On the Topology of Package Dependency Networks

A Comparison of Three Programming Language Ecosystems

Alexandre Decan Tom Mens Maelick Claes
Software Engineering Lab,
University of Mons, Belgium
{first.last}@umons.ac.be

ABSTRACT

Package-based software ecosystems are composed of thousands of interdependent software packages. Many empirical studies have focused on software packages belonging to a single software ecosystem, and suggest to generalise the results to more ecosystems. We claim that such a generalisation is not always possible, because the technical structure of software ecosystems can be very different, even if these ecosystems belong to the same domain. We confirm this claim through a study of three big and popular package-based programming language ecosystems: R's CRAN archive network, Python's PyPI distribution, and JavaScript's NPM package manager. We study and compare the structure of their package dependency graphs and reveal some important differences that may make it difficult to generalise the findings of one ecosystem to another one.

CCS Concepts

•**Software and its engineering** → **Software libraries and repositories;** *Software architectures;*

Keywords

software ecosystem, software distribution, component dependency graph, Python, R, JavaScript

1. INTRODUCTION

Software engineering research has traditionally focused on studying the development and evolution processes of individual software projects. With the omnipresence of the Internet, collaborative open source software development tools have become widely used. This has lead to bigger and more geographically distributed communities of developers, and made it possible to develop more complex software systems. It gave rise to so-called software ecosystems, i.e., "collections of software projects which are developed and evolve together in the same environment" [16].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ECSAW '16, November 28-December 02, 2016, Copenhagen, Denmark
© 2016 ACM. ISBN 978-1-4503-4781-5/16/11...$15.00
DOI: http://dx.doi.org/10.1145/2993412.3003382

An example of ecosystems are software package distributions. These ecosystems are often characterised by numerous dependency relationships between their packages. Many researchers have studied issues related to such dependencies in different ecosystems. Wittern et al. [22] studied the evolution of JavaScript packages in NPM. The R ecosystem has been studied previously [7, 13], and dependencies have been shown as an important cause of errors in R packages both on CRAN and GitHub [8]. Problems in operating system distributions, such as *Debian* [6, 1] and *FreeBSD* [14], have been extensively studied and efficient tools have been proposed to identify and solve these problems. Related to dependencies inside a software ecosystem, Bavota et al. [2] studied the evolution of dependencies in the Apache ecosystem and found that developers were reluctant to upgrade the version of the software they depend upon.

However, very few studies have compared different software ecosystems. Therefore, this paper studies and compares the dependency graphs of three ecosystems surrounding a particular programming language. We show that CRAN, PyPI and NPM have some important differences. We provide insights on the potential cause of these differences and how they impact each ecosystem. These observed differences may make it difficult to generalise the findings of one ecosystem to another.

2. SELECTED ECOSYSTEMS

In the top 10 of most popular programming languages today[1], three languages have increased in popularity compared to one year ago: *Python*, *JavaScript* and *R*. These will be the focus of the current paper. These languages come with an official distribution of software packages, which we will consider as the software ecosystem corresponding to that language. For R, the ecosystem is CRAN, the official distribution of R packages, available on cran.r-project.org. For Python, we have selected the official Python Package index PyPI (see pypi.python.org/pypi) as ecosystem. For JavaScript, we chose its NPM package manager (see npmjs.com). Each of them is big, as summarised in Table 1.

We extracted all package metadata for those three ecosystems, on April 2016 for CRAN and on June 2016 for PyPI and for NPM. As PyPI lacks dependencies data for many packages, we used the ones collected by Gullikson [10] on February 2016.

Figure 1 shows the evolution of the monthly number of new packages. It reveals that the considered ecosystems are

[1]Based on the september 2016 ranking of the PopularitY of Programming Language Index pypl.github.io/PYPL.html.

