Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is

Jesús M. González-Barahona, Paul Sherwood, Gregorio Robles, Daniel Izquierdo

Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment is

Jesus M. Gonzalez-Barahona[1], Paul Sherwood[2], Gregorio Robles[1], and Daniel Izquierdo[3]

[1] Universidad Rey Juan Carlos, Spain
[2] Codethink, UK
[3] Bitergia, Spain

Abstract. Large software compilations based on free, open source software (FOSS) packages are the basis for many software systems. When they are deployed in production, specific versions of the packages in the compilation are selected for installation. Over time, those versions become outdated with respect to the *upstream* software from which they are produced, and from the components available in the compilations as well. The fact that deployed components are outdated is not a problem in itself, but there is a price to pay for not being "as much updated as reasonable". This includes bug fixes and new features that could, at least potentially, be interesting for the deployed system. Therefore, a balance has to be maintained between "being up-to-date" and "keeping the good old working versions". This paper proposes a theoretical model (the "technical lag") for measuring how outdated a system is, with the aim of assisting in the decisions about upgrading in production. The paper explores several ways in which technical lag can be implemented, depending on requirements. As an illustration, it presents as well some specific cases in which the evolution of technical lag is computed.

1 From upstream to deployment

Many production systems are deployed as collections of FOSS (free, open source software) components. All of them are based on the software produced by the corresponding FOSS projects. And usually, as time passes, those projects deliver new releases with more functionality, more fixed bugs, and in many cases, more overall stability and performance [1]. We will use the term "upstream project" for referring to the project originally producing a FOSS component. Upstream projects release, from time to time, versions of the FOSS components they produce and maintain. This release may be continuous, each time a change is done to the code, or discrete, at specific points in time, when the project considers it convenient [2]. In fact, many projects release in both ways: they release continuously in their source code management system (one release per commit), but they also offer "official" tagged discrete releases. In any case, we will consider the released component as the "upstream released package".

But it is unusual that upstream packages are directly deployed in production systems. Instead of that, packages coming from software compilations, usually referred to as "distributions", are used for deployment. We will refer to the packages released as a part of a software compilation as "distribution packages" (to avoid using "compilation packages", which could be easily mistaken for "package produced as the result of compiling some software"). Distribution packages are produced by adapting upstream packages to the policies and mechanisms defined by the software compilation. That usually makes the deployment of components easier, more coordinated with other components, and in general more uniform. This adaption usually includes changes to the code, with respect to upstream. For example, Debian packages include certain files with information on how to build (produce a binary version from the source code) and install the package, and may include changes to improve or adapt it to the distribution [3].

We propose the following model for the process from the production of a FOSS component to its deployment in production:

- The upstream project produces an upstream package. This will be a new upstream release of the FOSS component. This can be just a commit in a Git repository, or a curated official tagged release.
- That new upstream package is used by a software compilation as the basis for a new release of their corresponding distribution package. For producing it, upstream code is used, maybe with some patches applied, and some extra files.
- Deployers use a certain release of the distribution package to deploy the FOSS component in production.

A real deployment may include hundreds or thousands of FOSS components, each corresponding to a certain release of the corresponding upstream package.

This model can be applied to deployments in many scenarios, such as: a collection of Debian packages in a virtual machine or container, providing some cloud-based service; a collection of JavaScript libraries used by a web app, installed from npm.org; or a collection of Python packages (or Ruby gems) installed in a certain machine to run a Python (or Ruby) program; a certain Yocto-based distribution deployed in a certain car (Yocto is a Linux-based distribution oriented towards embedded systems); etc.

2 Technical debt and technical lag

Each deployment scenario has different requirements with respect to their "ideal" relationship with upstream. But in all cases, if no updating action is performed, they stay static, "frozen in the past", while upstream evolves, fixing bugs and adding new functionality. The same happens with software compilations with respect to upstream, if they do not release new updated packages for their components.

Depending on the requirements of the final system, and the resources to maintain it, lags of deployed systems with respect to their software compilations, and

to the latest upstream packages, can be larger or shorter. For example, in deployments with a large number of components and high stability requirements, updating even a single new package can be a challenge: the whole system has to be tested, since the updated package could break something, specially if it is a dependency to many other packages [4]. Even if upstream developers and compilation maintainers did their own thoughtful testing, some integration bug could be triggered when deployed. A significant amount of effort has to be devoted to upgrading, and tracking the behavior of the system after the upgrade. Besides, in some cases the new version could break some assumption about how it works, affecting the overall functionality or performance. Therefore, every new version has to be carefully examined before it can be deployed.

