Update Propagation Practices in Highly Reusable Open Source Components

Heikki Orsila[1], Jaco Geldenhuys[2], Anna Ruokonen[3], and Imed Hammouda[3]

[1] Department of Computer Systems, Tampere University of Technology,
PO Box 553, FI-33101 Tampere, Finland, heikki.orsila@tut.fi
[2] Department of Computer Science, Stellenbosch University, Private Bag X1,
7602 Matieland, South Africa, jaco@cs.sun.ac.za
[3] Department of Software Systems, Tampere University of Technology, PO Box 553,
FI-33101 Tampere, Finland, firstname.lastname@tut.fi

Abstract. In today's business and software arena, more and more companies are adopting open source software. An example of this rising phenomenon is to base software products on highly reusable open source components. In this scenario, the evolution of the software product is coupled with the evolution of the open source component. A common assumption is that component updates are immediately and regularly propagated to the project. This paper investigates this assumption empirically by studying update propagation practices in two popular open source libraries, `zlib` and `FFmpeg`. For each library, we analyze various repository sources with information such as bug reports, revision history, and source code. The results of the case studies suggest that update propagation is subject to several technical and non-technical factors including the way the open source library is used, the extent to which updates are documented, and the degree of community involvement. Based on these findings, we propose a set of recommendations that would allow better follow-up of updates and smoother update propagation.

1 Introduction

Driven by various business and technical motives such as shorter development cycles, lower development costs, improved product quality, and access to source code, more and more software developers and companies are basing their software products on open source components (i.e., libraries, platforms, etc.) [1, 2]. Adopting open source software is sometimes considered a risky business strategy, mainly because of a lack of trust in community-driven software. The main concerns are that many quality attributes such as reliability, security, and safety are hidden properties that have to be carefully checked, and that fixing software defects pertaining to such quality attributes can never be guaranteed. Furthermore, empirical research shows that many advocated hypotheses made about open source software are not always true [3].

Please use the following format when citing this chapter:

Orsila, H., Geldenhuys, J., Ruokonen, A. and Hammouda, I., 2008, in IFIP International Federation for Information Processing, Volume 275; *Open Source Development, Communities and Quality*; Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, Giancarlo Succi; (Boston: Springer), pp. 159–170.

A basic practice to overcome part of the challenges sketched above is to 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. Thus, in the case of highly reusable components, one expects the following basic usage pattern: whenever a new version of a component is released, users of that component immediately switch to the new release. At the other end, one might hypothesize that most practices will eventually deviate from this basic principle due to various influential factors.

Based on the observations above, our research problem can be formulated as analyzing update propagation practices in the case of highly reusable open source components. In particular, we are interested in issues like the frequency of update propagations, the kind of interactions between the components and the projects using them, and the influential factors shaping these interactions. Our goal is ultimately to identify a set of guidelines that would promote better follow-up of updates and smoother update propagation. The good news is that open source projects come with a rich repository of models, source code, resource files, defect reports, change logs, etc., which makes it possible to mine such information.

To investigate our research problem, we focused on two popular software libraries: `FFmpeg` [5] and `zlib` [4]. We carried out our study empirically by analyzing the software repositories of these two libraries for update propagation. The updates concern bug fixes or new features contributed by the community. As expected, mining the information needed was not an easy task as we had to consider various repository sources such as bug reports, revision history and the source code itself. The results of the study show that practices vary from one case to another. In most of the cases, however, we were able to find answers to our research questions and make "educated guesses" for the reasons of the results. Our findings suggest that there are several technical and non-technical factors that have a direct effect on update propagation. These include the way the open source library is being used, the extent of documenting updates, and the degree of involvement in the community.

The rest of the paper is organized as follows. Section 2 presents the background of this work and discusses related studies. Sections 3 and 4 introduce the case studies and empirically explore our research questions. In Section 5 we present a set of recommendations supporting update propagation, and in Section 6 we conclude the paper.

2 Background

The repository sources most relevant to our empirical study are bug reports and revision logs. Most open source projects include an open bug repository, to which users of the software have full access. It is used to report and track bugs and potential enhancements. An open bug repository might potentially increase the number of problems identified in the system and enable more efficient fixing of problems. Bug reporting, resolving bug reports, and improving bug management have been discussed before [6, 7]. Although many open source software developers interact with the bug repository on a regular basis, there is little data available to characterize their interactions. Similarly, revision logs are useful sources in the sense that they record the evolution of an open source project, but they often come with challenges such as insufficient and unreliable data.

