UNIVERSITY OF TARTU Institute of Computer Science
Software Engineering Curriculum

Karina Sein

Analysis of dependency graphs of third party libraries in different package managers

Master's Thesis (30 ECTS)

Supervisor(s):
Kristiina Rahkema

Tartu 2021

Analysis of dependency graphs of third party libraries in different package managers

Abstract:

Code development can be a complicated and lengthy process. One option to speed up development is to use third party packages. Third party libraries are generally managed with a package manager. Considering that libraries can use other libraries to build their own functionalities, then the dependency graphs for third party libraries can get extensive. Mistakes or security issues in one library can also affect other libraries in which it is used. Package managers have already been analysed from many different perspectives. One popular dataset is the libraries.io dataset. Previous research using the 2017 libraries.io dataset concluded that package manager ecosystems grow either linearly or exponentially.

The goal of this thesis is to use the newer 2020 version of the libraries.io dataset to analyse growth trends in the number of libraries, versions and dependencies for each package manager. The results are then compared with previous research. It was discovered that not all package managers showed growing trends. Of the selected package managers, three were slowing down istead. In one case this trend was due to incorrect data. Additionally, experiences and potential problems with using the popular libraries.io dataset are described.

Keywords:

Dependency networks, package managers, libraries.io

CERCS: P170 Computer science, numerical analysis, systems, control

Erinevate paketihaldurite teekide sõltuvus graafide analüüsimine

Lühikokkuvõte:

Koodi väljatöötamine võib olla keeruline ja pikk protsess. Üks võimalus arenduse kiirendamiseks on kasutada kolmanda osapoole teeke. Kolmandate osapoolte teeke hallatakse üldjuhul paketihalduriga. Arvestades, et teegid saavad oma funktsioonide loomiseks kasutada teisi teeke, võivad kolmandate osapoolte teekide sõltuvusgraafikud olla ulatuslikud. Vead või turvaprobleemid ühes teegis võivad mõjutada ka teisi teeke, milles seda kasutatakse. Paketihaldureid on juba analüüsitud paljudest erinevatest vaatenurkadest. Üks populaarne andmestik on libraries.io andmestik. Varasemad uuringud, milles kasutati 2017-nda aasta libraries.io andmestikku, jõudsid järeldusele, et paketihalduri ökosüsteemid kasvavad kas lineaarselt või eksponentsiaalselt.

Selle lõputöö eesmärk on kasutada libraries.io andmestiku uuemat 2020. aasta versiooni, et analüüsida iga paketihalduri teekide, versioonide ja sõltuvuste kasvutrende. Seejärel võrreldakse tulemusi varasemate uuringutega. Avastati, et mitte kõik paketihaldurid ei näidanud kasvutendentsi. Valitud paketihalduritest aeglustusid kolm. Ühel juhul oli see suundumus tingitud valedest andmetest. Lisaks kirjeldatakse populaarse libraries.io andmestiku kasutamise kogemusi ja võimalikke probleeme.

Võtmesõnad:

Sõltuvuse graafid, paketihaldurid, libraries.io

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

Introduction

Code development can be a complicated and lengthy process. One option to speed up development is to use third party packages. Third party packages or libraries are software components which are meant to be reused in multiple different projects and are not developed by the vendor of the platform of development [1]. Each library has its own purpose. For example, the NumPy library for the Python programming language provides the developer with a wide array of functions meant to aid in scientific computing such as mathematical functions, support for large multidimensional array objects, linear algebra routines, Fourier transforms, and many more [2] [3]. For the Java programming language an example to bring out could be Jsoup, a library for working with HTML [4]. These are just a couple of examples of useful libraries developed by third parties.

Third party libraries are generally managed with a package manager which is a collection of software tools that help simplify installing, configuring, updating and deleting libraries [5]. Each programming language has its own package managers. For JavaScript npm, pnpm and Yarn are popular options to use [6]. Python has Pip, Anaconda and many other alternatives to choose from [7].

Considering that libraries can use other libraries to build their own functionalities, then the dependency graphs for third party libraries can get extensive. Mistakes or security issues in one library can also affect other libraries in which it is used. Therefore, a project which does not make use of a library with a security issue may still be affected by it through another library which does. Analysing the dependency graphs for third party libraries can show how big the chances of that happening are and how libraries affect each other in general. This information may prove useful for anyone who uses third party libraries in their projects but also to those who develop the package managers or their associated tools.

Package managers have already been analysed from many different perspectives. There have been research papers that have focused only on one package manager but there have also been projects that analyse and compare many package manager ecosystems. While works focusing on one package manager give great insight into the details of the observed package manager, the comparative works highlight the differences between them. For example, there have been research papers on technical lag and popularity metrics for specific ecosystems [8] [9]. On the other hand, comparative research has analysed the growth of various package manager ecosystems while noting that conclusions from one ecosystem cannot necessarily be applied to another [10] [11]. One source of information about package managers is the libraries.io website [12]. It contains databases from different years with information on various package managers but also detailed documentation about the data. The databases are hosted on Zenodo [13].

The goal of this project is to analyse the dependency graphs of third party libraries and how they differ from one package manager to another.

The following research questions will be addressed:

RQ1: For which package managers can RQs 2 and 3 be answered using the libraries.io database?

It is expected that the dataset will provide detailed information about libraries available from the 32 package managers including data on versions, tags, dependencies and repositories of the libraries. However, not every package manager might have enough information to find an

answer to the next research questions as was the case in the research of Decan et al [10]. This would be the case if the dataset has too limited information for a given package manager to answer the research questions. Based on the available data a selection will be made on the package managers to do further research into. Package managers that have less than a 1000 entries will be excluded. Package managers that don't have connections between libraries or don't have information about previous versions will also be excluded.

RQ2: How do library dependency networks grow over time?

Growth will be analysed by observing how the count of libraries and versions increase over the years. The information for past years will be based on the version history of the libraries which the dataset provides. The library dependency networks that have already been analysed from this perspective are expected to continue showing a similar trend in growth [10]. If there are any new ecosystems that can be added to the comparison, then they are also expected to steadily grow. However, the speed at which the networks grow may differ. If the expectations are not met, then it is necessary to take a closer look into the reasons and see if there have been any incidents or events within that specific package manager.

RQ3: How connected are libraries in different package managers through dependencies?

For this question, monthly snapshots are made for each package manager. This is done by generating a monthly snapshot for each package manager. The snapshot is made for the first day of each month between the first of January 2000 and the first of January 2020. For each of these dates the newest versions of each library that existed up to this date are found. This creates a snapshot for each month that shows how many unique dependencies each package manager had at that point in time.

The thesis consists of 8 chapters. The second chapter described previous works in the field. The third chapter states the goals in detail. The fourth chapter describes the methodology. In the fifth chapter the results are presented. The sixth chapter discusses the findings. The seventh chapter concludes the findings.

2. Related Work

Third party libraries and package managers are widely used by developers. Dependency networks of package managers have been studied from various perspectives. The following subsections cover works analysing one or multiple package manager ecosystems with a separate section dedicated to the security of ecosystems.

2.1 Specific Ecosystems

Works analysing just one package manager ecosystem give great insight into the details of these ecosystems.

Ahmed Zerouali et al. in their research paper titled “An Empirical Analysis of Technical Lag in npm Package Dependencies” delved into technical lag from the perspective of the npm ecosystem [8]. In their research they used data from the libraries.io dataset that was gathered between 2010 and 2017. They found that while between 2011 and 2015 there had been a steady increase in the time it took developers to upgrade to the newest version, from 2015 to 2017 the time had been going down. Overall, the median technical lag was three and a half months. They also presented possible reasons for technical lag such as the new update in a dependency having functionality which is not needed in the library or the update not being backwards compatible. As further work, they proposed to analyse the impact of bugs and vulnerability issues on technical lag as well as to expand the research to more package managers.

In “On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm” Ahmed Zerouali et al. take an in depth look at popularity metrics [9]. They used data from libraries.io, npmjs.com and GitHub from 2018 and extracted 9 unique popularity metrics. While their finding indicated that the majority of the analysed popularity metrics were not strongly correlated, they also found that depending on the used metrics, the results can vary significantly. One of the ways they suggested building on their work was to expand the research to other package managers as they noted that the results of their research cannot be accurately generalised to other package manager ecosystems.