As time passes, if deployed components are not upgraded, the system misses more and more new functionality and bug fixes: it is not "as good as it could be". This situation is akin to the one described as "technical debt" for software development. The metaphor of "technical debt" introduced in 1992, tries to capture the problems caused for not writing the best possible code, but code that could (and should) be improved later on [5]. The difference between code "as it should be" and code "as it is" is a kind of debt for the developing team. If technical debt increases, code becomes more difficult to maintain. A similar concept is "design debt", which translates the concept to the design of software components [6].

In the case we are considering in this paper, we are not exactly in a technical debt scenario, although the concept could be easily extended to include it. The main differences are:

- The concept does not try to capture that deployment is not done "as it should be done". On the contrary, the system "degrades" just with the passing of time, and not because some code needed to be improved when deployed.
- Software development is not really involved, since it only happens upstream, and to a certain extent, in software compilations. Only deployment decisions are considered.
- The metaphor of the debt is difficult to understand in this case, since it is not some "debt" being acquired at some spot, which has to be returned later. Our case could be more comparable to a tax, paid for not being updated, in the form of less functionality and more bugs that we could have if updating.

To recognize the differences, we are coining a new term, "technical lag", which refers to the increasing lag between upstream development and the deployed system if no corrective actions are taken. Deployers need to balance the technical lag their systems acquire as time passes, with the effort and problems caused by upgrading activities.

3 Computing technical lag for a deployment

When measuring technical lag, the first problem is to decide what is the "gold standard" with which to compare. Depending on requirements and needs, the comparison may focus on stability, functionality, performance, or something else.

For example, if there is interest in calculating the technical lag of a Debian-based distribution, with a specific interest in stability, we need to find the standard for stability for Debian-based distributions. One choice could be Debian *stable* (the Debian release which is currently considered "stable"[4]). In a different case, a system could be interested in being as much up-to-date as possible with respect to upstream, because they are interested in having as much functionality and bugs fixed as possible. In this case, the standard would be the latest checkout for each upstream package.

Once the gold standard is defined, we still need to find out the function to compute the lag between the component in the standard compilation and the deployed component. For example, if the focus is on security, the lag function could be the number of security issues fixed in the standard which have not been fixed in the deployed system. If the focus is functionality, the function could be the number of features implemented in the standard which have not been implemented in the deployed component. Some other interesting lag functions could be the differences in lines of source code between standard and deployed components, or the number of commits of difference between them, if both cases correspond to upstream checkouts.

Therefore, when defining the technical lag for a system, it is not enough to just define the deployment to consider. The standard to compare (or the requirements of the ideal deployment) and the function to calculate the lag between versions of the component need to be defined as well.

4 Formal definition of technical lag

Assume we have a **deployment** D composed of a set of certain **components** C, deployed as packages of a certain software collection, and a certain **standard distribution** S, composed by the same set of components, but packaged for that distribution. We denote d_i as a package in distribution D corresponding to component i, while s_i denotes a package in the standard distribution S corresponding to the same component i:

$$D = \{d_i : i \in C\} \qquad S = \{s_i : i \in C\} \tag{1}$$

We define the **lag function for packages** corresponding to a component, $Lag(d_i, s_i)$, as the function computing the lag between packages $d_i \in D$ and $s_i \in S$, for a given component $i \in C$. Lag is defined for all pairs (d_i, s_i), as long as s_i is more up-to-date than d_i, and zero in other cases. Lag has the following properties, which result in the technical lag of a deployment being a non-negative real number. For Lag to be useful, it should fulfill the "lagging condition": computing a larger value for distribution packages "lagging behind". That is, the more distant d_i is from s_i, for some lag requirements, the larger $Lag(d_i, s_i)$ should be.

[4] See `https://www.debian.org/releases/` for a description of the different Debian releases.

We define the **lag aggregation function**, $LagAgg$, as the function used to aggregate the package lags for a set of components.

