An Empirical Analysis of Technical Lag in npm Package Dependencies* **

Ahmed Zerouali, Eleni Constantinou, Tom Mens,
Gregorio Robles, and Jesus Gonzalez-Barahona

Software Engineering Lab, Université de Mons, 7000 Mons, Belgium
{ahmed.zerouali | eleni.constantinou | tom.mens}@umons.ac.be
GSyC/LibreSoft, Universidad Rey Juan Carlos, Madrid, Spain
{grex | jgb}@gsyc.urjc.es

Abstract. Software library packages are constantly evolving and increasing in number. Not updating to the latest available release of dependent libraries may negatively affect software development by not benefiting from new functionality, vulnerability and bug fixes available in more recent versions. On the other hand, automatically updating to the latest release may introduce incompatibility issues. We introduce a technical lag metric for dependencies in package networks, in order to assess how outdated a software package is compared to the latest available releases of its dependencies. We empirically analyse the package update practices and technical lag for the npm distribution of JavaScript packages. Our results show a strong presence of technical lag caused by the specific use of dependency constraints, indicating a reluctance to update dependencies to avoid backward incompatible changes.

Keywords: software library, technical lag, package dependency, npm

* This is a preprint of the paper published as Zerouali A., Constantinou E., Mens T., Robles G., Gonzalez-Barahona J. (2018) An Empirical Analysis of Technical Lag in npm Package Dependencies. In: Capilla R., Gallina B., Cetina C. (eds) New Opportunities for Software Reuse. ICSR 2018. Lecture Notes in Computer Science, vol 10826. Springer. DOI 10.1007/978-3-319-90421-4_6

** The work presented in this paper has been funded in part by the European Union's Horizon 2020 Research and Innovation Programme under the Marie Sklodowska-Curie grant agreement No 642954 (Seneca), by the bilateral FRQ-FNRS research program 30440672 SECOHealth, by the research project TIN2014-59400-R SobreVision funded by the Spanish Government, and by Excellence of Science project 30446992 SECO-Assist financed by FWO - Vlaanderen and F.R.S.-FNRS.

1 Introduction

Today's (open source) software systems are increasingly relying on reusable libraries stored in online package distributions for specific programming languages (e.g., npm, RubyGems, Maven) or operating systems (e.g., Debian and Ubuntu). The availability of such a huge amount of reusable packages facilitates software development and evolution. However, it can also cause problems, such as software becoming out of date with respect to more recent package releases. This implies that bug fixes and new functionality are not utilized by applications using such library packages [11]. A possible reason for this may be the mechanisms used to specify, track and maintain dependencies, that are often observed to induce a latency when updating a library [12].

Gonzalez-Barahona et al. [10] refer to this problem as "technical lag", indicating the increasing difference between deployed software packages and the upstream packages that they reuse. Technical lag captures the delay between versions of software deployed in production and more recent compatible versions available upstream. Reusable software packages may also suffer from technical lag, because they may depend on (i.e., reuse) other packages of which more recent releases are available. Depending on an outdated release of a reusable package may not be a problem in itself ("if it ain't broke, don't fix it"), but it comes at a price of not being able to benefit from new functionalities, or patches that fix known bugs and security issues [11].

While assessing the problem of technical lag is important at the level of individual packages, it becomes even more relevant at the level of large distributions of software packages where packages depend, directly or indirectly, on a large number of other packages [6, 7]. If a package imposes too strict version constraints on its dependencies, it may become outdated. As a direct consequence of this, all packages that directly or indirectly depend on such packages may suffer from technical lag. This may affect a large portion of the entire ecosystem. On the other hand, updating even a single package to address technical lag may be quite challenging, since the changes of the updated package may cause a ripple effect through the ecosystem, potentially causing many other packages to break and requiring significant portions of the ecosystem to be tested.

To advance the body of knowledge on how software systems are reusing packages through dependencies and how this induces technical lag, we carry out a large-scale empirical study on the npm package distribution. We focus on five research questions: **RQ$_0$** How do packages manage their dependencies?; **RQ$_1$** How often do packages release new versions?; **RQ$_2$** What is the technical lag induced by outdated dependencies?; **RQ$_3$** How often do dependencies inducing technical lag release new versions?; and **RQ$_4$** What is the appropriate moment to update dependencies? To answer these questions, we measured technical lag based on dependencies between package releases in the npm registry. As a result of our study, we found a high potential of technical lag. One of the causes of technical lag appears to be the use of too strict package dependency constraints, disallowing packages to benefit from more recent releases of their dependencies.

2 Dataset and background

There is a large variety of package managers for different programming languages (CRAN for R, RubyGems for Ruby, npm for JavaScript, etc.) and datasets providing their historical evolution. For example, the libraries.io dataset monitors $2.5M$ packages across 34 package managers. For our study, we selected the npm package manager because: (1) npm is a popular and growing package manager, allowing us to exploit a variety of ways that dependency-based reuse and updates occur, and limiting bias in our study; (2) npm has a large and active developer community [4], making our results relevant for a large number of developers; and (3) JavaScript is the most popular programming language on GitHub, the world's leading software development platform [3].