2.2 Analysing and Comparing Multiple Ecosystems

While there are many useful research papers on single ecosystems, it is also important to analyse other ecosystems and compare them to each other to get a more complete overview. This is brought up in “On the topology of package dependency networks: a comparison of three programming language ecosystems” by Alexandre Decan et al. [11]. In their research they concluded that findings based on one package manager ecosystem can not always be generalised to other ecosystems even if they belong to the same domain. They backed the claim up by analysing and comparing CRAN, PyPI and npm. Analysing the library dependency graphs for these ecosystems revealed crucial differences between them which would make accurately generalising attributes from one ecosystem to another difficult. For example, the portion of projects which have dependencies can greatly vary. While 31.5% of PyPI libraries have dependencies, both npm and CRAN have a much higher percentage with 58.7% and 69.8% respectively. Therefore, to get an overview of package managers in general, it is important to analyse many different ecosystems. In their study they suggested

that the research can be extended by looking at more ecosystems or looking at how the dependency graphs evolve over time for each ecosystem. Furthermore, the data used in the research is from 2016 which also gives reason to check if the same conclusions still apply.

There are many research papers which use multiple package manager ecosystems. Jacob Stringer et al. in their paper titled “Technical Lag of Dependencies in Major Package Managers” focused on comparing technical lag between 14 different package manager ecosystems [14]. Technical lag happens when dependencies of a library receive an update but the library lags behind in integrating the new updates. They discovered that while many dependencies lag, they do not lag by a large amount. Furthermore, the presence and severity of technical lag is dependent on the infrastructure and common practices of the package manager ecosystem. In this case as well, the research has been done on outdated data so there would be reason to see whether the same conclusions still apply.

Alexandre Decan et al. analysed seven different package managers based on data from libraries.io [10]. The chosen package managers were Cargo, CPAN, CRAN, npm, NuGet, Packagist and RubyGems. The reasons these were chosen was because they had the necessary data available for the research questions and they had a considerable amount of libraries. They found that all of the observed ecosystems grow overtime but the increase in size and complexity varies from one package manager to another. They discovered that updates are either steady or grow overtime. However, the amount of updates are not distributed equally among the libraries as required or newer libraries are updated significantly more often. They also analysed dependency graphs for the package managers and discovered that depending on other libraries is common across all package manager ecosystems. While libraries become more and more connected over time, the dependencies are not distributed equally as a small portion of libraries make up the majority of reverse dependencies. Furthermore, while libraries usually do not have many direct dependencies, they tend to have a considerable amount of indirect dependencies through the directly used libraries. This research has been done on the basis of data available in 2017. Since then an updated dataset has been released so there is reason to see if the conclusions drawn in the research still stand.

Riivo Kikas et al. analysed the dependency network structure and evolution of the JavaScript, Ruby and Rust ecosystems [15]. Their research showed that all three ecosystems were growing. Of the three, JavaScript was growing the fastest and had the highest amount of transitive dependencies. Even though ecosystems have become less dependent on a single popular package, both JavaScript and Ruby have packages that affect more than 30% of their respective ecosystems.

2.3 Analysing the Security of Ecosystems

Security is an important part of any package manager ecosystem. As such there have been multiple research papers written on the topic.

One such case is “PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities” by Qiang Li et al. in which they develop a tool called PDGraph to analyse the security issues of dependencies [16]. While analysing 337,415 projects they found 1,014 projects which had publicly known vulnerabilities. Furthermore, there were 67,806 projects that depended on the found libraries and only a small percentage of insecure dependencies were fixed over an 8 month period. Since this study was only conducted on the

Maven package manager there is reason to see if other ecosystems show a similar trend of infrequent dependency updates.

Markus Zimmermann et al. also noticed that developers are generally slow in removing vulnerable dependencies [17]. Furthermore, they noted that in the case of the npm package manager, the projects tend to have many dependencies and because of that a single project with a vulnerability is capable of affecting thousands of other projects.

Security issues exist as well on the package manager level. For example, Justin Cappos et al. analysed 10 package managers and found vulnerabilities that could potentially be taken advantage of with a man-in-the-middle attack or a malicious mirror [18]. Ruturaj K. Vaidya et al. focused only on 2 ecosystems: npm and PyPI, and also found that these ecosystems can be vulnerable to malicious attacks.

3 Methodology

There are several goals for this project. The first is to get familiar with the libraries.io dataset and the data that it contains. The next step is to learn how to use the dataset to answer questions about package managers. For this, the first research question is asked: For which package managers can RQs 2 and 3 be answered using the libraries.io database? This question requires learning how to use the dataset, what the structure of the dataset looks like and what kind of data is available.

The second research question is: How do library dependency networks grow over time? For this question the growth of the number of libraries is analysed over time. The same is done for versions as well. In addition, the count of monthly uploads is also analysed for versions. Analysing the growth of libraries and versions provides insight into the growth of the dependency networks. If there are any unusual trends or deviations from the trends that Decan et al [10] observed, then an attempt is made to find the reason.

The final research question is: How connected are libraries in different package managers through dependencies? For this question the number of unique dependencies for each package manager is analysed. Analysing how the count of dependencies for libraries changes over time shows the amount of connections between libraries in each package manager.

The goals of this project will be achieved in several steps. First, data is gathered about package manager ecosystems and prepared for processing. Then, the data is processed and analysed to find answers to the posed research questions.

3.1 Acquiring Data

Libraries.io provides a dataset with 32 package managers [12]. The dataset is published on Zenodo [13]. The latest version of the dataset is chosen which in this case is from the year 2020. The total amount of data contained in the unpacked archive of CSV files was 159 GB.

For processing the data PostgreSQL will be used. Loading the data into PostgreSQL first enables to read the data into the processing program only partially while selecting only relevant information. This can help save space on memory which is critical due to the large size of raw data.

A prerequisite for reading the data into PostgreSQL is creating the tables for each data type. The data types provided by the dataset are described in detail in the libraries.io documentation [12]. The dataset provides CSV files for the following tables: libraries of the 32 supported package managers, library versions, tags, dependencies, repositories, repository dependencies and libraries with related repository fields. Library versions provide data about immutable published versions of libraries. Tags mark specific versions. Repositories show where the code is located. Of these tables, libraries, versions and dependencies are of interest. The libraries tabel will provide information about which package managers the dataset supports and how many libraries each has. The versions table has information about the previous states of the libraries and helps timestamp creation and updating of both libraries and dependencies as neither have data on that themselves. Dependencies table shows how libraries are connected to each other through dependencies.

To create the tables for PostgreSQL, the CREATE TABLE command can be used coupled with the names and types for columns which can be determined either from the documentation or the CSV files. This needs to be done to each of the three tables of interest. Once the tables are set up, they can be filled with data. For the libraries table the following command was used:

COPY projects FROM 'D:/Thesis/libraries-1.6.0-2020-01-12/projects-1.6.0-2020-01-12.csv' DELIMITER ',' CSV HEADER;

However, this command does not work for the versions and dependencies tables as they most likely exceed the maximum size set by PostgreSQL [19]. The advantage of using PostgreSQL is that it simplifies reading in large files because of the possibility of processing the files in chunks. Therefore, a separate program is written to read the data in. The program processes the CSV files in one million line chunks. It takes a million lines, creates a new temporary CSV file for them and appends the entire temporary file into the given PostgreSQL table. After that the temporary file is deleted to save space and the next chunk of lines is processed. This is repeated until all the lines are copied into PostgreSQL. The program is executable from the command line. To stop the program from requiring to enter the PostgreSQL database password on every loop of the program, it is possible to set the password variable in the command line beforehand with the command "set PGPASSWORD=(password)" where (password) is replaced with the actual password. The program takes three inputs: location of the CSV file to read into PostgreSQL, the location to create the temporary files and the name of the table. With this program the versions and dependencies tables were filled with their corresponding data.

Psycopg2 library is used to connect the Python program for analysing with PostgreSQL. The data from PostgreSQL is then transformed into a dataframe with the use of the pandas library for ease of analysing.

3.2 RQ1: For which package managers can RQs 2 and 3 be answered using the libraries.io database?

For analysis, a Jupyter Notebook file will be made. Data will be read in when needed with the Psycopg2 library. This also enables reading in data selectively, which helps with memory management. All further analysis will be done in the Jupyter Notebook file.

For this, the tables for libraries, dependencies and versions are analysed. Firstly, the data needs to be categorised by the package manager so that different package manager ecosystems could be compared in each table. Then the number of entries for each package manager is analysed. If a package manager has less than a 1000 entries in any of the tables, then it is excluded. If a package manager has less than a 1000 entries in the libraries table then that means that there are too few libraries to draw meaningful conclusions. If there are less than 1000 entries in the versions table then that means that there is very little data on the previous states of libraries for that package manager. This is important because the progression of libraries over time is important while answering the posed research questions and the versions table is the place where that information is collected. Finally, the dependencies table is important because it shows how libraries are connected to each other. Since this is also vital information for answering the research questions then this table also cannot have less than 1000 entries to ensure that there is sufficient data to work with. Furthermore, given that Decan et al [10] used a threshold of 4000 libraries, the expectation is

that the threshold of 1000 entries allows for more package managers to be analysed. This results in a list of package managers which fit all the requirements to answer the research questions.

