Trust Issues in Open Source Software Development

Heikki Orsila
Tampere University of Technology, Finland
heikki.orsila@tut.fi

Jaco Geldenhuys
Stellenbosch University, South Africa
jaco@cs.sun.ac.za

Anna Ruokonen
Tampere University of Technology, Finland
anna.ruokonen@tut.fi

Imed Hammouda
Tampere University of Technology, Finland
imed.hammouda@tut.fi

ABSTRACT

Open source software and the associated development model holds great promise, but the issue of trust is a major challenge. This applies to companies wishing to adopt the open source model but also within open source projects. We investigate this issue by data mining open source repositories to study two related phenomena: update propagation and distributed version control.

1. INTRODUCTION

Companies can leverage open source development in many ways. The internal use of open source tools is widespread and largely uncontroversial. Some companies base products on highly reusable open source components, while still others adopt the open source model wholesale and make their profits from additional services and customized solutions. The open source model is appealing because it offers the promise of reduced costs and the potential to achieve significant market penetration. However, adopting the full model is seen as risky due to a lack of trust in community-driven software. The main concerns are that thoroughly checking quality attributes (reliability, security, safety, etc.) is difficult, and that fixing defects may incur substantial effort or may even be impossible. Moreover, empirical research shows that many claims about open source turn out to be false [2].

Open source development also presents unique opportunities and challenges to researchers. Propriety development is opaque and case studies are expensive and intrusive, whereas open source development produces a considerable amount of publicly available data. However, this data is often incomplete and not as "rich" as the documentation generated with traditional SE processes. Due to the distributed nature of open source, data accuracy can be difficult to establish, and, because the management of open source projects is non-existent or widely distributed, it is not easy to conduct controlled experiments.

Our research focuses on the issue of ***trust***, both between an open source project and its users, and also within the project itself. In this paper, we consider two aspects of this issue. First, we look at update propagation (i.e., how bug fixes and other changes move from project to project); Section 2 describes previously published research [4] that investigates update practices for two reusable components. Second, Section 3 discusses ongoing work to extract information about underlying social networks from open source project repositories, specifically from distributed version control systems.

2. UPDATE PROPAGATION

In the case of highly reusable components, one expects the following basic usage pattern: whenever a new version of a component is released, users immediately download and switch to the new release. Unfortunately, this is rare.

In related work, Capiluppi and Boldyreff provided guidelines to identify and improve highly reusable components [1]. Large-scale reuse involving open source repositories has also been studied by Mockus [3], who identified widely reused code blocks, typically whole components, and common reuse patterns. Reuse in open source can be categorized as follows:

- α the component source code is incorporated in the project during development (e.g., `Linux` kernel),
- β the component source code is added when the project is released (e.g., `xvidcap` project),
- γ users provide the component source code and recompile the project themselves (e.g., `eCos` tool chain),
- δ users provide the component binary (e.g., `OpenSSH`),
- • some combination of above (e.g., `AbiWord`).

Binary reuse (category δ) uses either static or dynamic linking. For static linking a local, stand-alone copy of the library routines is produced at compile-time, while dynamic linking means that library subroutines are loaded at runtime.

The research questions addressed, include the following:

- What reuse mechanisms are commonest?

- What update propagation patterns are commonest?
- How fast/often do users react to new releases of reused components?
- What technical and non-technical criteria influence the community response (e.g., reuse mechanism, product domain, and product development phase)?
- What best practices promote update propagation?

Our research methodology comprises five main steps: (1) formulate research questions, (2) pick good components candidates, (3) extract data by exploring the software repositories of the components, (4) analyze the data with respect to the questions raised, and (5) make recommendations.

We have selected two highly reusable libraries, `zlib` and `FFmpeg`. The software repository of the projects were downloaded, but mining the information is not an easy task and we considered various sources — such as bug reports, mailing lists, IRC conversations, and source code comments — in addition to the revision history. Many open source projects use a public bug tracking system to report and track defects and potential enhancements. Such a system can increase the number of identified problems and enable more efficient corrections. While developers frequently interact with the bug tracking system, little data to characterize their interactions is available. Similarly, revision logs are useful sources in the sense that they record the evolution of a project, but they also present challenges such as incomplete/unreliable data.

