Dependency version evolution in the NPM ecosystem

Stanley Scott Henry

CPSC 448 - Directed Study
Supervisors: Gail Murphy & Thomas Fritz
University of British Columbia

1. INTRODUCTION

Within the software industry, dependence on third-party software packages has become a standard practice. [1] These open source software libraries exist for a variety of languages, forcing systems to rely on different libraries such as JUNIT, NPM and Maven. Overtime these libraries grow and evolve, often relying on packages within the library to support future growth. This process underscores the importance of evaluating package evolution over time. The evaluation is especially pertinent to system maintainers who often must decide on when to upgrade, deprecate or transition to a new package. [2]

For system maintainers complex package dependency chains can be problematic. As brought to light in 2016 when the developer of left-pad - a node package that pads out the left hand side of strings with zeros or spaces - was removed from NPM, breaking thousands of builds. [3] Nothing could have predicted this incident, but others may be avoiding by attempting to understand the stability of a package. For the purpose of this paper, stability refers to the complex decision making process of a systamin mainter to use or update a package. Due to package complexity, it is important to evaluate the entire ecosystem that the package exists in.

The term software ecosystem was first defined by Messerschmitt to refer to “a collection of software products that have some given degree of symbiotic relationships.” [4] Mens extended this definition by taking into account the collaborative and social aspects, considering the communities involved as also being apart of the ecosystem. [5] A factor of stability is not only how frequently a package is updated, but how the update is adopted throughout the ecosystem. Packages will often specify the version number, or the minimum version required of a dependent package. Taking an empirical approach to evaluate dependency version evolution I consider two exploratory research questions:

RQ1 How long does it take to update package dependencies?

RQ2 What factors motivate dependency updates?

To address the above questions, I studied 530,793 packages in the NPM registry. NPM is a package manager for JavaScript packages. All public and paid packages are stored in a database called the NPM registry which contains metadata about the package and its dependencies. Using network analysis I evaluated package versions quarterly to understand their relative influence and importance on the ecosystem. Using empirical analysis I will explore five case studies of popular packages to identify package factors that may have motivated the updates.

2. RELATED WORK

Studies have often evaluated changes in software development practices in relation to open source software. Many have found it helpful to evaluate them in the context of software ecosystems. [4, 5]

Raemaekers et al. focus on library stability, identifying code metrics to determine if a library should be depended on. [7] Kula et al. propose two visualization tools for system maintainers to help them navigate library centric dependency chains. [8] Palyart et al. study socio-technical interactions of open source component use finding that interactions start occuring once a component has been chosen to be relied upon. [9] Very few of these studies consider their analysis in a time series format, evaluating the evolution of their metrics overtime.

3. APPROACH

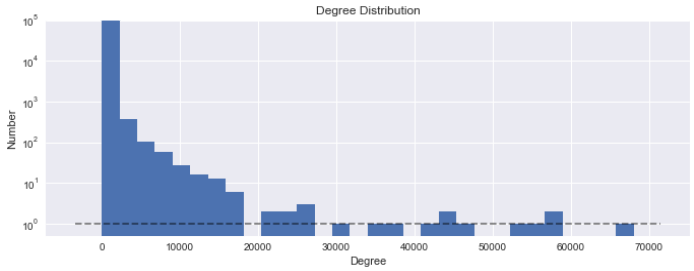
This section describes the methodology of extracting the metadata from the NPM registry. To investigate the research question, I needed to determine when a technical dependency occurred between two packages. When a package is uploaded or updated in the registry, it specifies the required technical dependencies and minimum version. 18,143,784 technical dependencies were identified from 530,793 packages within the NPM registry. 1,486,610 dependencies were excluded from the analysis because they contained incomplete version or package information. A sample of the data extracted for several versions of *csv* and *gulp* is provided below.

Package	Dependency	Package	Dependency
csv@0.4.2	csv-generate@0.0.4	gulp@0.0.7	event-stream@3.0.15
csv@0.4.2	csv-parse@0.1.1	gulp@0.0.7	glob-stream@0.0.3
csv@0.4.2	csv-stringify@0.0.6	gulp@0.0.7	mkdirp@0.3.5
csv@0.4.2	stream-transform@0.0.7	gulp@0.0.7	optimist@0.6.0
csv@0.4.4	csv-generate@0.0.6	gulp@0.0.8	event-stream@3.0.15
csv@0.4.4	csv-parse@0.1.3	gulp@0.0.8	glob-stream@0.0.3
csv@0.4.4	csv-stringify@0.0.8	gulp@0.0.8	gulp-util@*
csv@0.4.4	stream-transform@0.0.9	gulp@0.0.8	mkdirp@0.3.5