It is expected that all the same package managers as analysed by Decan et al [10] will also pass the selection process in this case as well. Furthermore, since the thresholds have been lowered it is expected that some additional package manager will be added to the comparison.

3.3 Cleaning data

First, any entries in the libraries, versions and dependencies tables are removed if they are from a package manager that has not made it through the filter of having enough entries. This limits the tables to only entries from relevant package managers.

3.3.1 Libraries

After filtering for only relevant libraries the data needs to be checked. It is necessary that each entry is unique so that there would be no duplicates to skew the results. Also 10 entries are chosen at random and are compared with their source material to make sure that the data represents reality accurately.

3.3.2 Versions

For versions, it is important that each row has a “projectid”, so that it would be possible to connect it to the corresponding entry in the libraries table, and a “publishedtime” field by which to timestamp the versions by. Anything before the first of January 2000 is filtered out. This removes unreasonable published times such as 01-01-1900 and other similar dates which are too early to be legitimate. Before the year 2000 there are also valid entries but an observation period of 20 years was deemed reasonable.

3.3.3 Dependencies

For dependencies, it is important to filter out dependencies that seem to suggest that the library depends on itself. That is because the goal is to find out how a library depends on other libraries which does not include the original library itself. A “dependencyprojectid” is needed to connect the dependency to its corresponding library. Each entry needs to have a “versionid” so that it would be possible to put a timestamp on each dependency.

3.4 RQ2: How do library dependency networks grow over time?

For this question it is important to have detailed information about the different versions of libraries. Most importantly, the time of publication for each version and to which package manager ecosystem it belongs. The results from RQ1 will determine which package managers will be included in this question. The methodology to find an answer to this

question will be the same as by Decan et al. for the sake of comparability with their results [10].

Based on the available data the time period over which the growth will be analysed can be determined. When there is a lack of data for a year, then that will be the cutoff point for the analysis. Package managers that appear in the research paper by Decan et al. [10] and fulfil the requirements for the current research will be compared. There will also be an analysis on how the growth has progressed since the previous research and if the same trends still apply. If there are unexpected changes in the growth then there will be a need to delve further into the causes for the changes. Since the 2020 dataset is much larger than the 2017 one, then it might also be possible to add more package managers to the comparison.

First, to compare the growth of package managers, it is analysed how the count of libraries increases on the libraries.io database. In this case the created date of each library is considered. The created date in the libraries table shows when libraries.io has started tracking the library. Then, the libraries are grouped together by their package manager and the created date. After that, the cumulative sum of tracked libraries over time is found. This result will be shown on a graph for ease of analysis.

Second step is to see how the total count of all libraries and all versions has changed over the years. For that, the entries in the libraries table will be grouped together by package manager. For each package manager the cumulative count of libraries is counted for each month. The same counting is also done for versions. Based on the data, a graph will be drawn to give a clear representation of the results.

Finally, the amount of newly published versions per month will be analysed for each package manager. For this, the entries in the versions table are grouped together by package manager and their published date. Then, the total sum of added versions for each month is found for each package manager. A graph will be drawn in this case as well.

It is expected that package managers will show similar growth trends as observed by Decan et al. [10] and that these trends continue. In general, it is expected that both the count of libraries and versions continues to increase over time and that their trend in growth either stays stable or increases. A steady or increasing trend is also expected when analysing monthly uploads of versions.

3.5 RQ3: How connected are libraries in different package managers through dependencies?

To find how connected dependencies are, it is important to have detailed information about the dependencies for the package managers. The only way to timestamp entries in the dependencies table is through the versions table as dependencies do not keep track of the time they were added. Therefore, it is also important that there is detailed information about versions.

This is done by generating a monthly snapshot for each package manager. The snapshot is made for the first day of each month between the first of January 2000 and the first of January 2020. For each of these dates the newest versions of each library that existed up to this date is found. This is done by grouping the entries in the versions table together by their library id, filtertering out versions that were released after the given month, and taking the latest one from each group. Then the dependencies are added to the filtered versions table by

the version id. This creates a snapshot for each month that shows how many unique dependencies each package manager had at that point in time. The results for this process will be shown on a graph.

It is expected that the amount of dependencies keeps growing over time and that the trend of growth will be stable or be increasing. It is expected that package managers analysed by Decan et al. [10] will show the same growth trends and stay stable or increase after the end of their research period.

4. Results

After the data is read into a Jupyter Notebook it is possible to start looking for answers to the posed research questions. This starts with analysing in detail the information that is available in the three relevant tables: libraries, versions and dependencies. Irrelevant or faulty data is filtered out after which the package managers are compared.

The following sections describe the data tables used and provide answers to the three posed research questions.

4.1 Acquiring data

Data is acquired from three tables: libraries, versions and dependencies. Before answering the posed research questions, it is important to analyse in detail the information that these tables provide. Based on this information, a selection is made on which package managers to conduct further research on.

4.1.1 Libraries

The libraries table which is called “projects” in the database has 21 columns which are named “id”, “platform”, “name”, “created”, “updated”, “description”, “keywords”, “homepage”, “licences”, “repourl”, “versioncount”, “sourcerank”, “latestrelease”, “latestreleasenumber”, “packetmanagerid”, “dependentprojectscount”, “language”, “status”, “lastsynced”, “dependentrepositoriescount” and “repositoryid”. Here, the most important column is the “platform” column as it is used to group libraries from each package manager together. This enables analysing each package manager separately and to find the total count of libraries for each as a result. Another important column is the “id” column by which it is possible to merge the libraries table with other tables such as the versions table as long as they have a reference to the library id in them. “Dependentprojectscount” and “dependentrepositoriescount” provide some information about how connected the library is to others. Each row in the “projects” table corresponds to a library.

Figure 1 has a logarithmic scale and shows the list of package managers found in the “projects” table and the corresponding count of libraries for each of them.

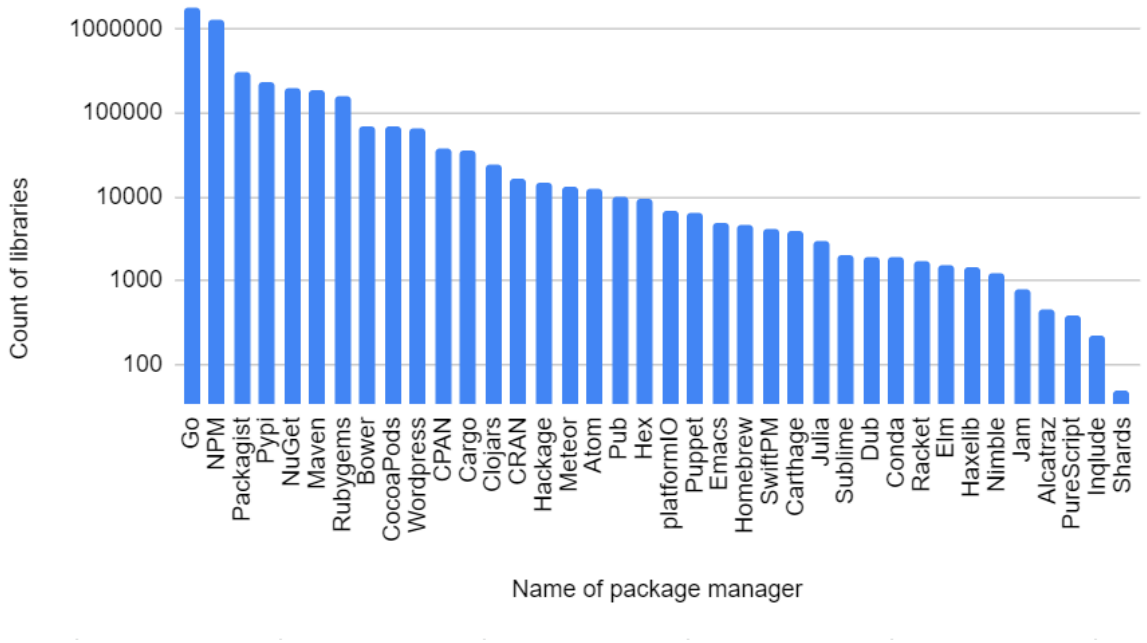

Figure 1: Library count for each package manager

In the table there are 38 different package managers. These package managers are: Go, NPM, Packagist, PyPI, NuGet, Maven, Rubygems, Bower, CocoaPods, Wordpress, CPAN, Cargo, Clojars, CRAN, Hackage, Meteor, Atom, Pub, Hex, platformIO, Puppet, Emacs, Homebrew, SwiftPM, Carthage, Julia, Sublime, Dub, Conda, Racket, Elm, Haxelib, Nimble, Jam, Alcatraz, PureScript, Inqlude and Shards. The Go package manager has the most libraries documented with a total count of 1818666 libraries. On the other hand, Shards has the fewest with only 33 libraries. The count of libraries can greatly vary from one package manager to another.