Table 1: Characteristics of CRAN, PyPI and NPM

Characteristic	CRAN	PyPI	NPM
snapshot date	2016-04-26	2016-02-17	2016-06-28
packages	9,568	56,231	317,159
dependencies	21,698	53,348	728,447
new pkg. in 2015	1,660	17,818	113,613
updates in 2015	8,140	131,072	711,317

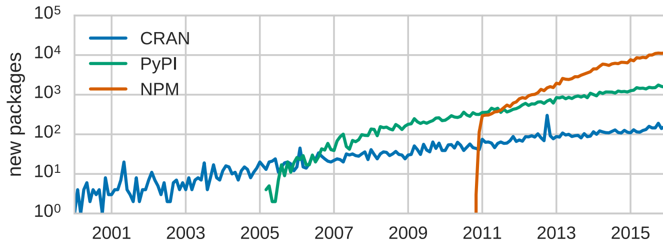

Figure 1: Monthly evolution of the number of new packages.

very active, but with a different rate of growth. To verify this, we performed a linear regression using an ordinary least square method on the logarithmic values of Figure 1. We obtained 0.976, 0.926 and 0.976 as R^2-values for respectively CRAN, PyPI and NPM. We also performed a power law regression but obtained lower R^2-values (resp. 0.897, 0.57 and 0.845). This suggests an exponential increase in the monthly number of new packages for all three ecosystems.

3. PACKAGE DEPENDENCIES

One of the main reasons why dependencies between components emerge in an ecosystem is because of software reuse, a basic principle of software engineering [20]. Software components often rely on (i.e., reuse) the functionality offered by other components (e.g., libraries), rather than reimplementing the same functionality.

In order to assess the extent of reuse in each ecosystem, we computed and analysed their component dependency graphs. Figure 2 shows the proportion of packages (relative to the total number of available packages) that have dependencies, reverse dependencies, or none of both (i.e., isolated components).

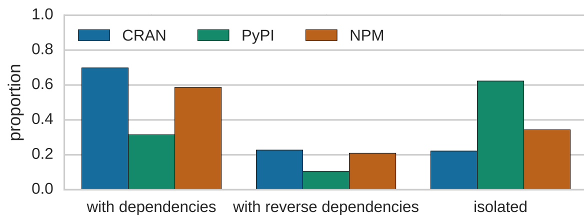

Figure 2: Proportion of packages with dependencies, with reverse dependencies and packages that are isolated

We observe an important difference between PyPI and the other ecosystems. While only 31.5% of all PyPI packages have dependencies, this proportion jumps to more than 58.7% for NPM and to more than 69.8% for CRAN. Similarly, while more than 62.3% of all PyPI packages are isolated, only 34.6% (resp. 22.2%) of the packages on NPM (resp. CRAN) have neither dependencies nor reverse dependencies.

Considering the structure of the component dependency graphs, we computed the set of weakly connected subgraphs for each ecosystem. Figure 3 relates the size of these weakly connected subgraphs to the number of subgraphs having this size. The three ecosystems appear to exhibit a similar relation between the number and the size of weakly connected subgraphs. The main difference comes from the size of the largest subgraphs, proportionally to the size of the ecosystem: NPM's (resp. CRAN's) largest weakly connected subgraph contains 202,417 (resp. 7,318) packages, amounting to around 63.8% (resp. 76.5%) of the ecosystem. In comparison, the largest subgraph for PyPI contains 20,167 packages corresponding to only 35,6% of the ecosystem.

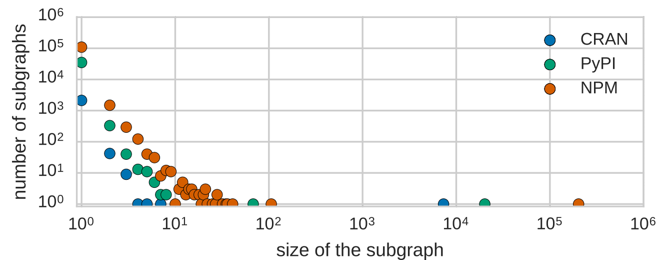

Figure 3: Relation between the size of a weakly connected subgraph and the number of subgraphs having this size.

