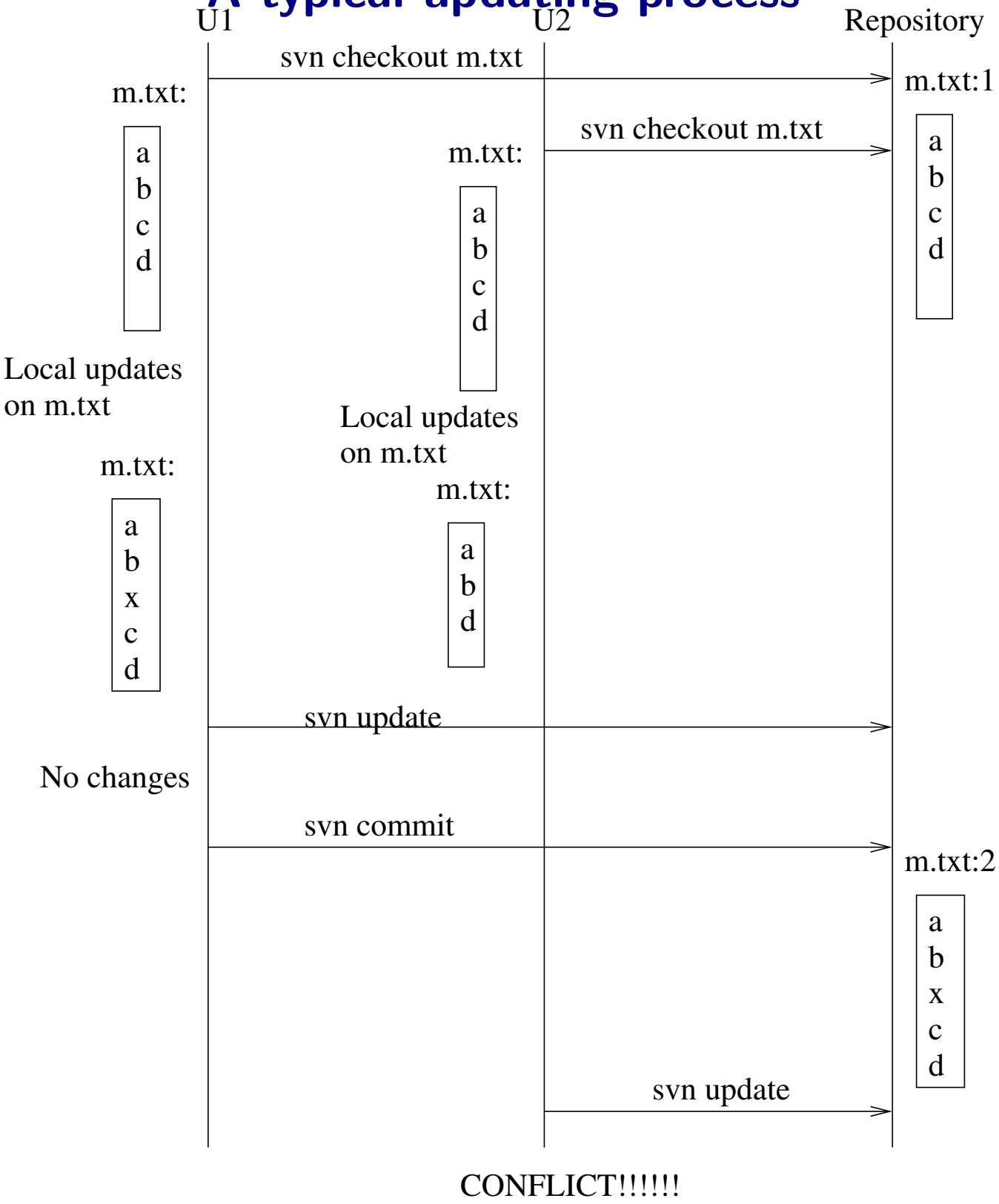
Clearly, before doing this, she/he should agree on the updatings to be applied with the involved users

