Update Propagation Practices in Highly Reusable Open Source

OSS 2008
Milan, Italy

Heikki Orsila[1], Jaco Geldenhuys[2], Anna Ruokonen[1], and Imed Hammouda[1]

[1] Tampere University of Technology, Finland
[2] Stellenbosch University, South Africa

Outline

- Introduction
- Research methodology
- The zlib case
- The FFmpeg case
- Guidelines for Managing Updates
- Conclusions

Introduction

- More and more software developers and companies are basing their software products on open source components (i.e., libraries, platforms)
  - Shorter development cycles
  - Lower development costs
  - Access to source code
  - Improved product quality
  - …

- Risks:
  - Quality attributes such as reliability, security, and safety are hidden properties → Fixing can never be guaranteed
  - Many advocated hypotheses made about open source software are not always true.

Introduction

- Possible solution: regularly update to newer versions of the used open source components, which leads to faster incorporation of community contributions such as bug fixes and new component features.

- Basic usage pattern: whenever a new version of a component is released, users of that component immediately switch to the new release.

- One might hypothesize that most practices will eventually deviate from this basic principle due to various influential factors.

Reuse of Open Source components

- A. Always part of source: the component is incorporated during development time (e.g., the Linux kernel)
- B. Added when released: the component is incorporated during release time (e.g., xvidcap project)
- C. User must provide source: the component source code is incorporated by the user when the project is recompiled (e.g., eCos tool chain)
- D. User must provide binary: the component binary is provided by the user when the project is linked (e.g., OpenSSH)

Research Methodology

- Research Questions:
  - What reuse mechanisms are adopted most often when reusing open source components?
  - What kind of update propagation patterns are practiced?
  - How fast/often does the user community react to new releases?
  - What technical and non-technical criteria influence the community response?
  - What best practices can be identified to promote better follow-up of updates and smoother update propagation?
- Selecting suitable component candidates:
  - zlib: a lossless compression library
  - FFmpeg: a collection of utilities for processing audio and video files and streams
- Extracting relevant data: bug reports, revision history, source code
- Analyzing the data w.r.t the research questions
- Making recommendations

The zlib case

- Three security bugs:
  - □ *A* double free bug reported on 2002-03-11
  - □ *A* DoS/crash bug reported on 2004-08-25
  - □ *A buffer overrun/DoS/crash* bug reported on 2005-06-30

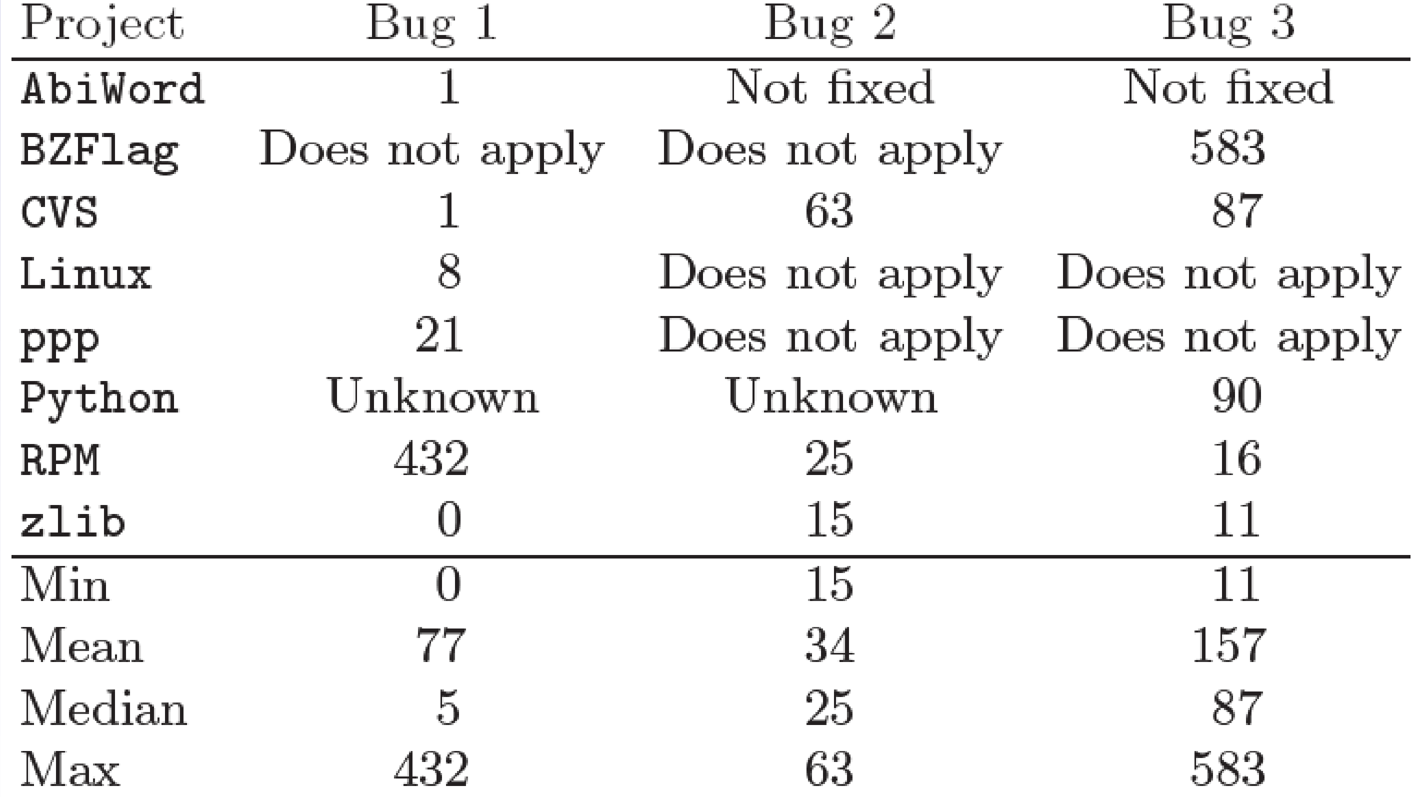
- 8 projects: AbiWord, BZFlag, CVS, Linux, ppp, Python, RPM, zlib