Dependency versions can be specified by indicating any version is acceptable(*), indicating the minimum version (^3.0.15) or specifying the specific version number (3.0.15). About 80% of the time, a minimum version was indicated. For simplicity in constructing the network graphs, the minimum version specified was used as the dependency version. For example, if *gulp* indicated the version for event-stream was anything greater than ^3.0.15, it was accepted to be 3.0.15. Identifying ecosystems shifts is more effective when analyzing the minimum version because an update to the minimum version required often signifies larger changes in the source package. Packages that specified dependencies requiring any version were still included, but due to the centrality calculations hold little relevance in the overall analysis.

4. CASE STUDIES

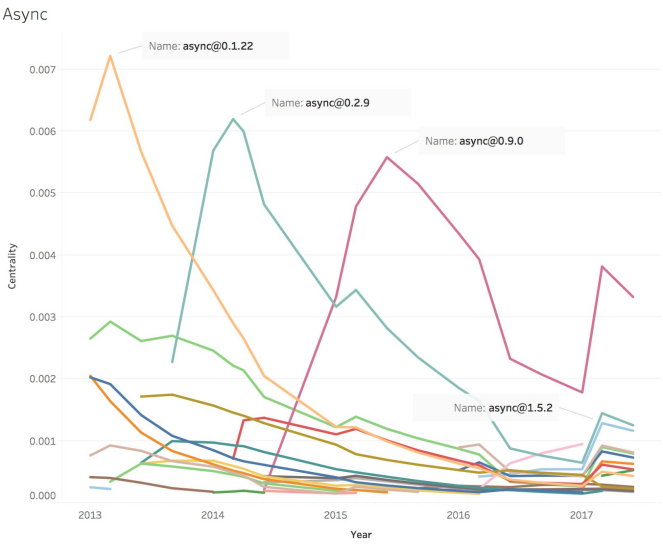
To answer each research question, I investigated 5 popular packages describing both the quantitative and qualitative analysis. I analyzed the eigenvector centrality of these packages by assembling a directed network graph quarterly from 2014 onwards. The vast majority of packages had a degree between 1 to 10. The degree of a package is calculated by adding up the in-degree (packages that depend on this package) and out-degree (number of packages that the package depends on). A graph showing the log distribution of the package degrees is provided below.



Pagerank Centrality (a variation of Eigenvector) is a measure of influence that a node has on a network. It counts the number and quality of edges of a node to determine an estimate of the nodes influence. High-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes. [15] This centrality was chosen in this analysis to take into account the importance of a package to the ecosystem. It matters significantly more if a package with an extremely high centrality updates one of it's technical dependencies due to the impact on the ecosystem. Further, we care more about the adoption rate of updating the dependency version if the package is important to the ecosystem. Both impact and importance is captured through using this centrality measure.

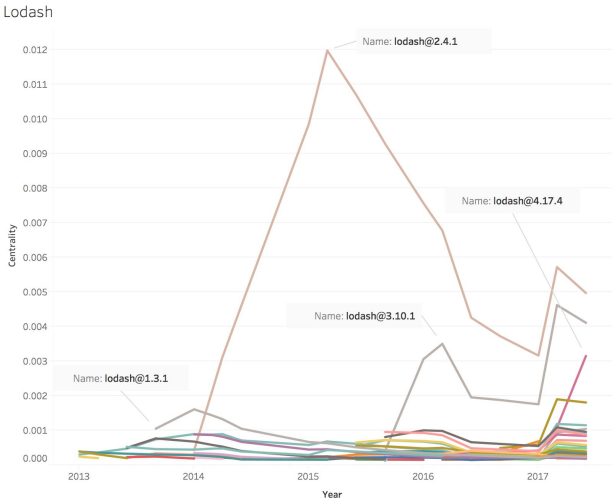
Async

Async is a utility module which provides functions for working with asynchronous JavaScript. [10] Async has an extremely high centrality within the ecosystem which means many important packages depend on async. It takes about a year for one version to gain a higher centrality over another. A change log was only implemented after version v1.0.0, so there is little visibility on the major changes. The long update tail is a strong indicator that new versions do not significantly modify existing methods.