5. Committing the local copy into the repository

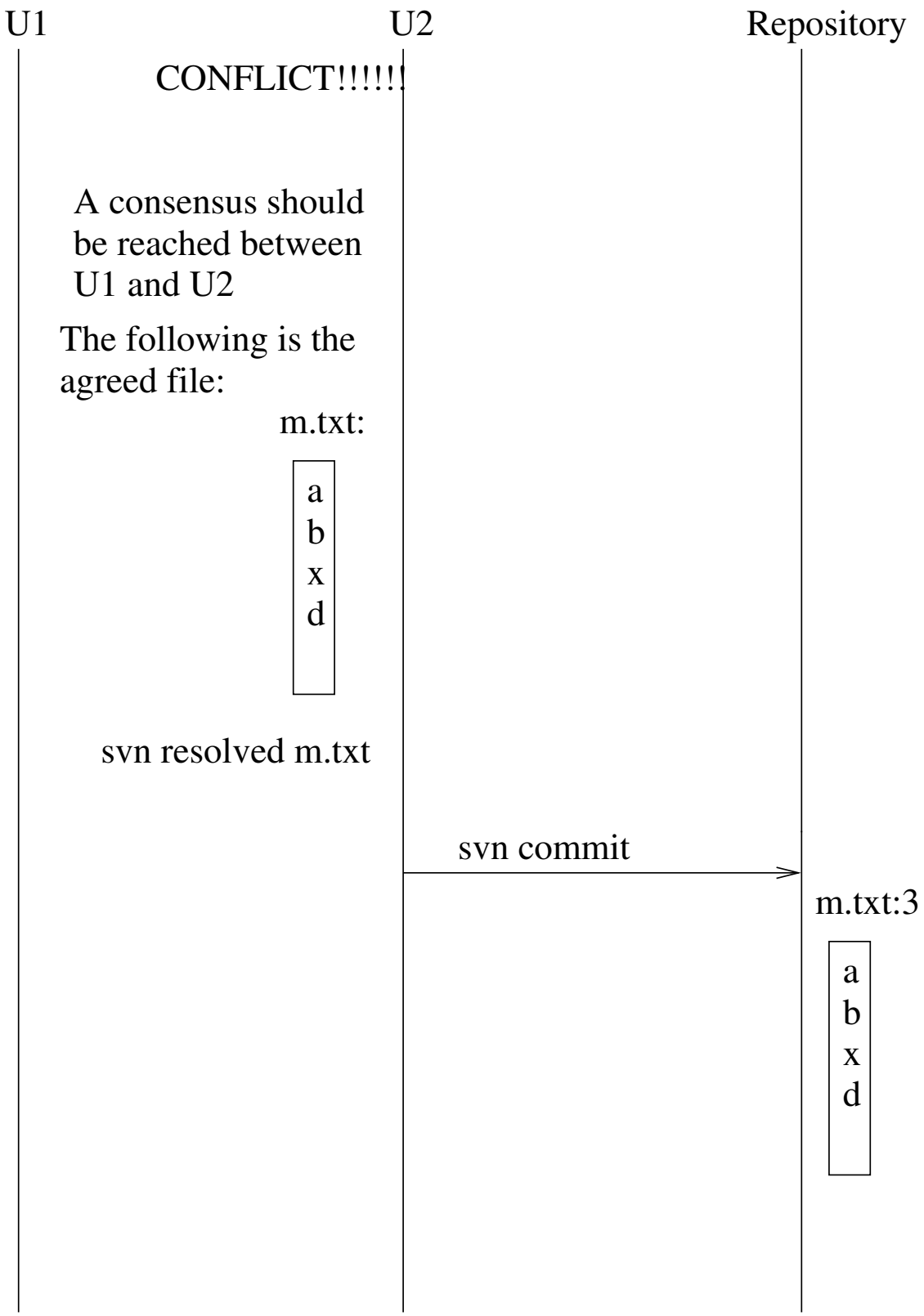
This can only be made if the local copy of the repository is up-to-date (i.e., nobody else has made any commit after the last svn update)

```
$ svn commit -m "minor changes in m.txt"
```