The libraries.io website claims to support 32 different package managers yet in their list of package managers they have named only 31 [12]. The 7 that are missing from the list are: Shards, JAM, Sublime, Emacs, PlatformIO, Atom and Wordpress. As there is no information about them on the libraries.io page then it is assumed that they are not fully supported by libraries.io. The reason could, for example, be that they are still gathering data on these package managers or that the data they have is faulty. Because of this, these seven package managers will be removed from further comparison. Additionally, any package manager with less than a 1000 entries will be removed. This also removes Inqlude, PureScript and Alcatraz. This leaves 28 package managers.

There were no cases where the id or name of the library was missing. There were no duplicate entries either. About 97.2% of all library names were unique. The most common name was “msgpack” with 15 occurrences. Most of which seemed to be an implementation of the same project but for different package managers. The length of the library name is between 1 character and 249 characters. In the maximum case the name was put as the website on which the code is hosted on.

All together the “projects” table had 4609662 libraries recorded. Of these, 4023790 (~87.3%) had no dependent libraries, 3970330 (~86,1%) had no dependent repositories and 3743497 (~81,2%) had neither. This data was extracted from the “Dependentprojectscount” and “dependentrepositoriescount” respectively.

4.1.2 Versions

The versions table has 8 columns named “id”, “platform”, “projectname”, “projectid”, “versionnumber”, “publishedtime”, “createdtime”, “updatedtime”. Here, the most important columns are “projectid” and “publishedtime”. By “projectid” it is possible to connect a version to the library that it belongs to. “Publishedtime” shows when this version of the library was published. This information is important because it is the only way to timestamp the creation of both versions and libraries. As libraries in the “projects” table do not have a publishing date indicated then this information is obtained from the earliest version the library has. Each row in this table represents a different version for its corresponding library. “Updatedtime” is used to make sure that libraries.io still keeps track of versions from the corresponding package manager.

25 different package managers are represented in this table. Any package managers that are missing from this table are removed from the comparison. These include Go, Bower, SwiftPM, Carthage, Julia, Conda, Racket and Nimble. Furthermore, to get up to date information about each package manager, if the latest update time was not in 2020, then the package manager was also excluded. This removed Elm, Hackage, Homebrew and Meteor. 16 package managers remain by now.

For NuGet out of the 2376827 entries 127705 (~5,4%) have the published time set as first of January 1900. Out of 2792392 entries for Maven, 297 (~0,01%) had the published date as first of January 1970. Both NPM, Packagist and RubyGems had one entry with the published date first of January 1970 out of 10862491, 1687460 and 1048625 respectively. One option was to use the created time in these cases instead. However, created time is not as accurate as it shows when the version was first detected by libraries.io, which may be later than the time of creation. As the portion of faulty data is not too high, it was decided to discard the data instead and only work with credible information. After discarding the faulty data, the earliest publishing date was 23.08.1994 for a Packagist library.

4.1.3 Dependencies

The dependencies table has 12 columns named “id”, “platform”, “projectname”, “projectid”, “versionnumber”, “versionid”, “dependencyname”, “dependencyplatform”, “dependencykind”, “optionaldependency”, “dependencyrequirements” and “dependencyprojectid”. The columns of interest are “platform”, “versionid” and “dependencyprojectid”. “Platform” is important because it holds information about the package manager which the dependency is connected to. “Versionid” is important so that dependencies could be connected with their corresponding version and in extension, the time it was added. “dependencyprojectid” shows which library the dependency belongs to.

Almost all dependencies were between libraries from the same package manager with the exceptions of NuGet which had 14 dependencies from NPM, Packagist which had 44 dependencies from NPM and Pypi which had 1 dependency from NuGet. Packagist had the

most dependencies with a documented count of 8929479. On the other hand, Haxelib had the least with only 7595 dependencies.

Entries that had missing data in any of the three relevant columns were filtered out. Furthermore, the entries that had no “dependencyprojectid” seemed to indicate that the library depended on itself. This is supported by the fact that if there was no “dependencyprojectid” value then in most cases the “projectname” was either partially or fully included in the “dependencyname”. Figure 2 shows how many entries each package manager had and how many were discarded after this point.

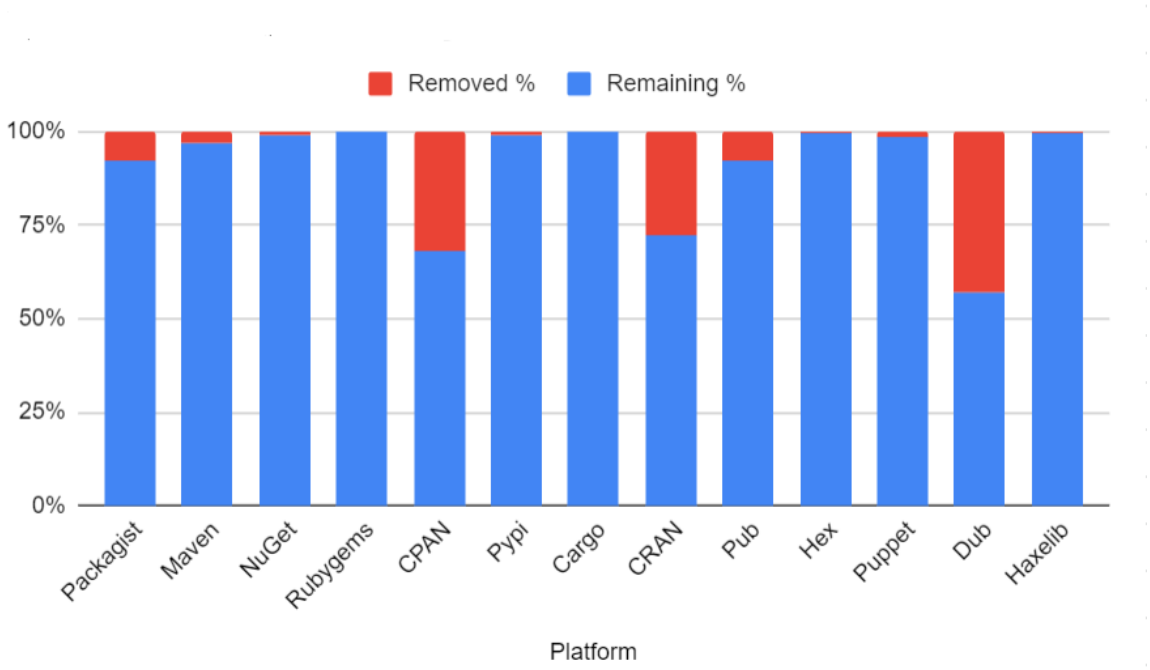

Figure 2: Percentage of removed dependencies

Clojars and CocoaPods had no entries in the table and were removed from the comparison. NPM was also removed as there was not enough memory to read all the entries in. Furthermore, as NPM is a popular package manager it already has a lot of research done on it. After this step, only 13 package managers remain.

4.2 Selected libraries

Not all of the available libraries met the conditions to be used in further analysis. Out of the 38 libraries that were found in the dataset only 13 met the necessary criteria. Before moving onto the research questions some background information about each of them is provided. For each package manager some interesting facts have been brought out.

CPAN (Comprehensive Perl Archive Network) package manager was created in 1993 and has been active online since October 1995 making it the oldest package manager on the list [20]. CPAN is used with the Perl programming language which has been around since

February 1. 1988. Most of the time CPAN libraries additionally include installation scripts and test scripts which ensure the user that everything is running as expected.

CRAN (Comprehensive R Archive Network) was announced April 23. 1997 [21]. It was made for the R programming language which has been around since 1993. CRAN forces use of the newest versions for libraries [22].

Cargo's oldest commit on Github is from March 4. 2014. Cargo is compatible with the Rust programming language which has been around since July 7. 2010 [23]. Rust refers to libraries as "crates" which are managed by the build system and package manager called Cargo that comes with Rust upon installation [23] [24].

Dub was published on October 31. 2012 and is for the D language which is from 2001 [25]. Dub is unique in the way that it uses git tags to determine the newest version of libraries.

Haxelib has been available online since August 23. 2006 [26]. It is targeted for the Haxe programming language which has been around since 2006. It comes by default with each Haxe distribution.