Our empirical study uses the libraries.io[1] dataset, an open source package tracker and repository containing metadata of package versions and their dependencies [14], available as open access under the CC Share-Alike 4.0 license. Package information includes the version number, date of publication and dependency information, i.e., used packages, constraints and dependency types. The used timeframe for our analysis was between *09-11-2010* (i.e., the first package release found in the dataset) and *02-11-2017*. Our dataset comprises $610K$ packages with more than $4.2M$ releases and $44.9M$ dependencies among them.

In npm, there are three different types of dependencies. *Runtime* dependencies are required to install and execute the package. *Development* dependencies are used during package development (e.g., for testing). *Optional* dependencies will not hamper the package from being installed if the dependency is not found or cannot be installed. Our analysis includes all three types of dependencies and the dataset contains 43% runtime dependencies, 56% development dependencies, and only 1% (133 in total) optional dependencies.

All code and data required to reproduce the analysis in this paper are available on https://github.com/neglectos/techlag_icsr/.

2.1 Semantic Versioning and Dependency Constraints

With many software packages being created and updated every day, it is important to standardize the way of versioning and keeping track of package releases and dependencies. *Semantic Versioning* (semver.org, referred as *semver*) has become a popular policy for communicating the kinds of changes made to a software package. It allows dependent software packages to be informed about possible "breaking changes" [2]. A *semver*-compatible version is a version number composed of a *major*, *minor* and *patch* number. The version numbers allow to order package releases. For example, *1.2.3* occurs before *1.2.10* (higher patch version), which occurs before *1.3.0* (higher minor version), which occurs before *2.1.0* (higher major version). Backward incompatible updates should increment the major version, updates respecting the API but adding new functionalities should increment the minor version, while simple bug fixes should increment the patch version. Unfortunately, the semantic versioning policy is not always

[1] http://www.libraries.io

respected by package maintainers, as an empirical study on Java packages in the maven package manager has revealed [15].

Constraint	Interpretation	Example	Satisfied versions
fixed	exact version required	= 2.3.1	2.3.1
minimal	only use releases above the declared version	>= 2.3.0	$\geq$ 2.3.0
maximal	only use releases below the declared version	< 2.3.0	< 2.3.0
latest	use latest available release	latest	$\geq$0.0.0
hyphen ranges	only use releases between two versions	1.2.3 - 2.3.4	$\geq$=1.2.3 $\wedge$ <=2.3.4
x ranges	only update where "x" or "*" is	1.2.x	$\geq$=1.2.0 $\wedge$ <1.3.0
tilde ($\sim$)	only update patches	$\sim$ 2.3.0	$\geq$ 2.3.0 $\wedge$ <2.4.0
caret ($\wedge$)	only update patches and minor releases	$\wedge$2.3.0	$\geq$2.3.0 $\wedge$ <3.0.0
or / and / ...	combine multiple logical constraints	2.5.3 || >=2.8.1	2.5.3 $\vee$ 2.8.1 $\vee$ >2.8.1

Table 1: Types of dependency constraints for npm package dependencies.

Like maven and many other package managers, npm recommends software packages to follow a specific flavor of semantic versioning[2]. Package releases specify their version number in the metadata (stored in a *json* file[3]), and use dependency constraints to specify the version ranges of other packages they depend upon. These constraints are built from a set of comparators that specify versions that satisfy the range. Table 1 summarizes the types of dependency constraints for npm, their interpretation and an illustrative example of each constraint type.

3 Measuring technical lag

The technical lag of a package release can designate anything that makes the release out of date to its upstream versions [10]. For example, bugs or security vulnerabilities could have been fixed, or new functionalities may have been integrated into newer versions of the package. In the remainder of this section, we introduce the necessary terminology to formally define technical lag, and accompany it with illustrative examples.

Definition 1. *Let* $\mathbf{P}$ *be the set of all* ***packages*** *belonging to the ecosystem, and* R *the set of all corresponding package* ***releases****. We define:*

- $\mathbf{releases} : \mathbf{P} \rightarrow 2^R$ *returns the set of releases of a given package. As a shortcut, for a package* $p \in \mathbf{P}$ *we denote* $R_p = releases(p)$
- $\mathbf{version} : R_p \rightarrow \mathbb{N}\times\mathbb{N}\times\mathbb{N} : r \rightarrow (Major, Minor, Patch)$ *associates a semantic version number to a release.*
- $\mathbf{major} : R_p \rightarrow \mathbb{N}$, $\mathbf{minor} : R_p \rightarrow \mathbb{N}$ *and* $\mathbf{patch} : R_p \rightarrow \mathbb{N}$ *obtain the first, second and third component of a version number, i.e.,* $version(r) = (major(r), minor(r), patch(r))$
- $\mathbf{time} : R_p \rightarrow Date$ *returns the release date of a package release.*

[2] https://docs.npmjs.com/misc/semver
[3] https://docs.npmjs.com/files/package.json