2.1 zlib

`zlib` [7] is a lossless compression library used for many file formats and protocols, and included directly in several projects. We investigated three security-related `zlib` bugs and their fixes, and analyzed the time taken to propagate the fixes to eight other projects: `AbiWord`, `BZFlag`, `CVS`, the `Linux` kernel, `ppp`, `Python`, `RPM`, and mirrors of `zlib` itself. The project has two core authors and 42 further contributors. We studied `zlib` version 1.2.3 released on 2005-07-18. The information comes from the `ChangeLog` file; log entries ranged from 1995-04-11 to 2005-07-18, a period of about 10 years. There were 628 documented changes, with 89% coming from the top five of the 42 contributors.

We looked at the following bugs: (1) a double free bug (reported 2002-03-11), (2) a DoS/crash bug (reported 2004-08-25), and (3) buffer overrun/DoS/crash bug (reported 2005-06-30). The time delay between bug report and bug fix is shown in Table 1. The mean and median times in days, computed over all projects in the table, are 97 and 19 days, respectively. From this we draw the following conclusions:

Bug Fix Delay Varies Significantly The time to fix bugs varies significant from project to project, as the distribution of the median and mean times show. In the case of `Python`, the project is still vulnerable to bugs that were discovered years ago. Except for `Linux`, none of the projects has an explicit system for checking for updates in reused projects. This is clearly a precondition for scalable code reuse. This omission is probably due to weaknesses in the project organization, a lack of explicit task lists, and a weak command hierarchy. Technical problems may also play a role: current maintainers of stabilized products may fail to update `zlib` because they do not have the resources for testing new versions and backporting the necessary fixes. Broadly speaking, the update propagation patterns of these projects can be classified as either systematic, negligent, or random.

Table 1: Number of days to fix three different `zlib` bugs. D=does not apply, N=not fixed, U=unknown.

Project	Bug 1	Bug 2	Bug 3	Reuse category
`AbiWord`	1	N	N	α, δ
`BZFlag`	D	D	583	α, δ
`CVS`	1	63	87	α, δ
`Linux`	8	D	D	α
`ppp`	21	D	D	α
`Python`	U	U	90	α
`RPM`	432	25	16	α, δ
`zlib`	0	15	11	α
Min	0	15	11	
Median	5	25	87	
Mean	77	34	157	
Max	432	63	583	

Microsoft Windows Programs are Biased Towards Binary and Source Duplication `Python` 1.6–2.4 and `AbiWord` run on both GNU/Linux and Microsoft Windows, but only the latter are vulnerable to the three bugs. `Python` version 2.5 and later are in category α, so both Linux and Windows are vulnerable. The last column of Table 1 shows the reuse categories for each project. Almost all Linux programs use `zlib` as a dynamic library, which can be updated using a system-wide package manager. Windows lacks such a manager for non-Microsoft products; consequently there is a general bias in Windows to use source duplication (categories α and β) instead of dynamic libraries (category δ).

2.2 FFmpeg

`FFmpeg` [5] is a collection of utilities for processing audio and video files and streams, tools to play and record different media, and a server for distributing media over the internet, for example, for live broadcasts. The library is used in more than 90 projects. We focused on one component of `FFmpeg`, a library called `libavcodec`, for encoding and decoding a wide range of multimedia formats. We studied the revision history of the library header file `avcodec.h` in the following projects: `avidemux`, `avifile`, `ffdshow`, `gstreamer`, `mythtv`, and `xbmc`. It soon became clear that this history by itself is insufficient. Many log entries were imprecise (e.g., "`libavcodec resync`", without any version information), or simply wrong. We addressed this by downloading all original and reused versions of `avcodec.h` comparing the files one-by-one to find matching versions. This produced the kind of information shown in Figure 1. 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, and small vertical lines at the bottom indicate different revisions of `avcodec.h`.

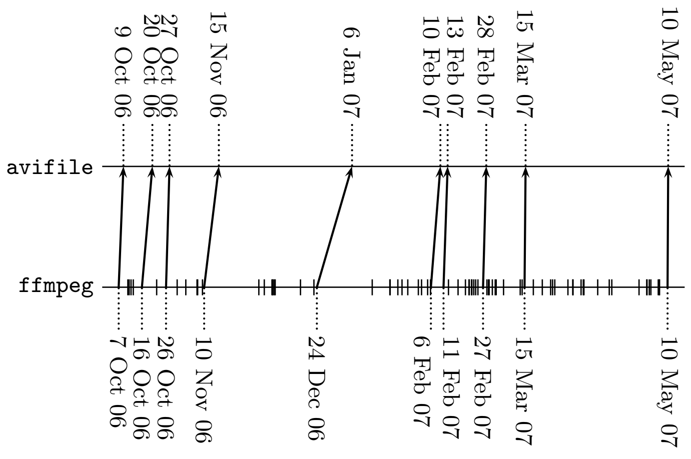

Figure 1: The 10 most recent updates (from 2006-10-09 to 2007-05-10) of `avcodec.h` in `avifile`