Hex was published on Github on December 23. 2013 [27]. Hex is meant for the Erlang language which is from 1986. Hex has 9452 libraries and ranks 10th by the total number of libraries for the remaining selection.

Maven has been in use since July 13. 2004 and is primarily used together with Java which has been around since May 23. 1995 [28]. Maven libraries have XML-type files called the Project Object Model which has the description, versioning and configuration of the library [29].

NuGet has been around since October 5. 2010 [30]. NuGet is used together with .NET which has been in use since February 14. 2002. A NuGet library is a ZIP file containing code and descriptions for the code [31]. Aside from public hosting, private hosting is also available for NuGet libraries for sensitive code [31].

Packagist was published on Github on June 9. 2011 [32]. Packagist is for the PHP language which has been around since June 8. 1995. Packagist has the most documented libraries on libraries.io out of the remaining package managers. Packagist also supports private hosting of libraries [33].

Pub has been around since October 5. 2011 [34]. Pub is for the Dart programming language which has been around since October 10. 2011. A Dart library has at least a pubspec.yaml file, which describes information about the library [35]. That information can also include a list of dependencies.

Puppet originates from 2005 and includes its own declarative language [36]. Puppet is one of the lesser represented package managers with only 6432 libraries documented on libraries.io

Pypi has been around since 2003 and is for the Python language which is from February 20. 1991 [37]. Some other package managers use Pypi as their main reference for libraries and their dependencies. This includes package managers such as pip for example.

RubyGems was published on Github on February 28. 2003 [38]. RubyGems is used with Ruby which has been around since 1995. RubyGems is included in a Ruby distribution. RubyGems has a considerable amount of data in libraries.io with 161608 recorded libraries.

4.3 Comparison of selected package managers

After suitable package managers were chosen and the data was cleaned, it was possible to analyse the data.

4.3.1 Libraries

First, the count of libraries for each package manager was analysed. Each library in the libraries table has a created date associated with it. This represents the date when libraries.io started tracking the library. The growth of the total count of available libraries for each package manager on libraries.io over time is represented on figure 3.

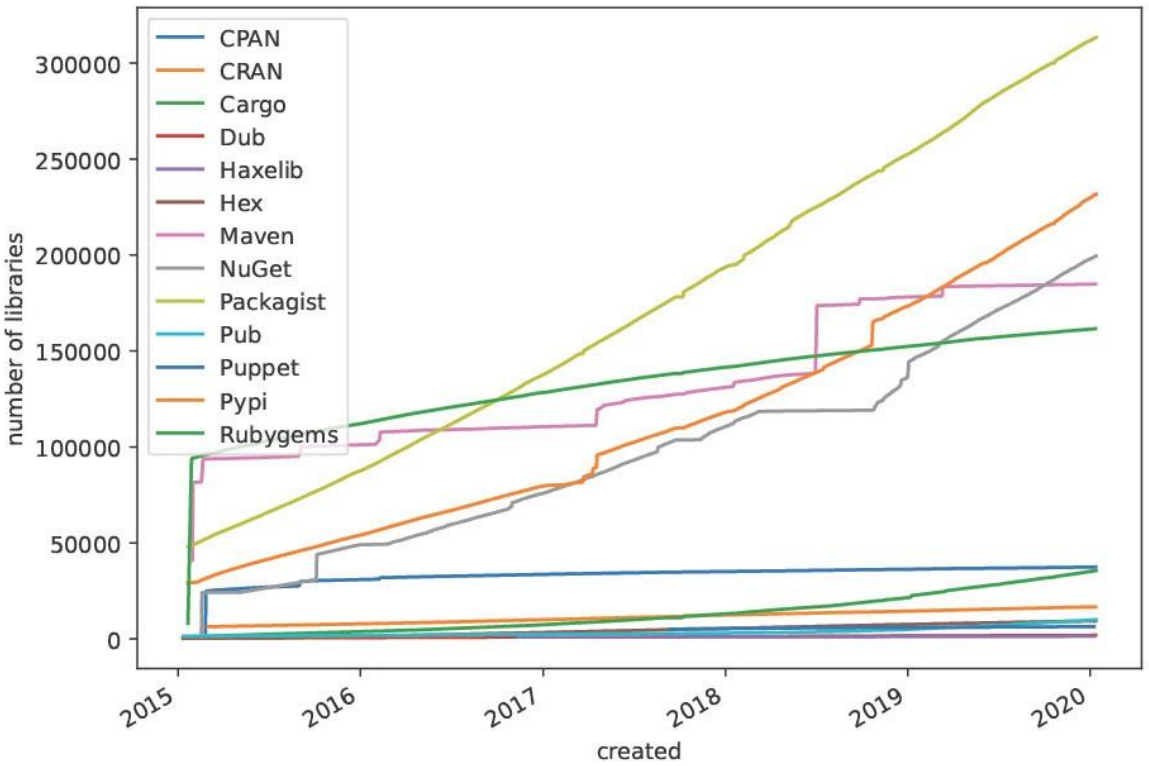

Figure 3: Growth in libraries count over time on libraries.io

The growth of available libraries differs from package manager to package manager. The growth for Packagist, Pypi, Cargo and most smaller package managers is smooth and mostly without major jumps but differs in rate and scale. Both CPAN and Rubygems have a high spike in the beginning but have a smooth growing trend afterwards. NuGet has a couple of jumps in the beginning but has a mostly smooth growing trend until 2018. For the majority of 2018 growth stopped but at the end of the year many new libraries were added. After 2018 the trend slows down and remains steady. Maven on the other hand does not have a smooth growing trend and has many spikes of sudden growth followed by long intervals of times where very few new libraries are added. The last major spike was in 2018 after which there have only been two more smaller spikes.

The publishing dates for libraries were found by connecting the libraries table with the versions table by the library id and then finding the earliest version for each library. The publishing date for the earliest version was taken as the publishing date for the library. Figure 4 shows the growth in total number of libraries for each package manager starting from the year 2000.

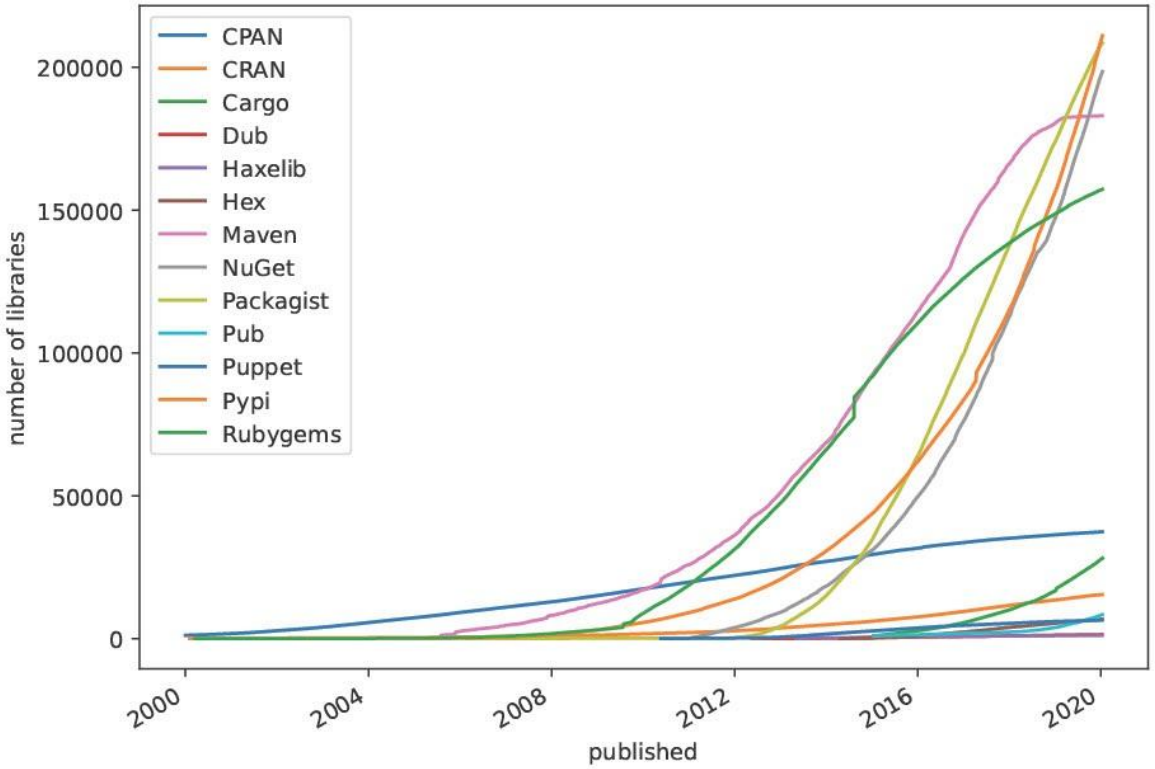

Figure 4: Growth in total libraries count over time

Packagist has a high exponential growth trend which has taken off closely following its release in 2011. Pypi has 231690 libraries in total and also has a high growth trend. Maven shows exponential growth between 2006 and 2016. After this, growth has started to slow down, especially since 2018 and completely stopping by the end of 2019. NuGet has been growing exponentially since its publishing and shows no sign of slowing down. NuGet has accumulated 199447 libraries in total. Rubygems had a strong growth trend since 2009 which starting from 2014 has started to slow down.