- **latest** : $\mathbf{P} \rightarrow R : p \rightarrow r$ *such that* $r \in R_p$ *and* $time(r) = max_{q \in R_p}(time(q))$ *returns the latest available package release of* p.
- *A* ***dependency*** $d \in \mathbf{D}$ *is a pair* (q, c) *where* $q \in \mathbf{P}$ *is a* ***required package****, and* $c : R_q \rightarrow Boolean$ *is a* ***version constraint*** *expressing the range of allowed releases of* q.
- ***deps*** : $R_p \rightarrow 2^{\mathbf{D}}$ *returns the set of dependencies of a package release.*
- *Given a dependency* $d = (q, c) \in deps(r_p)$ *we define* $\mathbf{lastAllowed}(d) = r_q \in R_q$ *such that* $time(r_q) = max_{\{r \in R_q | c(r) = true\}}(time(r))$

We use the technical lag of a package release to expresses the inability of that release to benefit from the most recent version of a required package, because the dependency constraint provides an upper bound on the allowed version of the required package. Lag can be expressed as a time difference (referred to as **time lag**), by measuring the elapsed time between the date of the used version of a dependency and the date of the latest available version of this dependency. Lag can also be expressed as a difference in version numbers (referred to as **version lag**), indicating how many major, minor or patch versions the release of a required dependency is behind. More specifically:

Definition 2. ***[Technical Lag]*** *Let* r *be a package release, and* $d = (q, c) \in deps(r)$ *a dependency of* r *on a package* q *satisfying the dependency constraint* c. *Assume that* d *is an outdated dependency, i.e.,* $lastAllowed(d) < latest(q)$. *We define the* ***time lag*** *of* d *(at the release date of* r*) as:*

$$\mathbf{tLag}(d) = time(latest(q)) - time(lastAllowed(d))$$

If $tLag(d) = 0$, *the dependency is up to date with the most recent available version. If* $tLag(d) > 0$, *the dependency is outdated, since a more recent version of the required package exists.*
We define the ***version lag*** *of* d *(at the release date of* r*) as:*

$$\mathbf{vLag}(d) = (\Delta Major, \Delta Minor, \Delta Patch), where$$

$\Delta Major = major(latest(q)) - major(lastAllowed(d))$

$\Delta Minor$ *is the amount of successive version numbers between* $lastAllowed(d)$ *and* $latest(q)$ *that changed their minor number without changing the major number.*

$\Delta Patch$ *is the amount of successive version numbers between* $lastAllowed(d)$ *and* $latest(q)$ *that changed their patch number without changing the minor and major numbers.*

For example, suppose that required package q of dependency d has the following series of version numbers (1.0.0, 1.0.1, 1.0.2, 1.1.0. 1.1.1, 2.0.0). The major number changed once (1.1.1 $\rightarrow$ 2.0.0), the minor number changed once without changing the major number (1.0.2 $\rightarrow$ 1.1.0), and the patch number changed three times without changing the minor and major numbers (1.0.0 $\rightarrow$ 1.0.1, 1.0.1 $\rightarrow$ 1.0.2 and 1.1.0 $\rightarrow$ 1.1.1). This results in a version lag of $vLag(d) = (1, 1, 3)$.

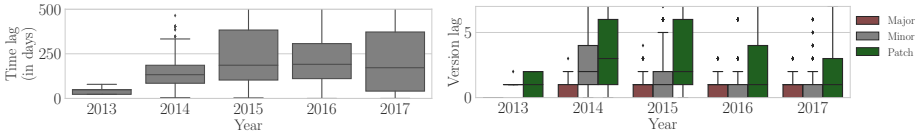
Table 2 illustrates the evolution of the different releases of npm package jasmine over time, in terms of its dependency on the glob package. jasmine version

2.0.1 released in *2014-08-22* contains a dependency d = (glob, ^3.2.11). This dependency implies that all versions of glob ranging between >= 3.2.11 and < 4.0.0 are allowed. At the release date of jasmine 2.0.1, the most recent version of glob in that allowed range was 3.2.11 (released in *2014-05-20*), but the latest available version of glob at that date was 4.0.5, which was released in *2014-07-28*. Since this version does not satisfy the dependency constraint, jasmine 2.0.1 had a technical lag $tLag(d) = 69$ days, for this dependency d at the time of its release. Its version lag was $vLag(d) = (1, 0, 5)$ since, between release 3.2.11 and 4.0.5 of glob, *one* major version and 5 patch versions were released.

release date	jasmine version	glob dep. constraint	lastAllowed(glob) version : date	latest(glob) version : date	tLag	vLag
2014-08-22	2.0.1	^3.2.11	3.2.11 : 2014-05-20	4.0.5 : 2014-07-28	69	(1,0,5)
2014-11-14	2.1.0	^3.2.11	3.2.11 : 2014-05-20	4.0.6 : 2014-09-17	120	(1,0,6)
2014-12-01	2.1.1	^3.2.11	3.2.11 : 2014-05-20	4.3.1 : 2014-12-01	195	(1,3,16)
...	...	...	...	...	...	...
2015-12-03	2.4.1	^3.2.11	3.2.11 : 2014-05-20	6.0.1 : 2015-11-11	540	(3,5,39)
2016-08-31	2.5.0	^3.2.11	3.2.11 : 2014-05-20	7.0.6 : 2016-08-24	827	(4,5,47)
2016-09-07	2.5.1	^7.0.6	7.0.6 : 2016-08-24	7.0.6 : 2016-08-24	0	(0,0,0)

Table 2: Evolution of jasmine's dependency on the glob package.