Finally, we define the **technical lag** for the deployment D with respect to the standard distribution S as the aggregation of the lags between the deployed and the standard distribution packages:

$$TechLag(D, S) = LagAgg(Lag(d_i, s_i) \forall i \in C) \tag{2}$$

When the aggregation function is summation, technical lag is defined as:

$$TechLag(D, S) = \sum_{i \in C} Lag(d_i, s_i) \tag{3}$$

This definition captures how technical lag depends on:

- the distribution selected as the standard distribution to compare
- the function used to calculate the lag for each of the components in the deployment
- the aggregation function for the lags of the deployed components

5 Calculating lag between packages

After the formal definition of the concept, this section will illustrate with an example how the lag can be computed for a certain component, how results differ depending on the distribution selected as the gold standard, and how they however make sense from a practical point of view. For simplicity, we will work with packages for which upstream is working openly in a Git repository. This allows us to model upstream as following a continuous release process, with each commit in the master branch of the Git repository being a release.

We selected components packaged for Debian, because it is a very popular distribution, basis for many other popular distributions, such as Ubuntu. It is common to find Debian or Ubuntu packages in real deployments, both of cloud and embedded systems, to mention just two domain areas. Debian provides the Debian Snapshot Archive[5], which offers for each component a very complete collection of all packages that have been in Debian distributions in the past. This collection includes not only packages in Debian stable releases, but also in Debian unstable and Debian testing, that –because of their nature– may include many interim versions. For each package in the Debian Snapshot archive, its version tag and the date of its release are available. This allows for easy plotting of the evolution of the technical lag of those packages, either just over time, or grouping by releases, as will be shown in the figures in this section.

The selected illustrative cases are the `acl` and `Git` packages. In the case of acl, we have found 24 packages in the Debian archive (released from 2005 to 2012), while for Git we have found 192 (from 2005 to 2016). Only since 2010 Debian Git packages correspond to the "current" Git package, the popular source code

[5] `http://snapshot.debian.org/`

Fig. 1. Lag functions applied to Debian acl package releases, by release date

management system. Before 2010, there were 7 packages which corresponded to GNU Interactive Tools, a set of tools for extending the shell. Therefore, only data since 2010 is really relevant, and we consider 185 Debian Git packages.

To estimate the technical lag of each Debian package, we will assume that it is deployed as such, and compared with the current upstream master HEAD checkout at the time of the study (Oct. 2016). Therefore, following the notation in the previous section: d_i is each of the Debian packages considered; s_i is the latest upstream continuous release (defined as the HEAD of the master branch in the upstream Git repository); and $LagAgg$ is summation.

As Lag, we computed four different functions, to offer different lagging criteria[6]:

- `different_lines` and `different_files`: number of different lines or files, including those that are present only in d_i or s_i.
- `diff_commits`: number of commits, following the master branch of the upstream Git repository, needed to go from the most likely upstream commit corresponding to d_i to the commit corresponding to s_i.
- `normal_effort`: total normalized effort for the commits identified when computing `diff_commits`. We define normalized effort (in days) for an author as the number of days with at least one commit between the dates corresponding to two commits in the master branch. We define total normalized effort (in days) as the sum of normalized effort for all the authors active during the period between two commits.

The first two lag functions capture how different is the deployed component is from the component in the standard distribution (in our case, the most recent

[6] For computing different and common lines and files, we used the Python3 *difflib* module.

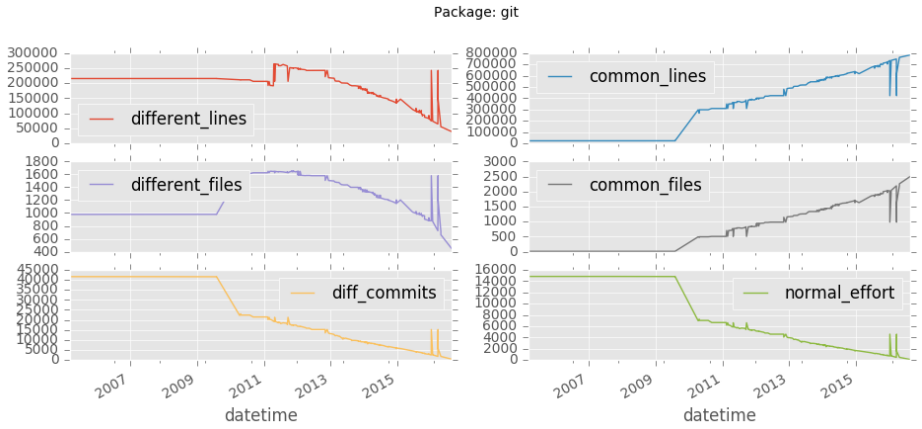

Fig. 2. Lag functions applied to Debian Git package releases, by release date

commit upstream). The last two functions capture how many changes (or, to some extent, effort in changing) were applied to the component in the standard distribution since the upstream release used to build the deployed package.