A typical updating process



....Continues ⟶


U1
U2
Repository
CONFLICT!!!!!!
A consensus should be reached between U1 and U2
The following is the agreed file:
m.txt:
a
b
x
d
svn resolved m.txt
svn commit
m.txt:3
a
b
x
d

Sending patches to a project

- A contributor that does not have write access to the version control repository can send a patch to developers mailing list in the following way:

  ```
  $ svn diff innova >innovaDiff.dif
  ```

  This command checks the differences between the local copy of the file/directory innova with respect to the corresponding committed copy (in the repository). Those differences are written in `innovaDif.dif`

- The contibutor sends `innovaDiff.dif` to the developers mailing list

- A core developer (with write access to the repository) reads the innovaDiff.dif file and:

  ```
  $ patch -i innovaDiff.dif
  ```

  Applies the changes contained in `innovaDiff.dif` into his/her own working copy of the repository

- The core developer commits the changes:

  ```
  $ svn commit -m ‘‘changes to....’’
  ```

Remark:

The patch may add new files (e.g., m.txt) or delete existing ones (e.g., m2.txt)

Therefore, before committing, it may be necessary to add/delete some files to/from the versioning system

```
$ svn add m.txt
```

```
$ svn delete m2.txt
```

However, keep in mind that different FOSS projects may have different patch policies

2.2 Tools to manage a FOSS project. The bug/event tracker

A **Bug tracker** is a piece of software responsible for keeping track of the bugs that the community has found to the system and its current statuts
Usually, bug trackers also keeps track of other issues such as new functionalities requests...
Sometimes, they are called **issue trackers, event trackers, ticket systems...**

The main component of a bug tracker is a **database** which stores all the bugs/issues along with *its state* and *some information about them*

Accessing this database can yield information on pending issues (i.e., issues/bugs that have not been solved yet)

In the next few slides we will use the term *issue* to refer to bugs, issues, tickets, events ...

Classic life-cycle of an issue:

1. The issue is filed

A community member (i.e., anybody) provides a issue description using the form provided by the bug tracker system

- If possible, this description should include a description on how to reproduce it
- The issue is filed in an *open state*
- It is not assigned to anybody yet
- This reference shows how to write good issue descriptions:
  `http://www.chiark.greenend.org.uk/ sgtatham/bugs.html`

Warnings:

- It may not be a bug but a system behaviour compatible with its specifications
  Before submitting a bug, be sure that it is really a bug
  If one of such *non-bugs* is submitted many times, the developers should think about redesign that specific part
- The issue description may be duplicated in the bug tracker database
- *Regressions*: Already fixed bugs are filed again

2. **The issue gets known by the community**

   The community of users and developers of the project reads the issue and enrich it with new comments

3. **The issue gets reproduced**

   If possible, the issue should provide *reproduction instructions*

   Some other members of the community (frequently, some developer) tries to reproduce the bug and confirms that it is real

4. **The issue is assigned to some developer**

   This developer takes the responsibility for fixing it. He/she *takes ownership* of the issue

   The issue priority is assigned at this stage

5. **The issue gets scheduled for resolution**

   The **release** by which the issue should be fixed is decided

   This step may be skiped

6. **The issue gets fixed**

2.2 Tools to manage a FOSS project. The mailing lists

Mailing lists constitute:

- The essential element for project communications
- *Medium of record* of the decisions that have been taken by the community concerning the project

Important remark:
It is mandatory that discussions concerning the project are public in some of the project mailing lists
No discussion should be kept privately between some project developers

Mailing lists should provide the following features:

- Both e-mail and web-based subscription
- Both digest-mode or message-by-message subscription

  Digest-mode: one e-mail daily containing the daily list activity
- Moderation features

  With the help of software to detect spam
- Header manipulation
- Archiving features

Mailing list software configuration:

Automatic setting of the *reply-to* header to reply to all the list members

Advantages:

- Encourages *public* discussions

Drawbacks:

- Members could send inadvertedly a confidential issue to all the list subscriptors

2.3 Releases. Numbering releases

There are several strategies to number releases but only one compulsory principle:
Be consistent!!!

- Release numbers are groups of digits separated by dots

  FooApplication 4.5.0

- 0.X.Y always goes before 1.0

- Sometimes a tag (Alpha, Beta) is added to the number

  They mean that in the future another release will come up with the same number without tag.