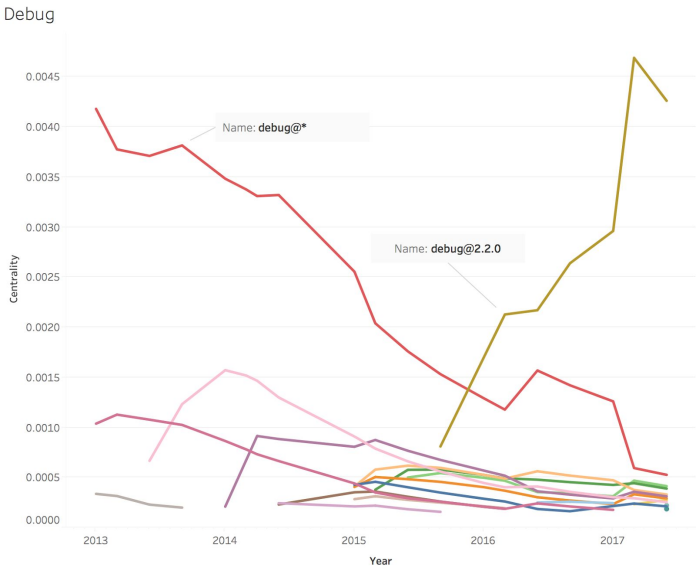
Lodash

Lodash is a JavaScript utility library that focuses on improving modularity and performance of arrays, tests and composite functions. [11] Version 3.10.1 was the stable version of v3 of Lodash which introduced several new utility functions and as well as a guide to help developers migrate their code to v3. This was the largest update since the packages inception. With the introduction of new features in v3 and v4 several breaking changes were outlined in the release log. This could explain the faster migration than normal to the newer versions.



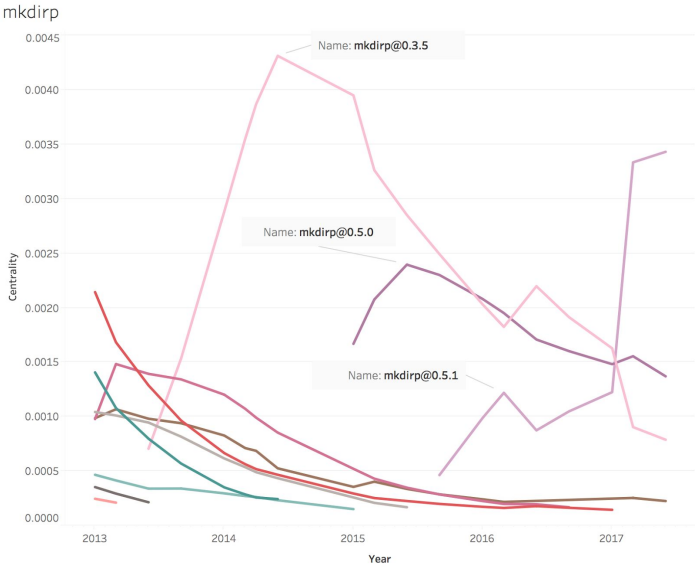
Debug

Debug is another utility function that exposes a function allowing a module name to passed to it, returning a decorated version of the error statement. [12] Debug is an interesting case because the package has only 1 out-degree, but an incredibly high amount of important in-degree packages. This explains the prominence of any version being required until late 2015. Version 2.2.0 was released in mid 2015, introducing two new methods. No breaking changes were completing but this new functionality was enough to entice the ecosystem to update to the new version in only a few months.



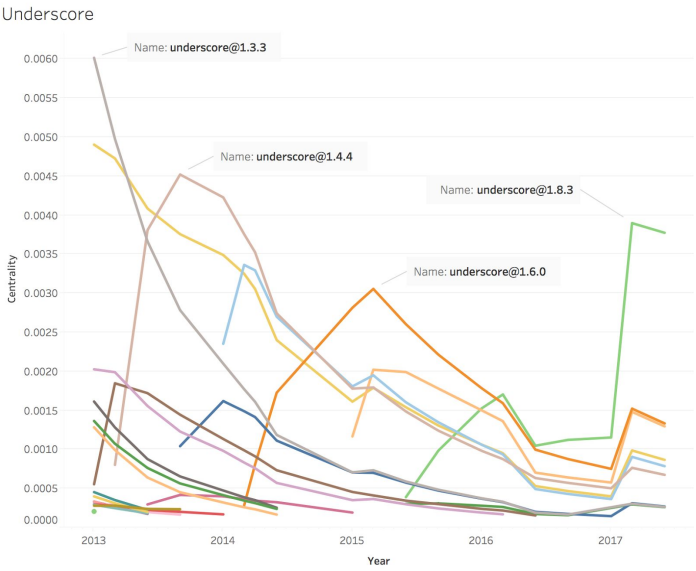
mkdirp

mkdirp is utility function that creates a new directory and any necessary subdirectories at dir with octal permission string. [13] Version 0.3.5 was an extremely large release fixing many bugs and adding multiple test cases. Version 0.5.0 had similar features but had not included travis CI integration. Published only hours later 0.5.1 fixed the integration issues. It's clear this package is of particular importance to developers in the ecosystem due to their quick adoption rate of the new version. It could also underscore the importance of Travis CI to these developers.