While dependencies tend to reduce development effort for individual packages, they increase the overall complexity of the ecosystem through the need to manage these dependencies. This complexity can cause many maintainability issues and failures in component based software ecosystems [3, 4, 5]. Leek's blog summarises the concern for CRAN: "*one of the best things about the R ecosystem is being able to rely on other packages so that you don't have to write everything from scratch. But there is a hard balance to strike with keeping the dependency list small.*" [15]

For each package in CRAN, PyPI and NPM, we computed the set of direct and transitive dependencies. Figure 4 shows the results obtained for the three ecosystems. Only packages with at least one dependency are reported on this figure. We primarily observe differences between the ecosystems for the number of transitive dependencies.

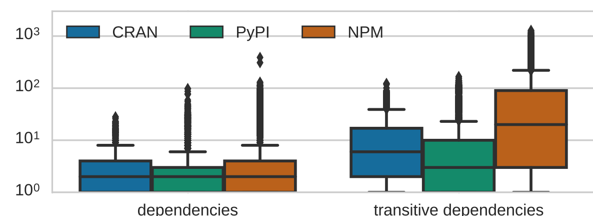

Figure 4: Distribution of the number of (transitive) dependencies by package, for packages having at least one dependency.

We statistically compared the number (≥ 1) of dependencies and transitive dependencies for packages in CRAN,

PyPI and NPM using a one-sided non-parametric Mann-Whitney's U test. We verified the null hypothesis that the distribution of both populations are equal. The alternative hypothesis was that the distribution of one population is greater than the distribution of the other population. The null hypothesis was significantly rejected, with p-values reported in Table 2.

As there is a huge difference in the number of packages having at least one dependency in each ecosystem (6,682 in CRAN versus 17,723 in PyPI and 186,027 in NPM), we report in Table 2 the effect size of Mann-Whitney's U tests using Cohen's d (using a pooled standard deviation) and Cliff's δ. Letters N, S, M and L stand respectively for a *negligible*, *small*, *medium* and *large* effect size, as usually interpreted [19].

Table 2: Statistical comparison of the distributions of the number (≥ 1) of (a) dependencies and (b) transitive dependencies

alt. hypothesis	p-value	Cohen's d	Cliff's δ
(a) PyPI < CRAN	$< 10^{-6}$	0.06 (N)	0.17 (S)
(a) PyPI < NPM	$< 10^{-6}$	0.18 (S)	0.18 (S)
(a) CRAN < NPM	0.001	0.13 (S)	0.02 (N)
(b) PyPI < CRAN	$< 10^{-6}$	0.07 (N)	0.20 (S)
(b) PyPI < NPM	$< 10^{-6}$	0.57 (L)	0.45 (M)
(b) CRAN < NPM	$< 10^{-6}$	0.55 (L)	0.33 (M)

Observed differenced between the three ecosystems may be explained by the completeness of the standard library in R, Python and JavaScript. Python comes with *batteries included*: its standard library typically contains a wide variety of packages that may be used to address many common tasks and problems. But this is not the case for R or JavaScript, forcing developers to rely more often on third-party packages. While R contains an extensive library to manipulate data and conduct statistical analyses, it might lack more generic programming tools that are provided by packages such as *XML*, *jsonlite*, *stringr*, *Rcpp* or *RCurl*. According to JavaScript creator Brendan Eich, the standard library that exists in JavaScript is kept intentionnaly small: "*The real standard library people want is more like what you find in Python or Ruby, and it's more batteries included, feature complete, and that is not in JavaScript. That's in the NPM world or the larger world.*" [12]

The high number of required packages in NPM can also be explained by the desire to distribute micropackages (pushing the *single-responsability* principle to the package level) and metapackages (packages whose features are provided exclusively by dependencies): "*In a lot of JavaScript environments, space is at a premium. [...] Several larger libraries like Underscore (and Lodash) have actually intentionally split themselves into sub-modules because people usually only ever load them to use a single merge function.*" [11]