Version 2.5.1 was released in *2016-09-07* and contains a dependency d = (glob, ^7.0.6). The latter constraint implies that all versions of glob ranging between >= 7.0.6 and < 8.0.0 are allowed. At the release date of jasmine 2.5.1, the last available version of glob was 7.1.2 (released in *2017-05-19*). Since this version satisfies the dependency constraint, jasmine 2.5.1 did not suffer from technical lag w.r.t. this specific dependency at the time of its release.



(a) Time lag (b) Version lag

Fig. 1: Time and version lag distribution, per year, for all outdated dependencies of eslint.

The technical lag of a package release can be computed in terms of the technical lag of all its outdated dependencies. Figure 1 shows the evolution of the time and version lag for all outdated dependencies of the eslint package.

Figure 1a shows that the time lag seems to have increased considerably since 2015. Figure 1b seems to reveal a decrease over time in the distribution of the patch, minor and major version lag. This is explained by the fact that at first, eslint was using tilde for its dependency constraints, but after 2015, it started using caret instead. The time lag increased in 2017 because eslint did not update its dependencies karma-babel-preprocessor and cheerio, while these packages did not release new versions after 2015 until mid 2017.

4 Results

RQ_0: How do packages manage their dependencies?

In order to gain insight in how npm package dependencies are managed, we investigate which types of dependency constraints are used, and how packages add, remove or update their dependencies throughout the package release history.

Figure 2 illustrates the proportion of the types of package dependency constraints used. We observe that caret (^) is most frequently used, covering 68.2% (30.6M) of all constraints. This suggests that most package maintainers want to avoid backward incompatible changes, but keep benefiting from bug fixes (patch updates) and new functionalities (minor updates). 15.7% (7M) of all dependencies were specified with an exact version, which is remarkable since this means that they prefer a compatible but possibly older version of a dependency, rather than benefiting from updates. 7.8% (3.4M) of all dependencies used a tilde (∼) constraint; and 4.1% (1.8M) of the dependencies use the "latest" version (i.e., **, x, latest, *.*.*, x.x.x*). Other dependency constraints, such as external URLs to Git repositories, local files and explicit boolean comparators, were used by 4.2% (1.9M) of all dependencies. The above findings indicate that developers are mainly concerned with backward compatibility, and that there is a potential of too strict dependency constraints leading to technical lag.

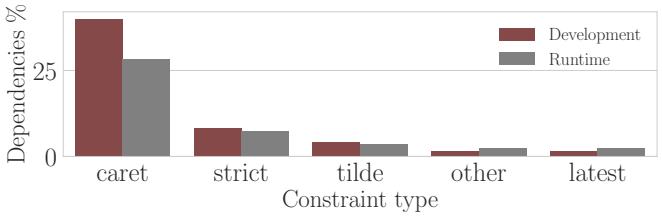


Fig. 2: Proportion of the types of dependency constraints used in npm.

Next, we investigate how npm developers change their dependencies. Figure 3 shows the distribution of added, removed and unchanged dependencies for each type of release during package evolution. Most packages do not appear to change

their dependencies over time. If they do, changes in dependencies mainly happen in major releases (both additions and removals). New dependencies are mainly added in major and minor releases, and only very occasionally in patch releases. Dependencies are removed almost exclusively in major releases.

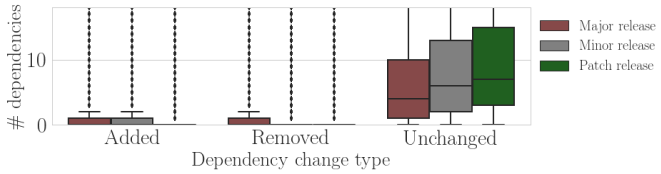


Fig. 3: Number of removed, unchanged and added dependencies between subsequent package releases

To investigate how packages update their dependencies, for each major, minor or patch release, we identified the type of dependency updates that can occur. Figure 4 shows that only in major releases, packages update their dependencies to a newer available *major* version. Also, packages tend to update to minor releases of their dependencies in both major and minor releases, but not in patches. On the contrary, npm packages update their dependencies at the patch level regardless of their new release type (major, minor and patch). Moreover, we found that 1.2M (i.e., 97.5%) of all dependency version changes were updated to more recent versions, while 29K (i.e., 2.5% of all changes) were downgraded to an older dependency version. 14K (i.e., 49%) of the dependency downgrades were made while the package had major version 0 (i.e., 0.*.*).

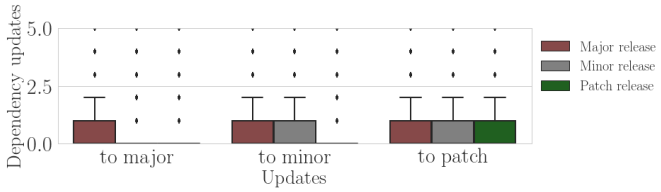


Fig. 4: Number of updated dependencies between package releases, classified by release type of the update.

Findings: npm packages are updated frequently. Dependencies are added or removed rarely, and mostly in major releases. Technical lag is induced because dependency constraints prevent package dependencies from being updated in presence of backward incompatible changes.