- Evolution: 11-04-1995 to 18-07-2005
  - □ 2 core authors, 42 contributors
  - □ 628 documented changes
  - □ 89% changes from the top 5 contributors

The zlib case

- Bug status in the projects:
  - Does not apply: The bug doesn’t have an effect on the project, because the vulnerable code never existed inside the project (e.g., Linux kernel)
  - Known: The time (in days) to fix a bug is known from version history (e.g., CVS)
  - Not fixed: The bug is still not fixed (e.g., AbiWord for Windows)
  - Unknown: Status of the fix is unknown due to unavailability of version history (e.g., Python)

The zlib case



Project	Bug 1	Bug 2	Bug 3
AbiWord	1	Not fixed	Not fixed
BZFlag	Does not apply	Does not apply	583
CVS	1	63	87
Linux	8	Does not apply	Does not apply
ppp	21	Does not apply	Does not apply
Python	Unknown	Unknown	90
RPM	432	25	16
zlib	0	15	11
Min	0	15	11
Mean	77	34	157
Median	5	25	87
Max	432	63	583

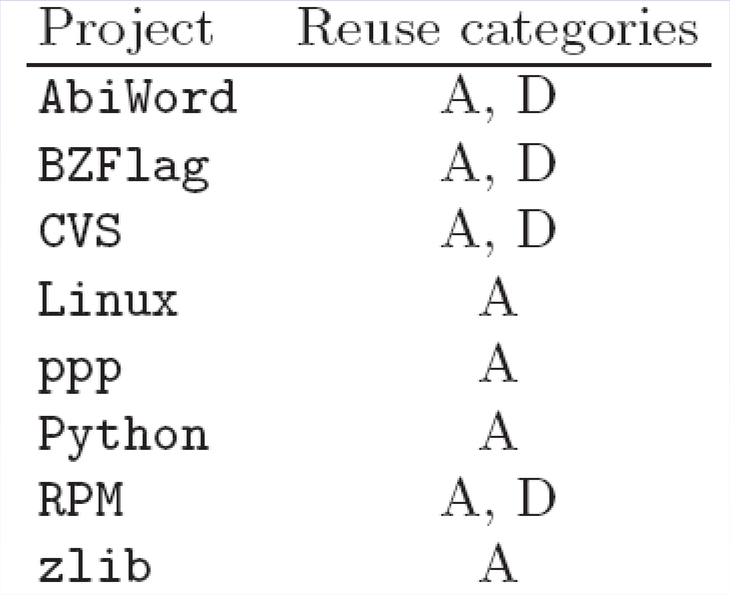
Number of days to fix 3 different zlib bugs