4. DEPENDENT PACKAGES

Bavota et al. [3] highlighted that dependencies have an exponential growth and must be taken care of by developers. When a change occurs in a package, this change may break its dependent packages, the dependent packages of their dependent packages, and so on. Haney confirms this in his blog: "*Dealing with the deeply nested dependencies has caused us no end of frustrations. A dependency of a dependency of a dependency breaks and we're left trying to trace the source of the error and figure out which repo to open an issue on.*" [11]

Robbes et al. [18] studied the ripple effect of API method deprecation and revealed that API changes can have a large impact on the system and remain undetected for a long time after the initial change. In the context of package-based ecosystems, Di Cosmo et al. [9] highlighted peculiarities of package upgrades and discussed that current techniques are not sufficient to overcome failures. Interestingly, CRAN already suffered from such an issue: "*One recent example was the forced roll-back of the ggplot2 update to version 0.9.0, because the introduced changes caused several other packages to break.*" [17]

In [8] we studied maintainability issues in CRAN related to inter-repository dependencies between GitHub and CRAN. While we are not aware of such problems in PyPI, the recent removal of left-pad from NPM had huge consequences on this ecosystem: "*This impacted many thousands of projects. [...] We began observing hundreds of failures per minute, as dependent projects – and their dependents, and their dependents... – all failed when requesting the now-unpublished package.*" [21]

For each package in each ecosystem, we counted the number of dependent packages, including the transitive ones. The results are shown in Figure 5.

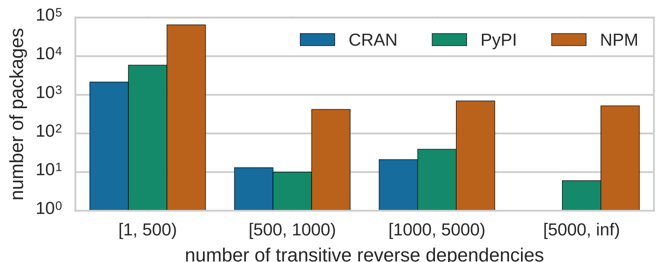

Figure 5: Number of packages by (range of) number of dependent packages

We observe that most packages have fewer than 500 dependent packages. While CRAN (resp. PyPI) only has 34 (resp. 55) packages with more than 500 dependent packages, we observe that there are 1,636 NPM packages with more than 500 dependent packages. This includes 420 packages having between 500 and 1,000 dependent packages, 696 packages between 1,000 and 5,000 dependent packages and 520 packages having more than 5,000 dependent packages.

While we expected NPM to have more packages with a high number of dependents, due to the size of this ecosystem, we didn't expect to see such a big difference between NPM and the other ecosystems. We identified some of these packages: inherits (91K), lodash (89K), isarray (77K), util-deprecate (72K), ansi-regex (69K), strip-ansi (69K), core-util-is (69K), string_decoder (68K), ... The list includes many micro-packages with very few lines of code but a huge number of dependent packages. For instance, inherits, isarray, ansi-regex and strip-ansi all have fewer than 10 lines of code, and have more than 60K dependent packages.

5. CONCLUSIONS AND FUTURE WORK

This empirical study compared the component dependency graph of three popular programming language ecosystems.

We found differences in their structure and related these differences to the specificities of each ecosystem. Most PyPI packages are isolated in the dependency graph because they only depend on the rather extensive standard library of Python. While R, as a statistical language, also comes with an extensive library for analysing data, it lacks general purpose functions which are provided by packages. Moreover, there are many popular packages that extend and improve the basic R features. The consequences are that there are much less isolated CRAN packages than in PyPI. Because JavaScript lacks a proper standard library, most NPM packages are connected between them. Moreover, many popular packages are very small and provide basic functions that are missing from JavaScript's standard library. In particular, this micro-package phenomenon might cause major problems in the ecosystem when one of those micro-package breaks or is removed.