In open source projects, code contribution and bug fixing can be regarded as alternating phases in a continuous, cyclical process. Maintainability has been identified as the core quality issue in open source development [7]: developing an OSS system implies a series of frequent maintenance efforts mainly for debugging existing functionality and adding new features to the system. Maintenance activities can be categorized into four classes: adaptive (e.g., supporting new platforms), corrective (e.g., fixing bugs), perfective (e.g., improving quality attributes), and preventive (e.g., code cleanup and refactoring) [8]. According to this view, open source maintenance is mainly adaptive and corrective. However, in this paper we are more interested in how rapidly the user community reacts to maintenance updates, and in what motivates their reactions.

Reuse of open source software has been the subject of many studies. For example, Capiluppi et al. propose guidelines to identify highly reusable components and to improve the reusability of open source components [9]. Large-scale reuse involving open source repositories has also been studied by Mockus [10]. He identifies widely reused code blocks, typically a component, and common patterns of reuse. In this empircal study the focus is mainly on the immediacy, frequency, usage patterns, and underlying motivation of updates.

In the case of open source components, reuse can be divided (roughly) into the following practices:

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 [11])
D. User must provide binary: the component binary is provided by the user when the project is linked (e.g., OpenSSH [12])

In special cases, the reuse may even be a combination of two or more of the above (e.g., AbiWord [13]). Reuse of binary distribution makes use of either static or dynamic linking. In static linking the component binary is included in a local, stand-alone copy of the project at compile-time. Dynamic linking means that the component binary is loaded at runtime, and can therefore be updated independently of the project that is using it. In this respect dynamic linking is not relevant to the questions addressed in this paper.

Our research methodology can be summarized in five main steps: we 1) formulate the research questions, 2) select suitable component candidates 3) extract the relevant data by exploring the component repositories, 4) analyze the data with respect to the questions raised, and 5) make recommendations. The research questions we explore include the following:

- 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 of highly reusable open source components?
- What technical and non-technical criteria influence the community response (e.g., reuse mechanism, product domain, product development phase, etc.)?
- What best practices can be identified to promote better follow-up of updates and smoother update propagation?

As candidate components, we have selected two highly reusable libraries, `zlib` and `FFmpeg`. The former is a lossless compression library which implements a standard coding system [14, 15, 16]; it is used in many file formats and protocols, and in many popular systems such as Linux and Python. The latter is a collection of utilities for processing audio and video files and streams. It includes tools to play and record different media, and a server for distributing media over the internet, for example, for live broadcasts. The library has been incorporated in more than 90 projects.

3 Case Study: `zlib`

3.1 Analysis

The `zlib` source code is included in numerous projects. We looked at three security-related bugs that were found in the `zlib` source code, and analyzed the time it took the bug fix to propagate into 8 projects: `AbiWord`, `BZFlag`, `CVS`, `Linux`, `ppp`, `Python`, `RPM`, and `zlib`.

There are only two core authors for the `zlib` project, but 42 authors have contributed code to the library. Of the 628 documented changes, 89% come from the top 5 out of 42 contributors. This information comes from the credits in the library's `ChangeLog` file. We studied the latest `zlib`, version 1.2.3 released on 2005-07-18. The `Changelog` entries are dated 1995-04-11 to 2005-07-18, a period of approximately 10 years.

We investigated fixes for the following bugs:

1. *A double free bug* reported on 2002-03-11
2. *A DoS/crash bug* reported on 2004-08-25
3. *A buffer overrun/DoS/crash bug* reported on 2005-06-30

For each of the projects that use `zlib`, the bug status was classified as follows:

- **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 results are shown in Table 1. The mean and median times for fixing a bug, computed over all the projects in the table, are 97 and 19 days, respectively.

Table 1. Number of days to fix 3 different `zlib` bugs

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

3.2 Discussion

We noticed two issues from the `zlib` results:

Bug Fix Delay Varies Significantly The time to fix bugs varies a lot from project to project, which means that the median and mean times to fix bugs are far apart. In one case (`AbiWord`) the project is still vulnerable for bugs that were discovered years ago.

We did not find any project, apart from one operating system distributor, with an explicit system for checking for updates in other projects. Such an automatic mechanism is a necessity for scalable code reuse. Possible reasons for this lapse may be weak virtual organization, lack of explicit task lists, and lack of command hierarchy. Another possible reason that stabilized products fail to incorporate `zlib` updates is that their maintainers do not have the resources for testing a new `zlib` version and/or backporting necessary fixes.

Investigated projects seem to fall into three update propagation patterns: random updates, negligence, and systematic. We address these issues further in the guidelines in Section 5.