To provide some context, we computed as well `common_lines` and `common_files`, which is the number of lines an files in common between D_i and C_i (lines exactly the same). Those are not really *Lag* functions, since they do not fulfill the lagging condition: both grew larger when d_i and s_i were closer.

Figures 1 and 2 show the evolution of the lag over time, considering the release time of Debian packages. Each chart shows the value of lag (using one of the lag functions mentioned above) for the release time of each Debian package. For all the four "Lag" functions, it can be seen that they are almost monotonically decreasing over time, clearly converging to zero as time approaches the release time of s_i (the rightmost values). For acl, there is a clear step in 2009, which corresponds to major changes in the component, as will be shown later. For Git the change around 2010 is due to the different packages being tracked (see above, that means that only the data from 2010 onwards is really meaningful). After that point there are some spikes and steps, notably two large spikes in late 2015 and early 2016. But in general, the trend in all charts is clearly decreasingly monotonic.

Figures 3 and 4 are more revealing, because they have into account two common practices in Debian: labeling package releases (in part) with upstream version tags, and releasing slightly modified versions for stable distributions.

The first is observed by the different colors and lines in the charts: all Debian packages corresponding to the same major release have been depicted in the same color, and linked with lines. Now, when we look at the charts for acl in Figure 3, we see how the step in 2009 corresponds to a change in version (from pink to red), which did a major refactoring of the code. That is clearly appreciated in the functions showing common and different lines. In the case of Git, the transition

Fig. 3. Lag functions applied to all releases of the Debian acl package, by release date, organized by version

from GNU Interactive Tools (horizontal line in the left) to the "real" Git is now evident.

The second practice is observed for Git in Figure 4: the red horizontal lines on the right correspond to new releases of "old" packages, fixing some important bugs, since they are still maintained after a long time for some stable distribution. That helps to explain the spikes we saw in Figure 2: those d_i are really "out of order" packages.

In all the figures for the same component, the different functions show similar trends. There are differences, but probably any of them would provide enough information for evaluating if the lag is large enough to justify an update of a deployed package.

6 Discussion and conclusions

Software compilations for FOSS components are usually complex and large, and decisions about when to upgrade specific deployed packages, or whole deployed distributions, is not easy. The complexity of dependency management [7–9], or their significant evolution over time [3] are reasons both to delay upgrading (because of the potential problems), and to consider it (because of the added

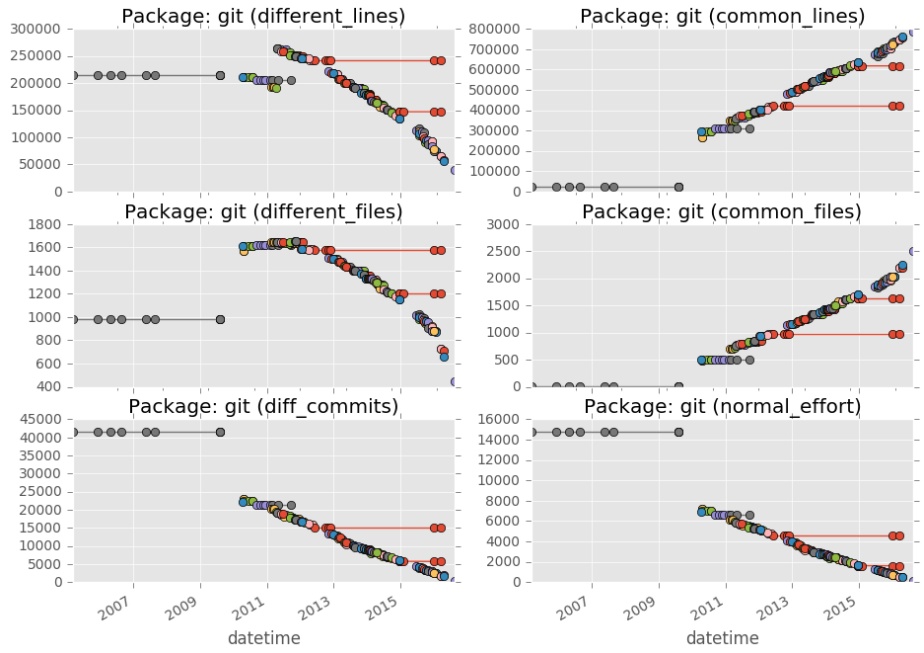

Fig. 4. Lag functions applied to all releases of the Debian Git package, by release date, organized by version

functionality and improved code). The same way that the complexity in dependencies, or the some parameters of their evolution [10] can be measured, we are exploring the concept of technical lag to measure their "degradation" over time with respect to some "ideal" gold standard.