Table 2 summarizes the results. The last three columns show the number of elapsed days between the release of a revision, and its use in an update. Often this is less than a week. The `avifile` and `mythtv` projects were updated much more frequently than the others, even when their longer time frames are taken into account. Still, at least once the `mythtv` project was not updated for about two months. What is not shown in the table is that the projects are updated less and less frequently as time passes, possibly because `libavcodec` has reached a level of stability where fewer bugs are reported. These results lead us to the following conclusions:

Sharing Interests, Features and Developers Some features were introduced first in child projects and later into the root project (`libavcodec`). The explanation is that the projects have shared developers and a shared focus. Feature propagation to and from the root project supports the general model of open source development. If new features appeared in the root project only, it would cast serious doubt on the open source model, and distributed models in general.

Update Propagation can Mean Significant Effort In the case of `mythtv`, mismatches in `avcodec.h` were numerous and substantial because of software requires specialized features (e.g., closed captioning, support for multilingual soundtracks) not provided by `libavcodec`. At least once a feature was first introduced in `mythtv` and only later in `FFmpeg`. As far as we can tell, the later implementation was not derived from the earlier. Thus `mythtv` is one example of an unforeseen pattern: code reuse takes place, but the code undergoes non-trivial modifications within the new setting, and update propagation therefore entails significant effort.

Update Propagation Patterns All the above projects include a complete copy of the `libavcodec` code and fall into category α. One example of a project in category β is `xvidcap`, a screen capture program that records user activity for video tutorials and other material. Whenever this project is released, the developers include the latest version of `libavcodec`. This may appear safer because the library is not as tightly integrated, but category α reuse always includes the option of downloading a more recent, albeit less stable, development version of the reused component.

Table 2: Summary of `libavcodec` updates

		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

Comparison Between zlib and FFmpeg Cases Bug fix delays in the `zlib` project were relatively long compared to the `FFmpeg` project. The crucial difference is that the functionality of `zlib` is fixed, while new features are continuously added to `FFmpeg`. It appears that a shared interests in developing similar features, and also, sharing developers, decreases update propagation delay.

2.3 Guidelines

Based on our experience of tracking bug fixes we argue for some guidelines that, we believe, would have improved bug fixing inside and between the projects we examined.

1. Avoid Source and Binary Code Duplication Use dynamic libraries instead of static libraries or source code inclusion. For the latter the reused code has to be maintained and monitored. This is by far the best practice for component reuse because it allows other parties, mostly operating system distributors, to help you.

2. Document Important Changes in the Version Control History Maintain a log file of corrective maintenance operations and specific security issues addressed. This facilitates tracking of important updates, backporting security fixes to older and stable versions. This is especially important for long-term production systems.

3. Tag Important Changes in the Version Control History Add a special tag (e.g., "`[SECURITY]`") to commit messages for security issues in the version control system.

4. Maintain a Global Notification System for Changes Fast updates between projects is important for security-related fixed and enhancements. A global notification system (e.g., based on a mailing list) can alert stakeholders of specific updates. It should focus on major issues, not small updates, and should be archived so users can review older changes.

5. Followup of Component Updates Complex/important projects need an official list of reused components, annotated with a timestamp and unique identifier (version number, commit id, etc.) and a developer specifically responsible for maintaining the list and promoting the project's interests in reused projects.

6. Write a Procedure for the Update Process Virtual organization and distributed development means any developer should be able to replace any other, at least in theory. One solution is a checklist of issues related to project maintenance and releases: managing updates (notifications to and from other projects) new releases, preferred communication between developers (e.g., IRC channels, mailing lists), etc.

3. DISTRIBUTED TRUST

Until recently, version control software have been based on a *centralized* model. In these systems, the software repository is stored in a central location and is updated by only the core developers. Other developers submit their changes to the core group who decide which contributions to commit to the repository. Committers are identified by nicknames, and contributors are (sometimes) credited in the version control log. The repositories are typically accessed through the internet and anyone can "check out" any version of the project, but the result is a snapshot of the project with little or no history information. Examples of such systems are `rcs`, `cvs`, and `subversion`.