Microsoft Windows Programs Are Biased Towards Binary And Source Duplication Table 2 shows each project and its reuse category. Almost all GNU/Linux projects use zlib as a dynamic library, which can be updated through a common packet manager for all applications. Unfortunately the Microsoft Windows operating system lacks a common packet manager for non-Microsoft products. This suggests that there could be a general bias in Microsoft Windows to use source duplication (category A and B) instead of dynamic libraries (category D), because it is so much harder to update those systems.

Table 2. Projects and their reuse categories

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

For example, AbiWord and Python 1.6-2.4 can be run on both GNU/Linux and Microsoft Windows, but the Windows versions are more vulnerable to bugs because the GNU/Linux versions use a dynamic library. Python 2.5 and later are in category A, and therefore they are vulnerable on both GNU/Linux and Microsoft Windows. Bug 3 is one manifestation of this problem.

4 Case Study: FFmpeg

4.1 Analysis

FFmpeg has a core library called libavcodec that contains encoders and decoders for a wide range of multimedia formats. The FFmpeg project web page lists some 90 other projects that incorporate parts or all of FFmpeg. We focused our attention on libavcodec and, in particular, the libary interface specification in the header file avcodec.h.

Material related to this case study is available at [17].

Figure 1 shows the development of avcodec.h. Each dot represents one change, and its color identifies the responsible user. During the period 2001-07 to 2007-06, 38 different users made a total of 617 changes and the file grew from 177 (5.1 kbytes) to 2940 (90 kbytes) lines of code. Only the most active users are listed in the figure.

As a first approximation we studied the revision history of avcodec.h in the following projects:

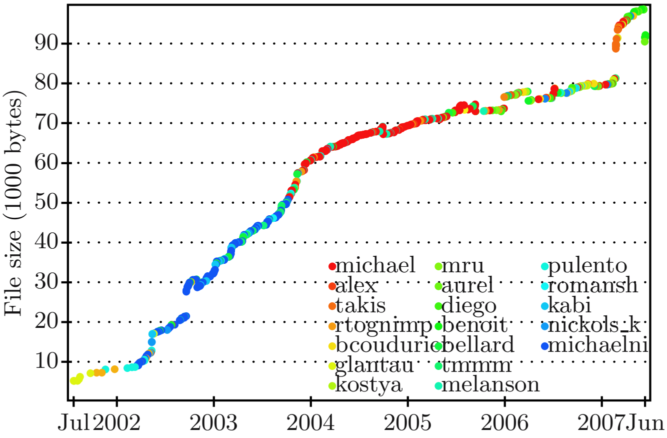

Fig. 1. The evolution of `avcodec.h`

- `avidemux` is a video editing suite
- `avifile` is a multimedia player for Linux systems
- `ffdshow` is a media decoder and encoder for Microsoft Windows systems
- `gstreamer` is a server for streaming audio/video over the internet
- `mythtv` is a software-based personal video recorder for Linux and Mac OS X
- `xbmc` is a multimedia player for Microsoft's Xbox

Unfortunately, it soon became clear that the revision history by itself is not sufficient. Log entries such as "`libavcodec resync`" do not identify the exact revision of `libavcodec` that was used for the update. Even when such information is given, there is no guarantee that the comment is accurate, and in several cases it proved not to be. To overcome this problem, different revisions of `avcodec.h` were downloaded and compared one by one to find matching versions. Together with the information in the revision logs, this produced the kind of information shown in Figure 2. The upper and lower horizontal lines represent the `avifile` and `FFmpeg` projects, respectively. Arrows indicate updates from the latter to the former on the dates shown above and below the project lines, and the small vertical lines on the `FFmpeg` line indicate different revisions of `avcodec.h`.

Table 3. Summary of update data

		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

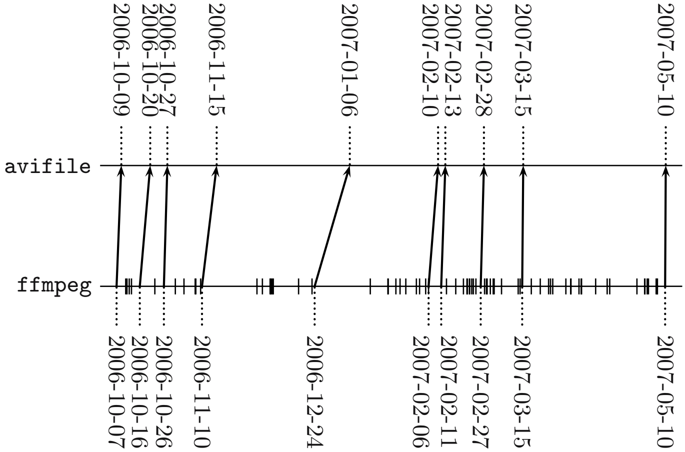

Fig. 2. The 10 most recent updates (from 2006-10-09 to 2007-05-10) of `avcodec.h` in `avifile`