Underscore

Underscore is the largest JavaScript utility library that provides functions without extending any built-in objects. [14] Version 1.8.3 is the most current version that was released in April of 2015, no further updates have been made to the package. The prominence of many different versions signifies that developers closely follow these package updates, constantly switching to newer minimum versions. Most releases add additional utility functions, that earlier versions do not have. So if a package uses a new function, it must update the dependencies to exclude older versions. Users are thereby motivated to stay up to date with the latest version to have access to more functionality. The overall downward trend in the centrality signifies that there are packages beginning to hold greater importance than underscore in the ecosystem.



5. LIMITATIONS

The validity of the results presented in this paper is exposed to several threats. *External Validity:* Only the NPM ecosystem was sampled, with over a million nodes excluded. A broader analysis across ecosystems with cleaner data should be conducted. *Internal Validity:* Due to the empirical nature of the case studies, drawing a relationship between centrality and developer motivations is difficult. In future, developer motivations should be analyzed alongside the centrality. *Construct Validity:* I only investigated the minimum version stipulated for the dependency. In most cases, these packages would be using the maximum, stable release version as assigned by the package manager. Using the minimum version only tells one story about the dependency chains. Another analysis should be conducted alongside seeking to understand the implications of current max versions used.

6. CONCLUSION

Through these case studies I have endeavored to answer how long it takes and what factors motivate developers to update their package dependencies. We can see that pagerank centrality is a useful metric to determine if one version has gained more importance over the other. Through taking into

account the influence of related nodes in the network, centrality does not focus solely on the number of packages, but the packages relative influence in the network. Three key trends can be identified from these case studies. (1) Packages that add new methods in each version release (Underscore) will more likely see faster adoption times and more specific minimum version control. (2) Packages will update almost immediately if they need access to an external integration such as Travis CI in mkdir. (3) Depending on the scale and importance of updates (async, debug) packages that are generally pretty stable typically take greater than a year to intersect in centrality of adoption rates.

These conclusions begin to start forming metrics around determining the stability of a package. More research evaluating the evolution of ecosystem metrics must be explored. Only through analyzing historical trends of packages can an accurate measure of stability begin to form.

REFERENCES

1. C. Ebert. (2008). Open source software in industry. *IEEE Software*.
2. R. G. Kula, C. D. Roover, D. German, T. Ishio and K. Inoue. (2014). Visualizing the Evolution of Systems and Their Library Dependencies. *2014 Second IEEE Working Conference on Software Visualization*. Victoria, BC.
3. C. Williams. (2016). How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript. The Register. https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/
4. Messerschmitt, D. G., & Szyperski, C. (2005). Software ecosystem: understanding an indispensable technology and industry. MIT Press Books, 1.
5. Mens, T., Claes, M., Grosjean, P., & Serebrenik, A. (2014). Studying evolving software ecosystems based on ecological models. In Evolving Software Systems (pp. 297-326). Springer Berlin Heidelberg.
6. P. C. Rigby, B. Cleary, F. Painchaud, M. D. Storey, and D. M. German. (2012). Contemporary peer review in action: Lessons from open source development. *IEEE Software*.
7. Raemaekers, S., van Deursen, A., & Visser, J. (2012, September). Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on* (pp. 378-387). IEEE.
8. Kula, R. G., De Roover, C., German, D., Ishio, T., & Inoue, K. (2014, September). Visualizing the evolution of systems and their library dependencies. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on* (pp. 127-136). IEEE.
9. Palyart, M., Murphy, G. C., & Masrani, V. (2017). A Study of Social Interactions in Open Source Component Use. *IEEE Transactions on Software Engineering*.
10. npmjs.com/package/async
11. lodash.com
12. npmjs.com/package/debug
13. npmjs.com/package/mkdirp
14. Underscorejs.org
15. Bonacich, P. (2007). Some unique properties of eigenvector centrality. *Social networks*, *29*(4), 555-564.