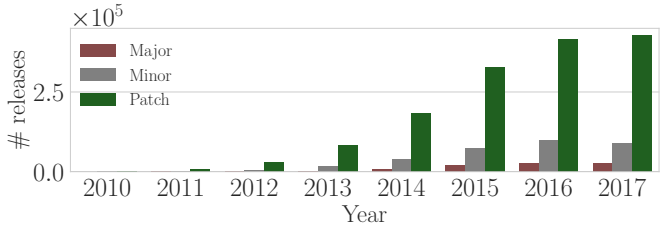


Fig. 5: Number of releases of required packages in npm, per year and per release type.

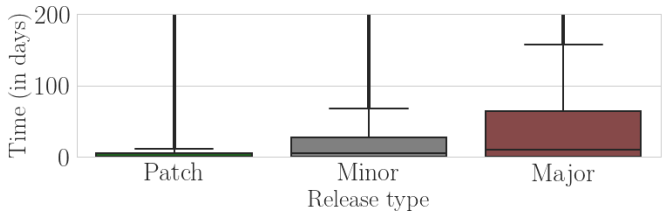


Fig. 6: Time needed to update a package to a patch, minor or major release.

RQ_1: How often do packages release new versions?

Technical lag is not only induced by dependency constraints, but also by the rate at which packages produce new releases when they serve as dependencies of other packages. More concretely, packages that release often can cause technical lag in packages using them if these packages cannot keep up with the updating process. We thus analyze how often packages release new versions by only considering required packages, i.e., packages used as dependencies by other packages.

Figure 5 shows the number of releases of required npm packages per year and release type. It shows that the number of releases of required packages increases every year, which is not surprising because of the growing number of npm packages over time. If we consider the entire dataset from 2010 to 2017, the majority of all package releases (80.1%) are patches, while only 15.9% are minor releases, and 4.0% are major releases. This seems to imply that dependent packages in npm mainly benefit from patch releases. It also implies that technical version lag is mainly occurring at the patch level.

Figure 6 shows the time it takes to update the release of a package to a new patch, minor or major version. We found that npm packages take more time to release a major version than a minor or patch version. This is expected since simple bug fixes are very frequent and easier to make, as compared to adding new functionalities. The same is true for backward incompatible changes, i.e., they require more effort to implement, and thus take more time to release. We also found that the average time to release a patch, minor and major versions corresponds to 13 days, 1 month and 2 months respectively. Thus, package main-

tainers must *monitor and update their dependencies often*. Otherwise, technical lag can pile up throughout time from different outdated dependencies.

Findings: Required packages release new versions frequently, increasing the likelihood of dependent packages suffering from an increased technical lag.

RQ_2: What is the technical lag induced by outdated dependencies?

To identify outdated package dependencies, we use all package releases in our dataset, i.e., 4.2M releases for 610K packages, and all 44.9M dependencies between them. We use the npm-semver constraint definitions to calculate the range of versions that satisfy each constraint. We filter out 2% (743k) of the dependencies for which we don't have any information about the dependency version (i.e., local files, git URLs). In the following step, we use the date of each package release to find the time lag induced by its dependencies. This is achieved by identifying the range of versions that satisfy each dependency constraint, and by comparing the time difference between the latest version that satisfies the constraint and the latest available version of the dependency. We found 27% (i.e., 11.9M) of the 44.1M dependencies to be outdated. These outdated dependencies were used by 60% of all considered packages.

Figure 7 shows the proportion of the types of constraints used for outdated dependencies only. We observe that most of the outdated dependencies used the caret constraint (57%), which is expected since 68.2% of all dependencies use this type of constraint. Constraints using a fixed version can also induce technical lag. Indeed, we found that 28% of all outdated dependencies used an exact version constraint. Contrary to our expectations, only 12% of the outdated dependencies occur due to the tilde constraint. The remaining 3% used yet another constraint type (e.g., 1.2.x). Overall, developers often use caret because it ensures compatibility, but it should only be used when they *closely monitor their dependencies*. Given the fact that most of the outdated dependencies use caret, developers either misuse this constraint or neglect to upgrade their dependencies.

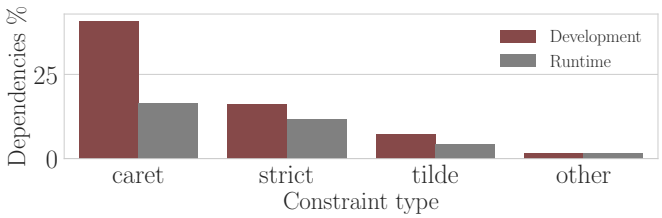
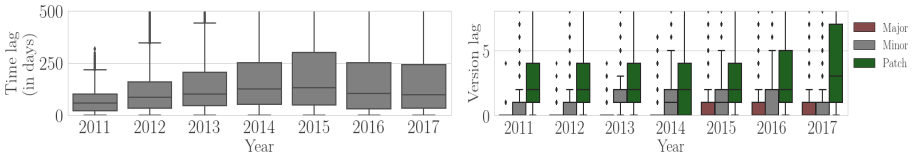


Fig. 7: Proportion of the types of constraints used for outdated dependencies.