The results are summarized in Table 3. In the three rightmost columns, *Delay* refers to the number of elapsed days between the production of a revision and its use in an update, in other words, how “fresh” it is. In many cases, the update delay is less than a week. `avifile` and `mythtv` were clearly updated much more frequently than the others, even when the longer time frames are taken into account. Still, on at least one occasion the `mythtv` project was not updated for about two months. One fact not shown in this table is that these two projects are updated less and less frequently over time, possibly because `libavcodec` has reached a level of stability where fewer bugs are reported.

4.2 Discussion

We noticed some issues from the `FFmpeg` results:

Shared Interests, Features and Developers New features in the `avifile` project were introduced first in child projects and later introduced into the parent project. This is the result of common developers and interests between the projects. For example, one active developer is a member of both the `FFmpeg` and `avifile` projects.

Feature propagation to and from the parent project supports the theory of OSS development model. If new features are only added in the central project, it would cast serious doubt on the OSS model, and distributed models in general. Bugs are often fixed through inter-project co-operation by users and developers directly communicating with each other. We argue that code reuse in some OSS projects comes close to sharing developers, not just sharing features (code).

Update Propagation Entails Significant Effort In the case of `mythtv` the mismatches were more numerous and more substantial. The requirements of this software include specialized features such as closed captioning and support for multilingual soundtracks, which are not provided by the libavcodec library. In at least one case, a feature was first introduced in the mythtv project and only later in FFmpeg, but, as far as we can tell, the later implementation was not derived from the earlier one. In all events, mythtv is one example of an unforeseen pattern: code reuse takes place, but the code undergoes non-trivial modifications within the new setting, requiring significant effort in update propagation. This happens either because the new setting is significantly different from the original, or because there are new feature requirements.

On Update Propagation Patterns All the projects mentioned above include a complete copy of the libavcodec code and fall into category A. Sometimes, as in the case of mythtv, this is unavoidable if the code needs modification before use.

One example of a project that belongs to category B is xvidcap, a screen capture program for recording user activity, to create video tutorials and other material. Whenever this project is released, the developers include the latest version of libavcodec. This may appear safer, since the library is not as tightly integrated in the source code. However, category A reuse allows users to download a more recent, albeit less stable, development version of the software. In theory, users could themselves replace the copy of libavcodec in xvidcap with the latest release in order incorporate the latest bug fixes, but it is not realistic that this would occur widely in practice.

In the case of ffdshow, we failed to trace updates accurately, because the project modifies avcodec.h too dramatically, but we observed that at least some bugs were reported back to the FFmpeg project.

Comparison of zlib and FFmpeg Bug fix delays in the zlib project were relatively long compared to those in FFmpeg. The main difference in these projects is that functionality of zlib is fixed; it just implements a specific functionality. In constrast, new features are continuously added to FFmpeg. This suggests that that shared interests in developing the same features, as well as shared developers, can decrease the update propagation delay.

5 Guidelines for Managing Bug Fixes

Based on our experience of tracking bug fixes we argue for some guidelines for managing bug fixing inside and between open source projects.

5.1 Avoid Source and Binary Code Duplication

Use dynamic libraries instead of static libraries or source code inclusion. Source code inclusion means that reused code has to be constantly maintained and monitored. In general, dynamic libraries avoid redundant information in the system. This is by far the best practice for code reuse, because it also allows other parties, mostly operating system distributors, to help you.

5.2 Document Important Changes in Version Control History

Maintain a special `SECURITY` file that lists the specific corrective maintenance operations in version control history that fix a security issue. This helps operating system maintainers and other interested parties to track important updates. Also, backporting the security fixes to older and stable versions is easier when the specific commit is known. This is important for production systems that operate for several years. This approach is not limited to security fixes. It can also be used to document other bugs, properties, or interesting factors.

5.3 Tag Important Changes in Version Control History

A special tag (e.g., "`[SECURITY]`") should be added to each commit message that fixes a security issue in the version control system. This makes searching for security fixes easier.