Defining this degradation requires identifying the "ideal" packages to deploy (the "gold standard" to compare), and finding distance metrics (lag functions) to compare deployed software with that standard collection. To be useful, these metrics should track characteristics linked to requirements of the deployed system. As it was discussed in the first part of this paper, a system interested in stability may define very different metrics and gold standard than one interested in maximum functionality. In this paper we have just explored one kind of ideal distribution (the latest available upstream code), and two kinds of metrics: those based on differences in source code (in terms of lines or files), and those based on the number of changes (either the number of commits or the normalized effort). However, many other could be explored.

In particular, the exploration of criteria to define "gold standards" for general or specific scenarios seems promising. Complete industries, such as automotive, embedded systems or cloud, could be interested in finding standard collections with which to compare any deployment, in a way that they may decide better when and what to upgrade, given a set of requirements.

The definition of lag functions requires careful exploration as well. Some of them may be difficult, because the needed information may be heterogeneous, and distributed. But some seem feasible: the number of bugs fixed, or security advisories addressed; the number of new features implemented; improvements in performance, etc. (obviously, when there are ways of collecting that information). This makes us think that there is a lot of work to do in this area, and that we have not even collected all the low hanging fruits.

In this paper, we have considered that distribution packages are directly deployed in production, and therefore make no real difference between the packages in a distribution, and those packages when deployed. In the real world, packages may be deployed with some differences with respect to the distribution packages used. For example, some patches could be applied to fix known bugs. However, this does not make the model less general: the patched packages can be modeled as a new distribution, based on the "original" one, and all the above considerations will apply.

As a kind of a conclusion, we propose technical lag as useful concept to deal with large FOSS deployments. As real-world systems are increasingly built by assembling large collections of FOSS components, it is evident the need of techniques for managing their complexity. In some areas, such as dependency management or architectural evolution, research has been producing results for many years. But there is little evidence that may help in the system-wide maintenance procedures, including those relatively easy, such as when and what to upgrade. With this paper we propose a new line of research, trying to provide support practitioners in may fields of the industry.

Although we are focused on FOSS compilations, it is interesting to notice that the concept of technical lag can in theory be extended to non-FOSS components. However, in practical terms that may be difficult, except if source code and other related information needed to estimate lag is present. This can be the case in some special cases, such as when a company deploys systems composed by a mix of FOSS and proprietary components, but it has access to all the needed information for proprietary ones.

Acknowledgments and reproduction package

The work of Jesus Gonzalez-Barahona and Gregorio Robles has been funded in part by the Spanish Gov. under SobreVision (TIN2014-59400-R), and by the European Commission, under Seneca, H2020 Program (H2020-MSCA-ITN-2014-642954). The research described in this paper was started thanks to a contract funded by Codethink.

All the code and data needed to reproduce the results in this package is available from a GitHub repository[7] (checkout as of December 2016).

[7] `https://github.com/jgbarah/techlag/`

References

1. D. M. German, "Using software distributions to understand the relationship among free and open source software projects," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, p. 24, IEEE Computer Society, 2007.
2. M. Michlmayr, B. Fitzgerald, and K. Stol, "Why and how should open source projects adopt time-based releases?," *IEEE Software*, vol. 32, no. 2, pp. 55–63, 2015.
3. J. M. González-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. Germán, "Macro-level software evolution: a case study of a large software compilation," *Empirical Software Engineering*, vol. 14, no. 3, pp. 262–285, 2009.
4. M. Claes, T. Mens, R. Di Cosmo, and J. Vouillon, "A historical analysis of debian package incompatibilities," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pp. 212–223, IEEE, 2015.
5. W. Cunningham, "The wycash portfolio management system," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, OOPSLA '92, (New York, NY, USA), pp. 29–30, ACM, 1992.
6. J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley Professional, 2004.
7. F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pp. 199–208, IEEE Computer Society, 2006.
8. M. Wermelinger and Y. Yu, "Analyzing the evolution of eclipse plugins," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*, pp. 133–136, 2008.
9. G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015.
10. S. Gala-Pérez, G. Robles, J. M. González-Barahona, and I. Herraiz, "Intensive metrics for the study of the evolution of open source projects: case studies from apache software foundation projects," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, pp. 159–168, 2013.