Figure 8a illustrates the yearly distribution of time lag from 2011 until 2017. From 2011 to 2015, we notice an increase in time lag. This is likely to be a statistical artifact because the time lag is likely to increase as more releases

become available. However, from 2015 to 2017, we observe that the time lag started to decrease, although there still is a very high median of 100 days in 2017. This decrease may suggest that developers started to care more about their dependency updates, or simply that a lot of package versions were released after 2015 and they were used as outdated dependencies, but did not have sufficient time yet to be able to accumulate an important time lag.

Figure 8b illustrates the yearly evolution of the version lag distribution. We observe that version lag increases for all release types from 2011 to 2016. In 2017, we notice that the patch version lag is increased compared to the other release types. This means that packages serving as dependencies release much more patch versions in 2017, complying with our earlier observation that time lag decreases in the same year. To statistically verify this, we carried out the non-parametric *Mann-Whitney U* test that does not assume normality of the data. The null hypothesis assumes that the patch distributions of two yearly populations are identical. We rejected H_0 with statistical significance $p < .001$ when comparing the patch distribution of any previous year with the year 2017. We only found a small effect size at $|d| <= 0.2$ using Cliff's Delta, a non-parametric measure quantifying the difference between two groups of observations. However, when we compute the median time lag and version lag over the entire npm lifetime, we find: $tLag = 106$ days and $vLag = (0, 1, 2)$.



(a) Time lag (b) Version lag

Fig. 8: Distribution of technical lag, per year, in outdated dependencies.

Findings: Outdated dependencies induce a median of time lag of three months and a half, and median version lag of one minor and two patch versions.

RQ_3: How often do dependencies inducing technical lag release new versions?

This research question investigates if required packages that release often are more likely to lead to technical lag in dependent packages. To this extent, we calculate the number of available versions and the number of versions created each year for such packages. Figure 9 shows the number of available versions of the used outdated and up-to-date dependencies. Packages that are required as dependencies and are outdated have more frequent releases than other required packages. Hence, the package release frequency is a source of technical lag. To

statically confirm our observations, we carried out a two-sided *Mann-Whitney U* test between both distributions and we found a statistically significant difference ($p < .001$) for both the available versions per year and throughout their lifetime. Using *Cliff's Delta*, we found a strong effect size at $|d| = 0.54$ and $|d| = 0.65$, implying that the observed difference between the two groups is big.

Findings: Packages that are required as dependencies and are outdated have more frequent releases than other required packages.

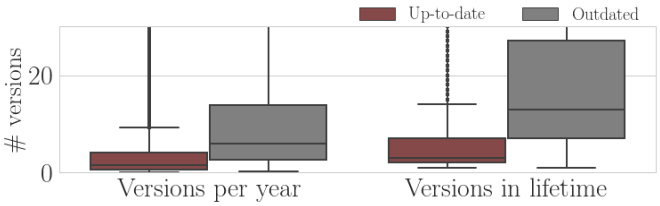


Fig. 9: Number of available versions of packages used by outdated and up-to-date dependencies.

RQ_4: What is the appropriate moment to update dependencies?

If a required package has updated to a new (minor or major) release, dependent packages may wish to delay upgrading to this new release, as it may still be unstable or contain bugs. We thus analyze how long it takes before a patch, minor or major release of a package is updated to new patch versions. Answering this question will help package maintainers to choose the suitable moment to update their dependencies.

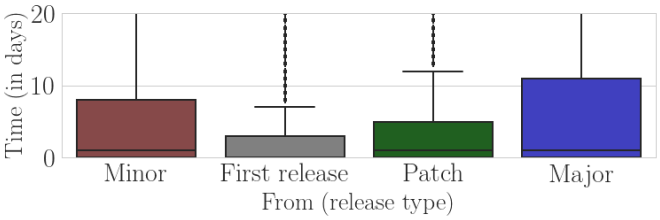


Fig. 10: Time required to update a package release to a more recent patch.

Figure 10 shows the distributions of the time require to update a package release to a more recent patch. We observe that it takes slightly less time to

release a patch version after a minor release ($mean = 17$ *days*) than after a major release ($mean = 21$ *days*). It takes even less time to update a patch release with another patch ($mean = 13$ *days*). We also notice that the time required to make a patch to the first package release is very short ($mean = 14$ *days*, $median = 0$ *days*). A possible explanation could be that first releases tend to be unstable and immature, since most of them have major release number 0. Studying the different behaviour of releases with major version 0 is part of future work.

New major releases seem to be more stable. To verify if there is a statistically significant difference between the distributions of Figure 10, we used the *Wilcoxon rank-sum* non-parametric test. We indeed found statistically significant differences in the mean ranks of each release type with $p < .001$.

Findings: New major releases take longer to receive a patch update.

This suggests that developers should not start using newly available packages immediately because they may still contain bugs that need new patches.

5 Discussion

Ideally, deployed packages would depend on the most recent available version of their dependencies, thus benefiting from the latest functionality and bug fixes. In practice, however, packages have a certain technical lag because many developers choose not to upgrade certain dependencies ("if it ain't broke, don't fix it"). Moreover, in many cases developers choose not to update because new major releases may include new functionality that is not needed. Dietrich et al. [8] found that 75% of all version upgrades in 109 Java programs are not backward compatible, but only few are actually affected by the incompatible changes. In npm, where there is a strong presence of micropackages for trivial functions, developers are concerned with the risk of breakages such packages introduce [1]. In a similar spirit, our findings show concrete evidence of technical lag and the need to monitor dependencies.