New version control systems follow a *distributed* model where every potential developer manages his/her own personal copy of the entire repository, which includes all previous versions and full history information. Developers are no longer classified as core or non-core, and every contributor can act as a committer. Developers are identified by their real names and email addresses. A developer may merge the work of other developers into his/her own repository and may choose to publish his/her own repository on the internet, so that others can merge it into theirs. In essence, distributed version control means that all developers are equal. Examples include `git`, `mercurial`, and `bazaar`. Distributed version control systems facilitate a new social organization within a project. A network of trust is formed in the virtual organization: developer X merges changes from developer Y only if X trusts Y and/or if X has reviewed Y's changes.

We are interested in the impact of these new systems on open source development, and in comparing the predicted effects with the actual. This includes facets such as the (1) technological, (2) social, (3) psychological, and even (4) ideological/political impact on developers. One of the attractive features of the `git` [6] version control system from our point of view is that `git` records detailed information in the version control history. (It is not unique in this regard.) When one developer merges the work of another by merging the remote repository into his/her own local repository, `git` includes the remote version history. Furthermore, `git` distinguishes between the roles of *author* and *committer*, and, for every change (*commit*), includes information about each role in its version history.

One way to measure the trust relationship between developers X and Y is to calculate how long X delays before merging the work of Y into his/her own repository, and how often X merges the work of Y directly or through a third party. Figure 2 shows a small subgraph of the "trust graph" for the `xserver` project. Although it is really a directed graph, it is shown as undirected for the sake of clarity. Each node represents a developer, and the edge labels count the number of times the developers have merged from each other's repositories. The subgraph only shows the most frequent merges; the full graph contains 172 further developers and 940 further merges. This kind of analysis is impossible to perform using older, centralized version control systems. Note that this graph is merely an approximation and does not include any information about delays between merges.

Our plans for further work in this direction is to look at the

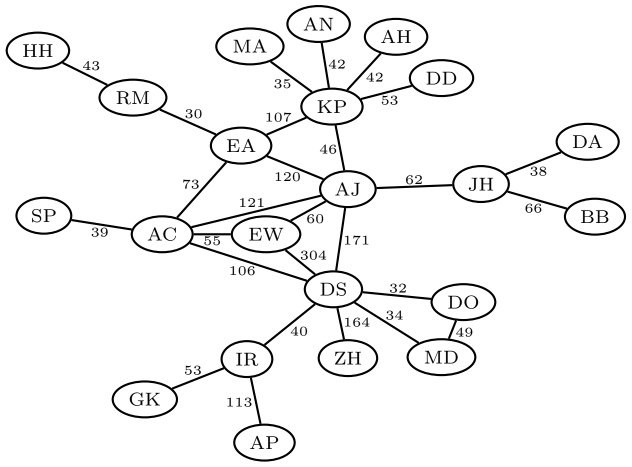

Figure 2: Subgraph of the `xserver` update graph

effects of trust networks with different topologies on other features of open source projects, such as the number of bug reports and how quickly such issues are resolved. Trust networks (and other graphs derived from the repository history) are of course not static, but evolve over time. It is interesting to consider the impact this has on project success.

4. CONCLUSION

We have described earlier work on update propagation, and outlined current work on extracting information about social networks of open source projects. The greatest threat to the validatity of this work is *selection bias*. Unfortunately, the kind of fine grain analysis in Section 2 (given in more detail in [4]) is difficult to carry out on a large scale. Our tonic is to stress the context of our results and to be careful when we generalize. We expect to rectify this in our current work. Ultimately, our goal is to make not only descriptive but also predictive observations to improve, or at least make suggestions to improve, open source practices and to promote trust in community-driven software development.

5. REFERENCES

[1] A. Capiluppi, C. Boldyreff. Coupling patterns in the effective reuse of open source software. In *Proc. 1st Intl. Workshop Emerging Trends in FLOSS Research and Development*, page 9. IEEE Computer Society, 2007.

[2] T. R. Madanmohan, Rahul De'. Open source reuse in commercial firms. *IEEE Software*, 21(6):62–69, 2004.

[3] A. Mockus. Large-scale code reuse in open source software. In *Proc. 1st Intl. Workshop Emerging Trends in FLOSS Research and Development*, page 7. IEEE Computer Society, 2007.

[4] H. Orsila, J. Geldenhuys, A. Ruokonen, I. Hammouda. Update propagation practices in highly reusable open source components. In *Proc. 4th IFIP Intl. Conf. Open Source Software*, pages 159–170. Springer-Verlag, 2008.

[5] `http://ffmpeg.mplayerhq.hu`

[6] `http://git-scm.com`

[7] `http://zlib.net`