To get a clearer overview of package managers with fewer libraries, a separate figure was made where the top 5 package managers in terms of number of libraries were left out. The result is shown on Figure 5.

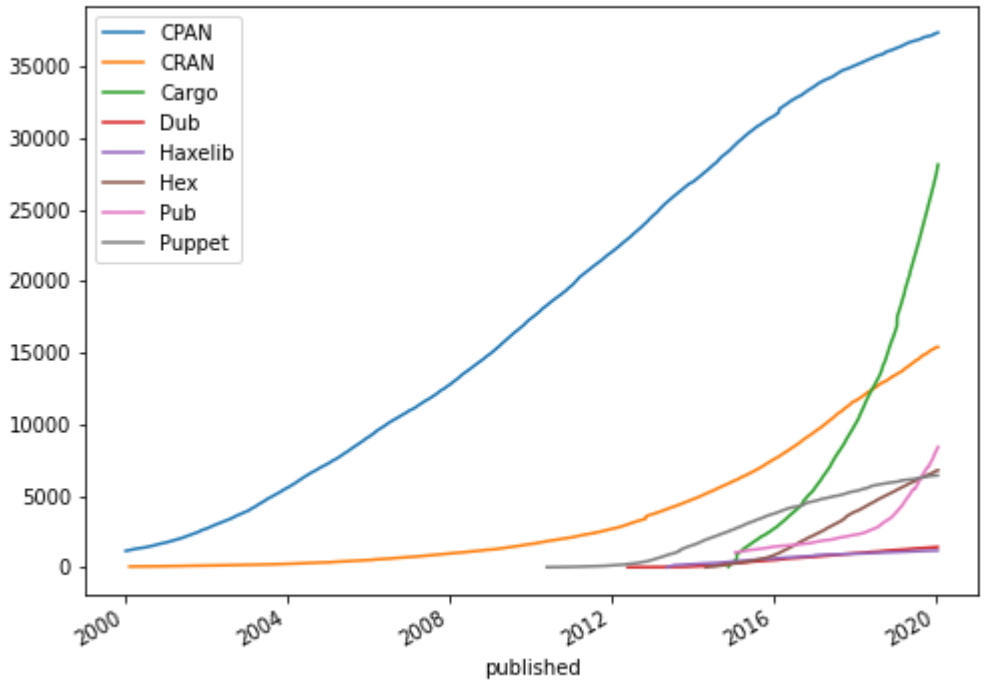

Figure 5: Growth in total libraries count over time for the bottom 8 package managers

CPAN's growth in the beginning is increasing but by the end of the observation period it has started to slow down. Cargo has been growing quickly since its release in 2014 and shows no sign of slowing this trend. CRAN has maintained a steady trend in growth over the observation period. Pub experienced an increase in growth in 2018. Hex experienced an increase in growth at the end of 2015 and retains this trend. In 2012 Puppet experienced a slight increase in growth. Both Dub and Haxelib have few libraries and have not shown major increases in trends.

4.3.2 Versions

Next, the growth in the total count of versions was analysed for each package manager. For that, all entries in the versions table were grouped together by package manager and published date. Then the cumulative sum of versions was found over time for each package manager. The result is shown on Figure 6.

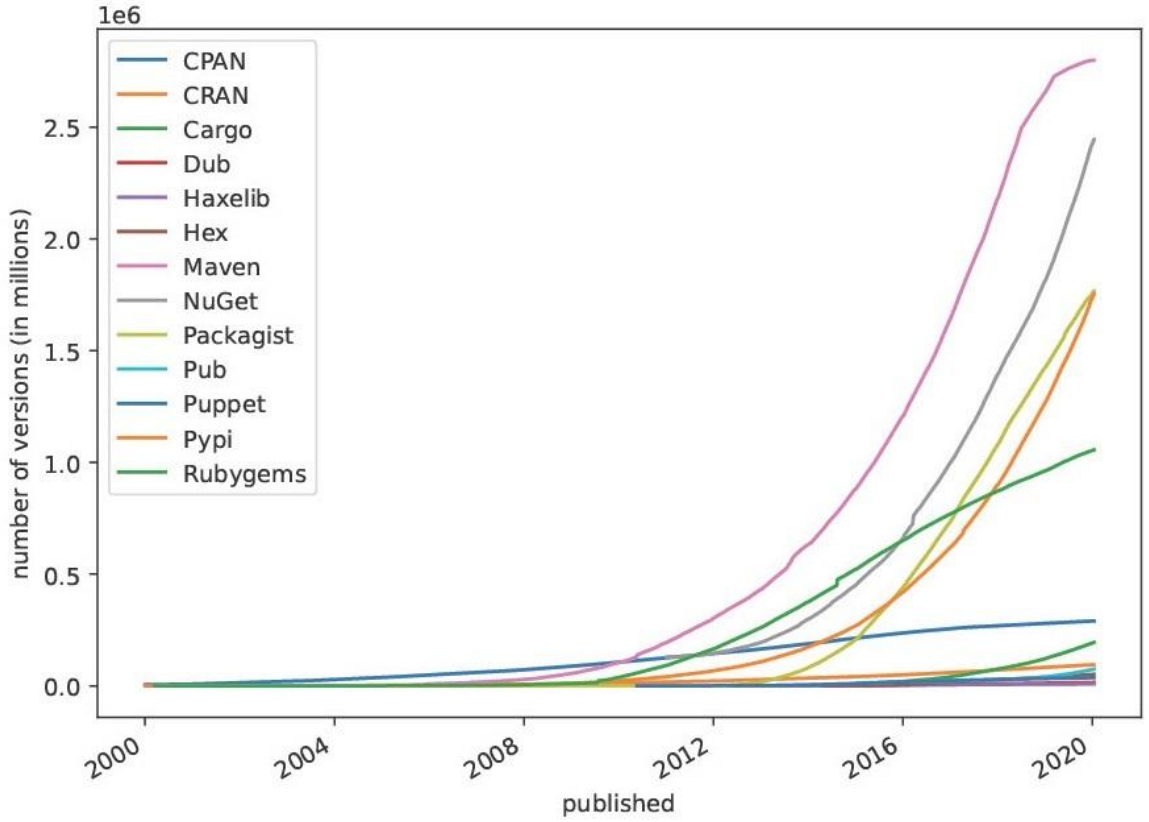

Figure 6: Growth in total number of versions by package manager

Overall, the trends are similar to the growth in the number in libraries. While Maven does not have the most libraries, it does have the most versions recorded on libraries.io. Maven grows exponentially until 2018. From 2018 growth has greatly reduced and has almost levelled off. NuGet's exponential growth does not show signs of slowing down by the end of 2019. Packagist and Pypi grow similarly with a slightly different rate. Neither are showing strong signs of slowing down. Rubygems shows a more of a linear trend that has picked up around 2010 but has since started to slow down.

To get a clearer look at package managers that have fewer versions recorded, Figure 7 was made separately for them.