The zlib case

- Only 1 system for explicit checking for updates

- Possible reasons for this lapse:
  - □ Weak virtual organization
  - □ Lack of explicit task lists
  - □ Lack of command hierarchy
  - □ Lack of resources for testing new versions of zlib

The zlib case

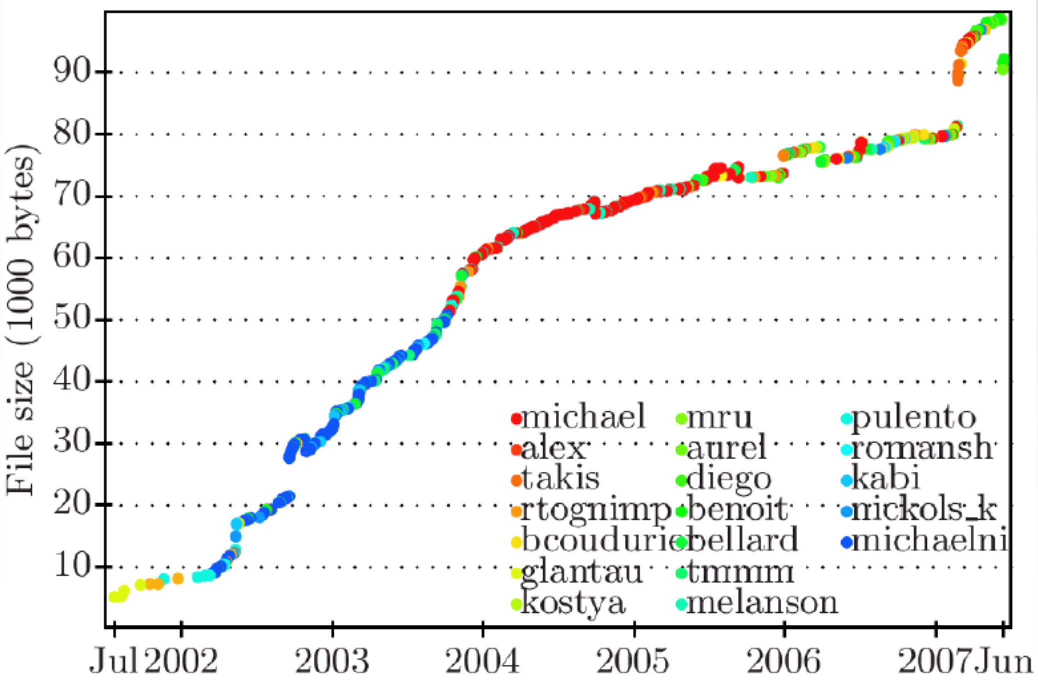


Project	Reuse categories
AbiWord	A, D
BZFlag	A, D
CVS	A, D
Linux	A
ppp	A
Python	A
RPM	A, D
zlib	A