  FooApplication 4.5.0 (Alpha) FooApplication 4.5.0 (Beta) FooApplication 4.5.0 (RC1) FooApplication 4.5.0 (RC2) FooApplication 4.5.0 (RC3) FooApplication

  RC means Release candidate

Three-component system:

- Major number
- Minor number
- Micro number
- **Backward compatibility**

  The ability of a system to accept inputs intended for an earlier versions of itself

- **Forward compatibility**

  The ability of a system to accept inputs intended for a later versions of itself

  Uaually, forward compatibility can be lost when new features are added to the application

- **Changes in the micro number**

  Forward and backward compatibility

  Only bug fixes

  New features not introduced

- **Changes in the minor number**

  Backward compatibility

  Forward compatibility not guaranteed.

  New features but not too many new features at once

- **Changes in the major number**

  Forward and backward compatibility not guaranteed

  new featueres, possibly new features set

The even/odd strategy

Minor number:

odd: software unstable

even: software stable

Major and micro: same meaning as before. Not affected

Release branches

The project must stop at a given time in order to make a release:

- Decide what changes are mature enough to be incorporated into the release
- Fix minor issues to get a stable release
- Finish documentation

But how is it possible to do this if a FOSS project is in continuous evolution?

Solution: Create a release branch

A release branch is a branch in the version control system intended to isolate the release code from the normal development

Release branches are seldom used in traditional SE environments (everybody is working in the release)

If a FOSS project does not use release branches, the usual situation is that several developers are idle while others are working in the release.

Creating release branches

```
$ svn copy http://ex.proj.com/svn/trunk
           http://...../svn/branches/1.0.x
```

This creates a branch for the releases of the minor line 1.0

All the releases with changes in the micro number (micro releases) will be done in this branch

```
$ svn copy http://ex.proj.com/svn/branches/1.0.x
           http://...../svn/tags/1.0.0
```

Release candidates

Before an important release containing many changes it is usual to issue one or several release candidates, which will be tested by developers.

When it is sufficiently tested it will become the official release

fooAPplication-1.5.0-Beta1

fooAPplication-1.5.0-Beta2

fooAPplication-1.5.0-Beta3

fooAPplication-1.5.0

(RC1... instead of Beta1...) is also possible

Packaging

A package is an archive that contains the distribution of a project release

The canonical form of distribution of FOSS projects is as source code

If there is a binary package it is derived from a master source distribution

Standard of packaging:

- **Format:**

  Compressed tar format .tar.gz (.tgz) Tarballs

  Zip format (in windows)

  Rar format (in windows)

- **Name:**

  Application's name+release number+appropriate file suffix (e.g. tgz)

  `innovaCampus-1.2.0.tgz`

- **Directory structure:**

  The unpackage of `innovacampus-1.2.0.tgz` should create a directory called `innovacampus-1.2.0` with the following internal structure:

  - The source code arranged in a shape ready for compilation and installation
  - README file explining the purpose of the software and its release number. In plain text!!!
    README should have pointers to the following files:
  - INSTALL file with the installation instructions
  - COPYING or LICENSE file with the project license
  - CHANGES or NEWS file with the new issues in the current release
    These *new issues* include:
    * List of new features/enhancements
    * List of bugs that have been fixed

Every source file should contain, at the beginning, a notice like this one:

```
/*************************************************
APPLICATION'S NAME

Copyright (C) YEAR AUTHOR'S NAME

This program is distributed under the terms of the GNU General Public License

APPLICATION'S NAME is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

APPLICATION'S NAME is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with InnovaCampus; if not, write to the Free Software
Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA


*************************************************/
```

There should be a copyright line for each piece of software used

Installation:

Standard installation is preferred.

These are examples of standard installation in different languages:

- C/C++ languages:

  ```
  $ ./configure
  $ make
  # make install
  ```

  configure: Detects as much of the environment as it can

  make: builds the software in place

  make install: installs it

- Java

  ```
  $ ant
  ```

- Perl

  ```
  $ perl Makefile.pl
  ```