5.4 Maintain a Global Notification System for Changes

Fast updates between projects is important if new features are needed or security matters. Achieving this demands easy updates. We propose that each project create a global notification system for important changes (e.g., a mailing list), that alerts interested parties of specific fixes and features. The system should not flood interested parties about small changes, only the important ones. Also, notifications should be archived so that users can access older changes.

5.5 Facilitate Follow-up of Component Updates

To address the issue of weak command hierarchy, we propose that each project creates a list of reused software components, annotated with a timestamp and an unique identifier (such as the version number or a commit identifier) about the last update.

Also, if possible, a responsible person should be assigned to particpate actively in the project community of the reused component. This will decrease update propagation time and may help to promote a project's interests. This happens when companies contribute to projects like Linux, `gcc` and Samba.

5.6 Write a Procedure for the Update Process

Virtual organization and distributed development means that any developer should be able to replace another developer — at least in theory — but unfortunately experience and knowledge is not easily transfered. A detailed set of guidelines to raise awareness about maintenance operations should help new developers to be more productive. Such guidelines could, for example, spell out how to cross-check for important updates, such as security fixes.

Guidelines should cover issues relating to project maintenance and releases, including managing updates (notification of changes to and from other projects), managing new releases, and the preferred communication style between developers (IRC channels, mailing lists, etc.).

6 Conclusions

Many software companies and software developers are adopting open source components. These components often undergo constant maintenance actions. This paper studied the causes of maintenance in open source components and what controls the user community reaction to maintenance updates. We studied update propagation delay. In particular, we analyzed the effect of the following factors on update propagation delay: reuse category, documentation of changes, and the update process itself.

To find answers we explored updates and bug fix delays in the `zlib` and `FFmpeg` software repositories. We found that update propagation delay varies significantly among projects. Based on this information, we formulated the following guidelines for reusing open source components:

1. Avoid source and binary code duplication.
2. Document important changes in version control history.
3. Tag important changes in version control history.
4. Maintain a global notification system for changes.
5. Facilitate follow-up of component updates.
6. Write a procedure for update process.

Since we have only studied update propagation in the context of two libraries, we cannot claim that the results are generalizable. For further investigation, more case studies should be considered. In our case studies we used custom-built scripts and analysis by hand; a full record of our experiments is available at the website [17]. Although the details are specific to the libraries we looked at, the approach is applicable to other cases, and scalable to larger projects. If the guidelines we have suggested are followed, our approach would be even easier and faster. In order to validate the relevance of the proposed guidelines, a questionnaire to the open source community could be planned and carried out.

References

1. Bolado, M., Castillo, J., Posadas, H., Sanchez, P., Villar, E., Sanchez, C., Blasco, P., Fouren, H.: Using open source cores in real applications. In: DCIS 2003. (2003), 683–688
2. Madanmoha, T., Deapos, R.: Open source reuse in commercial firms. Comm. ACM (2004), 62–69

3. Paulson, J.W., Succi, G., Eberlein, A.: An empirical study of open-source and closed-source software products. IEEE Trans. on Softw. Eng. (2004), 246–256
4. zlib web site: `http://zlib.net`
5. FFmpeg web site: `http://ffmpeg.mplayerhq.hu`
6. Anvik, J., Hiew, L., Murphy, G.C.: Coping with an open bug repository. In: eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange, ACM Press (2005), 35–39
7. Samoladas, I., Stamelos, I., Angelis, L., Oikonomou, A.: Open source software development should strive for even greater code maintainability. Comm. ACM (2004), 83–87
8. Lientz, B.P., Swanson, E.B.: Software Maintenance Management. Addison-Wesley (1980)
9. Capiluppi, A., Boldyreff, C.: Coupling patterns in the effective reuse of open source software. In Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07), IEEE Computer Society (2007)
10. Mockus, A.: Large-scale code reuse in open source software. In Proceedings of the 1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07), IEEE Computer Society (2007)
11. eCos tool chain: `http://ecos.sourceware.org/build-toolchain.html`
12. OpenSSH web site: `http://www.openssh.com`
13. AbiWord web site: `http://www.abisource.com`
14. Internet Society RFC 1950: ZLIB Compressed Data Format Specification version 3.3, `http://tools.ietf.org/html/rfc1950`
15. Internet Society RFC 1951: DEFLATE Compressed Data Format Specification version 1.3 `http://tools.ietf.org/html/rfc1951`
16. Internet Society RFC 1952: GZIP File Format Specification version 4.3 `http://tools.ietf.org/html/rfc1952`
17. `http://www.iki.fi/shd/publications/oss2008/`, and a full dump of the web site: `http://www.iki.fi/shd/publications/oss2008.tar.gz`