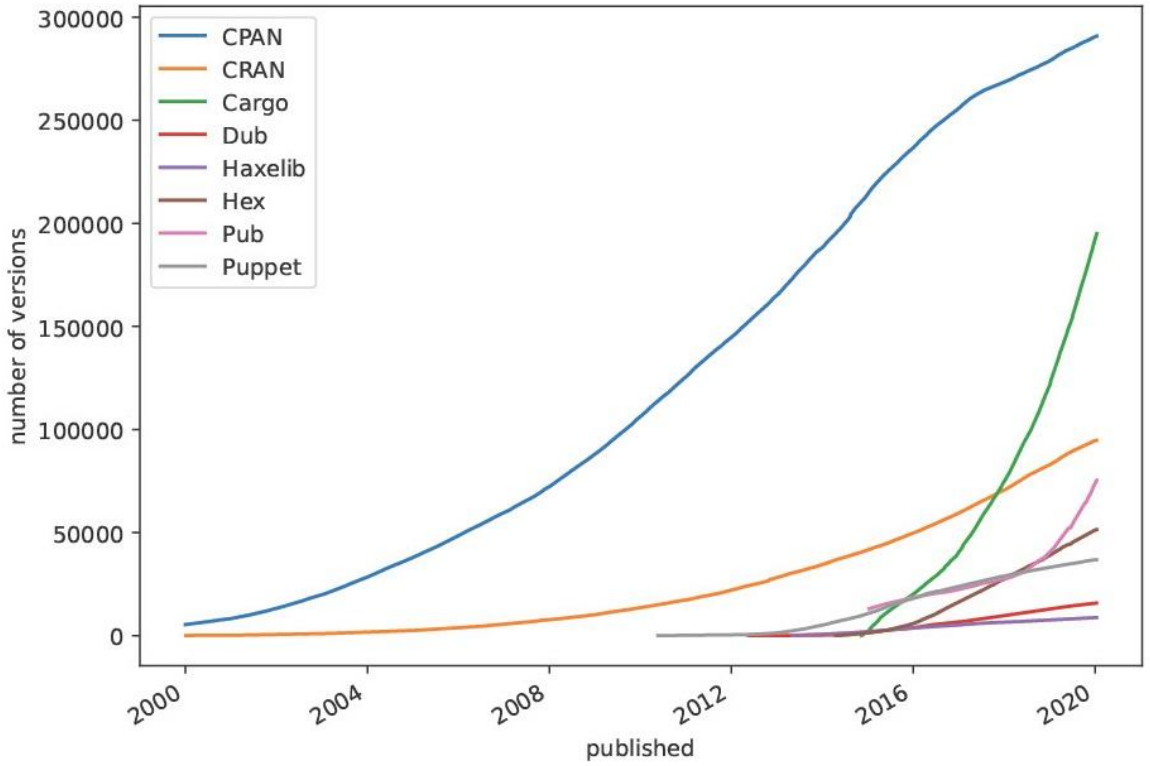

Figure 7: Growth in total number of versions by package manager for the bottom 8 package managers

CPAN shows a growing trend at the beginning but by the end of 2019 this trend has started to slow down. Cargo has a very fast growing trend which shows no signs of slowing down. CRAN has been steadily growing over the observation period. Pub's growth picks up at the beginning of 2019 and remains steady. Hex's growth picked up in 2015 and like most others, keeps a steady trend. In 2013 the growth for Puppet increased and remains steady for the rest of the observation period. Dub and Haxelib have few versions but also have steady trends in growth.

Next, how many versions were being added every month was analysed for each package manager. For that, the combined table of libraries and versions was grouped together by package manager and published date. After that, all entries in the same month were grouped together by the package manager and a sum was found for all months. The result of this process is depicted on Figure 8. The figure starts from 2005 as opposed to 2000 like the previous figures because there was not much of interest happening during that time. As such the first five years of the observation period were cut off from the figure to get a clearer look at the relevant parts of time.

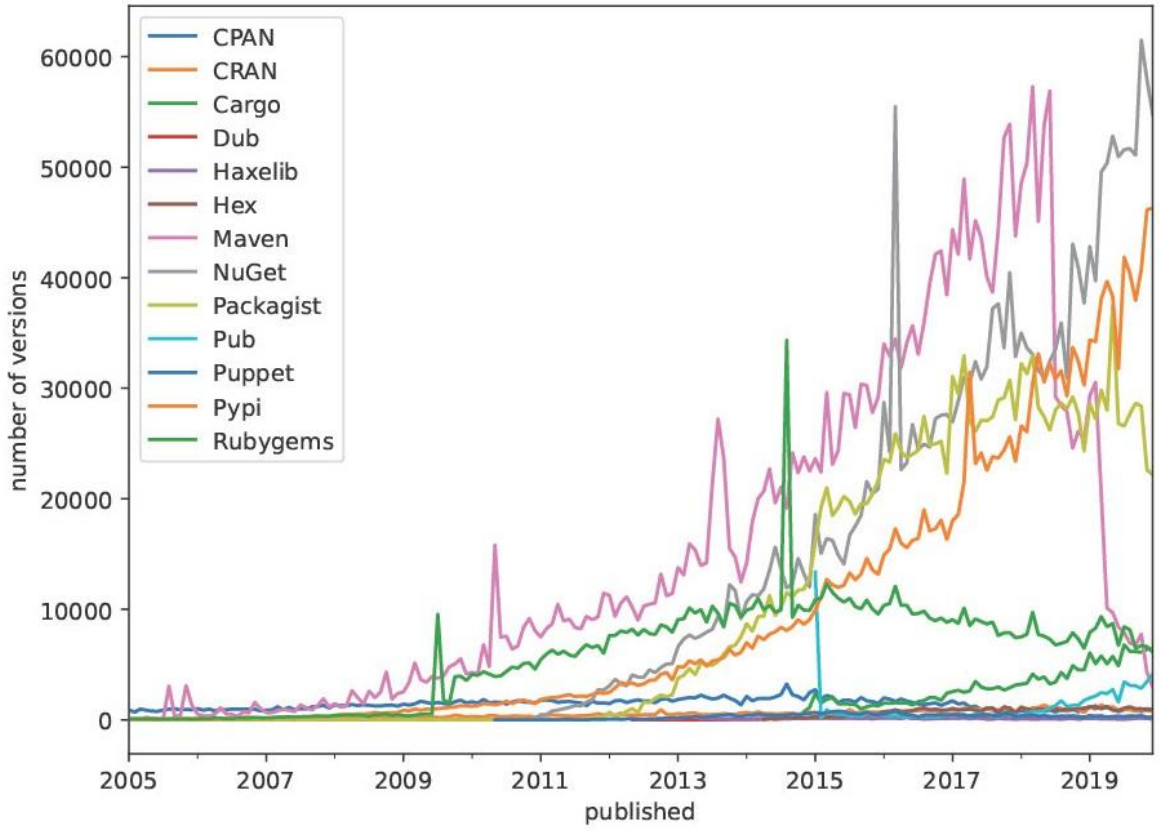

Figure 8: Added versions per month for each package manager

Maven's trend grows steadily until 2018 with two smaller spikes in 2010 and 2013. From 2018 however, the amount of added versions have significantly reduced and drops to almost zero by the end of 2019. NuGet also has a mostly steady growth trend. NuGet has a spike in uploads at the beginning of 2016 and a slight dip around 2018. Pypi sees a surge in uploads in 2017 but otherwise keeps growing steadily. Packagist grows steadily until 2017 only seeing a slight increase in growth at around 2015. From 2017 the uploads of new versions keeps steady until the end of 2019. Rubygems had a growing trend until 2015 with two spikes in 2009 and 2014. After 2015 there has been a decreasing trend in new versions.

To get a clearer look at package managers that have fewer versions recorded, Figure 9 was made separately for them.