These findings reveal that conducting an empirical analysis on software ecosystems may give different results depending on the ecosystem under study. Results are therefore not necessarily generalisable because of the specifics of each ecosystem. Thus, further studies spanning and comparing multiple ecosystems are required to assess which findings can be generalised across ecosystems.

Our study can be extended in many ways. We would like to study how the ecosystem dependency graphs evolve over time. Another direction is to consider more ecosystems, e.g., RubyGems for the Ruby language and Maven for the Java language. We expect to observe differences between statistically and dynamically typed languages. Moreover, extending the study to compare ecosystems that are not centered around a programming language might reveal other differences. For example, Linux distributions will probably have few packages that are not weakly connected to all other packages because of packages such as *libc6* that are transitively required by most packages.

Acknowledgements. This research was carried out in the context of ARC research project AUWB-12/17-UMONS-3.

6. REFERENCES

[1] P. Abate, R. Di Cosmo, L. Gesbert, F. L. Fessant, R. Treinen, and S. Zacchiroli. Mining component repositories for installability issues. In *Int'l Conf. Mining Software Repositories*, pages 24–33, 2015.

[2] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: the case of Apache. In *Int'l Conf. Software Maintenance*, 2013.

[3] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, 2015.

[4] C. Bogart, C. Kästner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Automated Software Engineering Workshop*, pages 86–89, Nov. 2015.

[5] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Int'l Symp. Foundations of Software Engineering*, 2016.

[6] M. Claes, T. Mens, R. D. Cosmo, and J. Vouillon. A historical analysis of Debian package incompatibilities. In *Int'l Conf. Mining Software Repositories*, pages 212–223, 2015.

[7] A. Decan, T. Mens, M. Claes, and P. Grosjean. On the development and distribution of R packages: An empirical analysis of the R ecosystem. In *European Conference on Software Architecture Workshops*, pages 41:1–41:6, 2015.

[8] A. Decan, T. Mens, M. Claes, and P. Grosjean. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In *Int'l Conf. Software Analysis, Evolution, and Reengineering*, pages 493–504. IEEE, Mar. 2016.

[9] R. Di Cosmo, S. Zacchiroli, and P. Trezentos. Package upgrades in FOSS distributions: details and challenges. *CoRR*, abs/0902.1610, 2009.

[10] K. Gullikson. Python dependency analysis – adventures of the datastronomer. http://kgullikson88.github.io/blog/pypi-analysis.html, February 2016.

[11] D. Haney. NPM & left-pad: Have we forgotten how to program? http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/, March 2016.

[12] Z. Hemel. Javascript: A language in search of a standard library and module system. http://zef.me/blog/2856/javascript-a-language-in-search-of-a-standard-library-and-module-system, February 2010.

[13] K. Hornik. Are there too many R packages? *Austrian Journal of Statistics*, 41(1):59–66, 2012.

[14] N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[15] J. Leek. How I decide when to trust an R package. http://simplystatistics.org/?p=4409, November 2015.

[16] M. Lungu. Towards reverse engineering software ecosystems. In *Int'l Conf. Software Maintenance*, pages 428–431, 2008.

[17] J. Ooms. Possible directions for improving dependency versioning in R. *R Journal*, 5(1):197–206, June 2013.

[18] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to API deprecation? the case of a Smalltalk ecosystem. In *Int'l Symp. Foundations of Software Engineering*. ACM , 2012.

[19] J. Romano, J. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In *annual meeting of the Florida Association of Institutional Research*, pages 1–3, 2006.

[20] J. Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[21] I. Z. Schlueter. The npm blog: kik, left-pad, and npm. http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm, March 2016.

[22] E. Wittern, P. Suter, and S. Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Int'l Conf. Mining Software Repositories*, pages 351–361. ACM, 2016.