Our findings suggest that npm packages could benefit of better procedures for updating. However, it is essential to mention that one should always balance between being up-to-date and the increasing effort, cost and risk of updating. Therefore, further analysis is required to investigate how much functionality is added, how many bugs were fixed and which other changes occurred to the package when updating.

There are many ways to quantify technical lag of package releases w.r.t. outdated dependencies. In this paper we focused only on two definitions, time lag and version lag. We computed these metrics at the package dependency level, but it is also possible to compute them at the package release level by aggregating the lag of all outdated dependencies, e.g.,

$$\mathbf{vLag}(r) = (\sum_{d \in lag(r)} major(vLag(d)), \sum_{d \in lag(r)} minor(vLag(d)), \sum_{d \in lag(r)} patch(vLag(d)))$$

As an example, the aggregated lag of jasmine 2.0.1 is caused by 4 outdated dependencies (commander, grunt-contrib-jshint, glob and shelljs), which cause an

aggregated lag of 1 major, 5 minor and 7 patch releases in total. Since different lag definitions may produce different results; we intend to ask feedback from package maintainers about how informative each definition and way of measurement is when inspecting technical lag.

The concept of technical lag is related to, but different from, the metaphor of technical debt [9, 5]. Technical debt refers to the qualitative difference between code "as it should be" and code "as it is". Technical lag refers to the increasing lag between the latest available upstream versions of packages used by a software system and those actually used in the deployed system. As real-world software systems increasingly rely on reusable libraries, techniques for managing their deployment are compulsory. In this work, we studied technical lag as a useful way to assess reuse in large open source deployments. By analyzing npm package dependencies, we found that a large number of package releases suffer from technical lag. This finding opens the door for comparative research with other package repositories of reusable libraries (e.g., Maven for Java, RubyGems for Ruby) and other ways of assessing technical lag.

During our npm analysis, we observed some specific use of version numbers that seems to go against the semantic versioning policy. For example, many packages remain in major version zero (0.*.*) for a long time. If packages are in the initial development phase for long, then a (0.*.*) version should be used. On the contrary, if packages have progressed beyond this phase, then they should upgrade to a (1.*.*) version.

We noticed that many package releases append pre-release tags (e.g., -alpha.1, -beta.3, -rc.0) to their version number or dependency constraints. The semantics of these pre-release constraints is different from normal version constraints according to npm semantic versioning. For example, constraint ^1.2.3-alpha accepts releases in the range ≥1.2.3-alpha.3 and <2.0.0, but pre-releases in other patch or minor versions are not allowed (e.g., 1.2.3-alpha.4 would be allowed, but 1.2.4-alpha.5 would not). In this paper we ignored such pre-release tags. Taking them into account when computing technical lag is part of future work.

6 Related work

In order to help developers to decide when to use which version of a library, Mileva et al. [13] mined hundreds of open-source projects for their library dependencies, and determined global and important emerging trends in library usage. Decan *et al.* [6, 7] analyzed the evolution of multiple package dependency networks (including npm, CRAN and RubyGems). They studied the presence and impact of package updates on (transitively) dependent packages, and provided an initial exploration of the presence of semantic versioning and dependency constraints. They observed that, for npm and RubyGems, both the proportion of packages and dependencies that declare a minimal dependency constraint is relatively stable through time, but the proportion of packages or dependencies that declare maximal constraints increases over time. The latter suggests that an increasing number of packages relies on maximal constraints to prevent, limit or

control dependency updates. They also observed a high proportion of strict dependency constraints for npm and RubyGems. In September 2016, this accounted for around 15% of all the dependencies in npm. Our work additionally measures the technical lag induced by the use of such dependency constraints in npm.

Kula et al. [11] analyzed 6,374 systems in *Maven* to investigate the latency when adopting the latest library release. They found that maintainers are less likely to adopt the latest releases at the beginning of project; and *Maven* libraries are becoming more inclined to adopt the latest releases when introducing new libraries. In a more recent work [12], they empirically studied library migration that covers over 4,600 GitHub projects and 2,700 library dependencies, and found that 81.5% of the studied systems still keep their outdated dependencies. Surveying the developers, they found that 69% of the interviewees claim that they were unaware of their vulnerable dependencies. In contrast to these works, we analyze the npm ecosystem, where the use of micropackages is prevalent. We do not only measure the number of packages with outdated dependencies, but we quantify the technical lag incurred by outdated dependencies, suggesting the need to monitor such dependencies.

Gonzalez-Barahona et al. [10] proposed for the first time the theoretical model of "technical lag", for measuring how software systems are outdated. In their work, they explored many ways in which technical lag can be implemented and could be used when deciding about upgrading in production. They also presented some specific cases in which the evolution of technical lag is computed. Extending [10], we define technical lag in terms of time lag and version lag and empirically investigate how technical lag is induced by the dependency constraints used by packages at the ecosystem level.