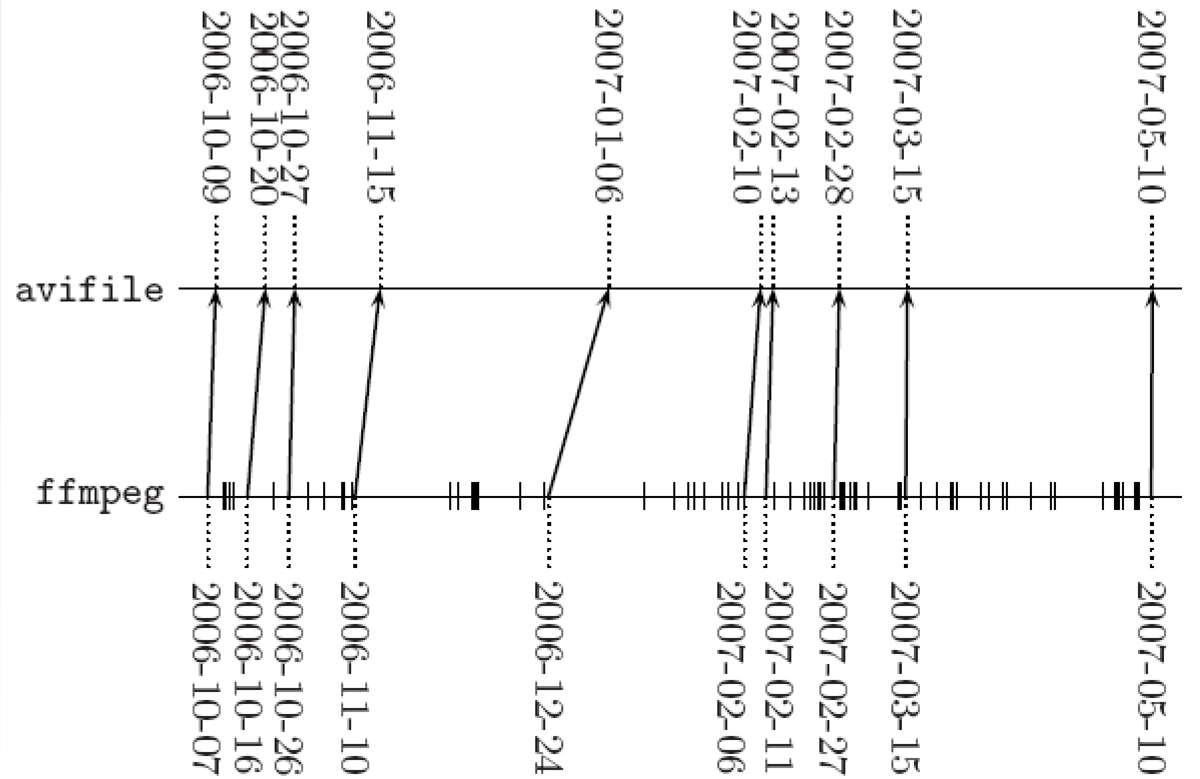
Projects and their reuse categories

The FFmpeg case

- A core library called libavcodec
- A library interface specification in the header file avcodec.h
- 6 projects: avidemux, avifile, ffdshow, gstreamer, mythtv, xbmc
- Evolution: 07-2001 to 06-2007:
  - □ 38 contributors
  - □ 617 changes
  - □ From 177 (5.1 kbytes) to 2940 (90 kbytes) lines of code



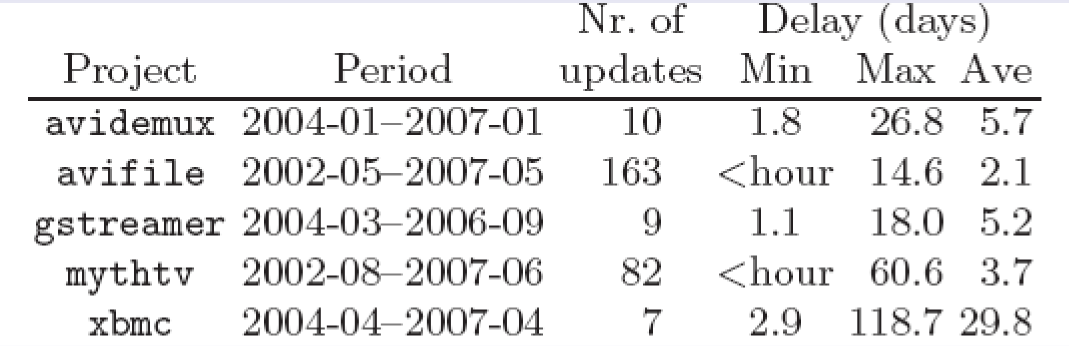
The FFmpeg case



The 10 most recent updates (from 2006-10-09 to2007-05-10) of avcodec.h in avifile

The FFmpeg case

- Shared interests, features and developers
- Update propagation entails significant effort
- Most projects fall into reuse category A, few go for option B



		Nr. of	Delay (days)		
Project	Period	updates	Min	Max	Ave
avidemux	2004-01–2007-01	10	1.8	26.8	5.7
avifile	2002-05–2007-05	163	<hour	14.6	2.1
gstreamer	2004-03–2006-09	9	1.1	18.0	5.2
mythtv	2002-08–2007-06	82	<hour	60.6	3.7
xbmc	2004-04–2007-04	7	2.9	118.7	29.8

Summary of update data for FFmpeg

Guidelines for Managing Updates

- Avoid source and binary code duplication!
- Document important changes in version control history!
- Tag important changes in version control history!
- For components: maintain a global notification system for changes!
- For projects: facilitate follow-up of component updates!
- Write a procedure for the update process!

Conclusions

- We have analyzed update propagation practices in zlib and FFmpeg.
- Scripts/results/experiences are found online.
- We have found that update propagation delay varies significantly among projects.
- We cannot claim that the results are generalizable.
- For further investigation, more case studies should be considered.
- In order to validate the relevance of the proposed guidelines, a questionnaire to the open source community could be planned and carried out.

Thank You!

Q&A