7 Threats to validity

We relied on the libraries.io dataset for our analysis. If the dataset is incomplete (e.g., due to missing package releases), then there is a risk of underestimating technical lag. To migitate this threat, we manual inspected a sample of the dataset, and did not find any major information missing.

We ignored dependencies that use constraints formed by a reference to a URL or a local file. This does not bias our findings, since the proportion of such dependencies is only 2% of all the dependencies in our dataset.

While package dependencies are a major source of software reuse, our empirical study did not differentiate between dependency types (runtime, development and optional), package age or size when studying version or time lag. This means that packages created one year ago or packages with 100 lines of code (LOC) are compared on equal terms with packages created many years ago or containing thousands LOC. If the amount of reused functionality were to be considered, then the dependency type, package size and age should also be taken into account to provide further insight into the factors influencing technical lag.

When choosing the allowed versions for each dependency constraint, we rely on npm semver specifications. This may bias our results, since we found that

in the past, npm semver had faced issues such as considering pre-releases when using the * constraint[4]. However, we are still confident about our analysis since these issues were bugs and not changes in the npm semver specifications.

A final threat to validity concerns the definitions of technical lag that may influence our findings. For example, the time lag is an over-approximation of the actual one because we include the period of time from the latest allowed version until the next release. However, installing the package during that period of time, will fetch the latest allowed release based on the dependency constraint.

8 Conclusion

This paper presented an empirical analysis of package dependency updates in the npm ecosystem, in order to assess technical lag between the deployed version and the latest available version of package dependencies. We found that a large number of packages suffer from technical lag, where their outdated dependencies are several months behind the latest release. We analyzed when patch versions are released and our results suggested that package maintainers should wait a few days before updating to the new available dependency release. These findings can be turned into actionable guidelines about npm dependency updates, and may open the door for more research about technical lag measurements in package libraries.

In future work, we aim to extend our analysis of technical lag by taking into account other issues such as vulnerabilities and bugs. We also aim to carry out similar analyses for other package managers, while considering transitive dependencies, and carry out cross-ecosystem comparisons.

References

1. Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., Shihab, E.: Why do developers use trivial packages? An empirical case study on npm. In: Int'l Symp. Foundations of Software Engineering, pp. 385–395. ACM (2017)
2. Bogart, C., Kästner, C., Herbsleb, J., Thung, F.: How to break an API: Cost negotiation and community values in three software ecosystems. In: Int'l Symp. Foundations of Software Engineering, pp. 109–120. ACM (2016)
3. Borges, H., Valente, M.T., Hora, A., Coelho, J.: On the popularity of GitHub applications: A preliminary note. arXiv preprint arXiv:1507.00604 (2015)
4. Constantinou, E., Mens, T.: An empirical comparison of developer retention in the RubyGems and npm software ecosystems. Innovations in Systems and Software Engineering **13**(2-3), 101–115 (2017)
5. Cunningham, W.: The wycash portfolio management system. ACM SIGPLAN OOPS Messenger **4**(2), 29–30 (1993)
6. Decan, A., Mens, T., Claes, M.: An empirical comparison of dependency issues in OSS packaging ecosystems. In: SANER, pp. 2–12. IEEE (2017)

[4] https://github.com/npm/node-semver/issues/123

7. Decan, A., Mens, T., Grosjean, P.: An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empirical Software Engineering pp. 1–36 (2018). DOI 10.1007/s10664-017-9589-y
8. Dietrich, J., Jezek, K., Brada, P.: Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In: CSMR-WCRE, pp. 64–73 (2014). DOI 10.1109/CSMR-WCRE.2014.6747226
9. Digkas, G., Lungu, M., Chatzigeorgiou, A., Avgeriou, P.: The evolution of technical debt in the apache ecosystem. In: ECSA, pp. 51–66. Springer (2017). DOI 10.1007/978-3-319-65831-5_4
10. Gonzalez-Barahona, J.M., Sherwood, P., Robles, G., Izquierdo, D.: Technical lag in software compilations: Measuring how outdated a software deployment is. In: IFIP International Conf. on Open Source Systems, pp. 182–192 (2017)
11. Kula, R.G., German, D.M., Ishio, T., Inoue, K.: Trusting a library: A study of the latency to adopt the latest Maven release. In: Int'l Conf. on Software Analysis, Evolution, and Reengineering, pp. 520–524 (2015). DOI 10.1109/SANER.2015.7081869
12. Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K.: Do developers update their library dependencies? Empirical Software Engineering (2017). DOI 10.1007/s10664-017-9521-5. URL https://doi.org/10.1007/s10664-017-9521-5
13. Mileva, Y.M., Dallmeier, V., Burger, M., Zeller, A.: Mining trends of library usage. In: Proceedings of (IWPSE) and (Evol) workshops, pp. 57–62. ACM (2009)
14. Nesbitt, A., Nickolls, B.: Libraries.io open source repository and dependency metadata (2017). DOI 10.5281/zenodo.808273. URL https://doi.org/10.5281/zenodo.808273
15. Raemaekers, S., van Deursen, A., Visser, J.: Semantic versioning versus breaking changes: A study of the Maven repository. In: SCAM, pp. 215–224 (2014). DOI 10.1109/SCAM.2014.30