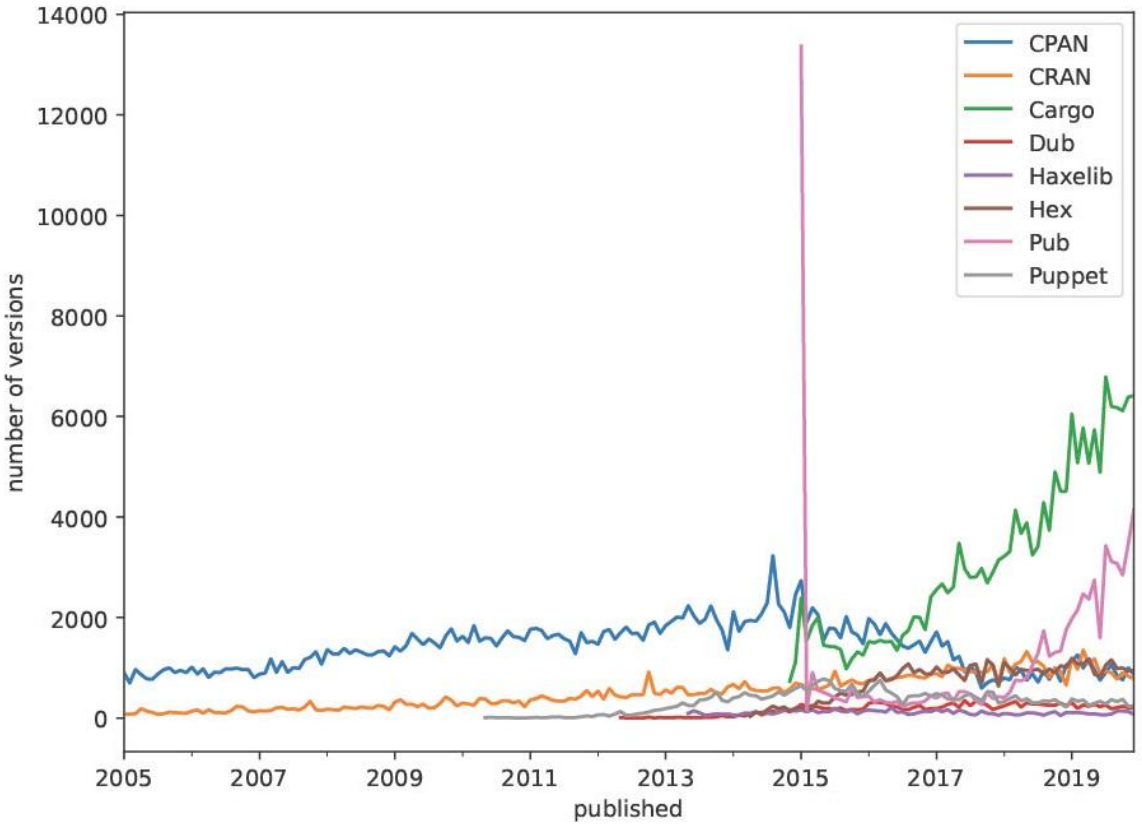

Figure 9: Added versions per month for the bottom 8 package managers

Cargo has a spike at the time of its publishing. Afterwards it has a growing trend until the end of 2019. CPAN's uploads have stayed steady from 2005 to 2015 with only a slight increase taking place over that time period. From 2015 there has been a steady declining trend. Pub has one significant spike at the beginning of 2015. Other than that, the number of new versions per month is steady until 2018. From 2018 there is an increase in growth. CRAN, Dub, Haxelib, Hex and Puppet have a low count of versions and a steady upload of new versions without any significant changes.

4.3.3 Dependencies

Finally, the growth in number of dependencies is analysed. This is done by generating a monthly snapshot for each package manager. The snapshot is made for the first day of each month between the first of January 2000 and the first of January 2020. For each of these dates the newest versions of each library that existed up to this date is found. This is done by grouping the entries in the versions table together by their library id, filtertering out versions that were released after the given month, and taking the latest one from each group. Then the dependencies are added to the filtered versions table by the version id. This creates a snapshot for each month that shows how many unique dependencies each package manager had at that point in time. The results of this process are shown on Figure 10.

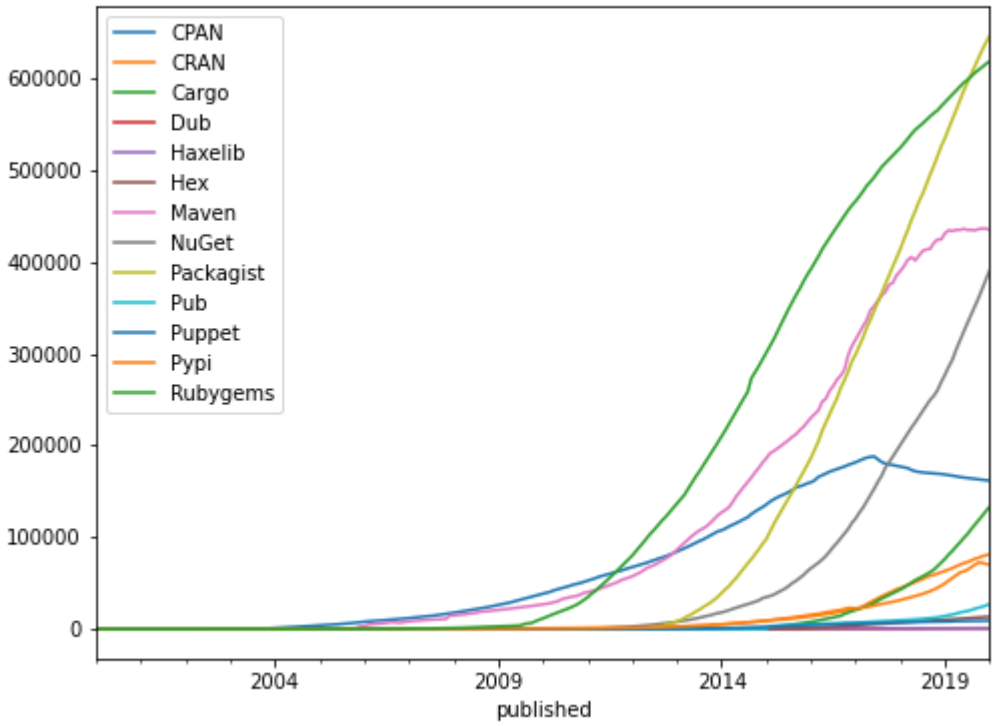

Figure 10: Number of dependencies in monthly snapshots for each package managers

Packagist has the most unique dependencies at the end of 2019. The count of dependencies for Packagist grows exponentially and shows no signs of slowing down. RubyGems comes in second. However, in the case of RubyGems, the trend is slowing down as also observed in the case of libraries and versions on figures 4, 6 and 8. Maven grows exponentially in the beginning but growth levels off by the end of 2019. NuGet grows exponentially and does not show signs of slowing down. CPAN grows steadily until 2017. In 2017 the number of dependencies in the monthly snapshots has started to decrease.

To get a clearer look at package managers that have fewer dependencies recorded, Figure 11 was made separately for them.

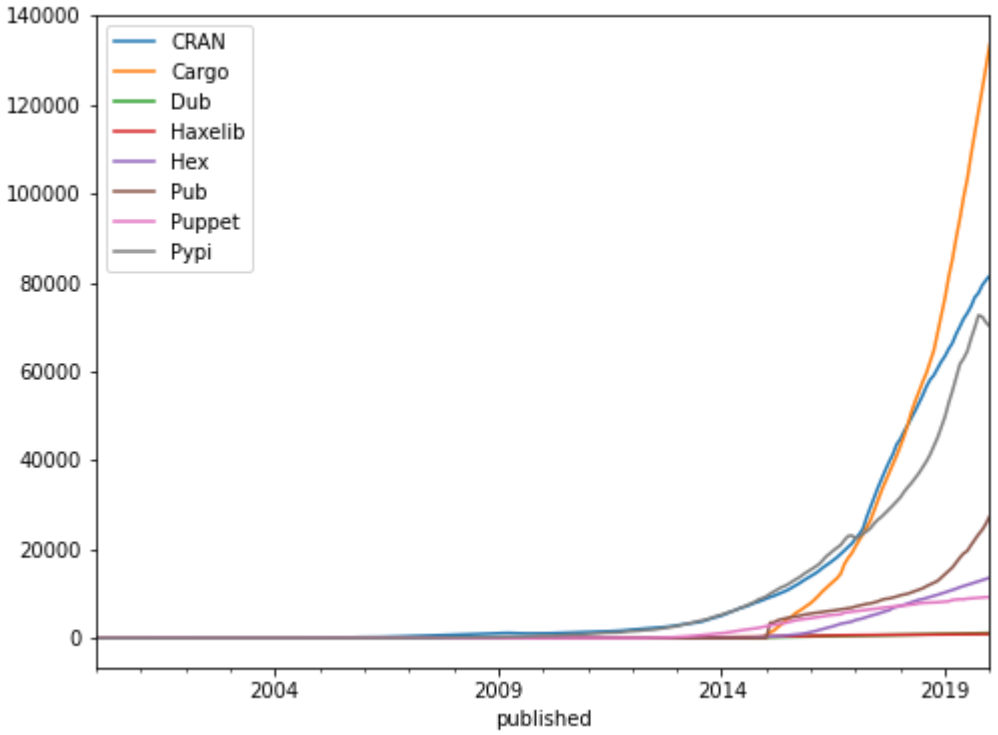

Figure 11: Number of dependencies in monthly snapshots for the bottom 8 package managers

While Pypi had high counts in libraries and versions it has relatively few dependencies recorded in the libraries.io database. The dependencies in the monthly snapshots for Pypi have an exponential growth trend. Yet, a decrease in dependencies is observed at two points for Pypi. Cargo's dependencies in the monthly snapshots are growing exponentially. CRAN is growing quickly as well with the speed picking up from 2017. Pub has a small jump in dependencies around 2015 and has been growing steadily since then. Hex had an increase in growth at the end of 2015. Puppet has grown linearly since 2013 without deviation. Haxelib and Dub have few dependencies and have not experienced much growth.

4.4 Answers to the research questions

RQ1: For which package managers can RQs 2 and 3 be answered using the libraries.io database?

The libraries.io database has data on 38 different package managers. Since the libraries.io homepage lists only 31 package managers, then any package manager that was not mentioned was removed from further comparison. Then, if the package manager had less than 1000 entries in either the libraries, versions or dependecies tables, they were removed as well. Package managers were also removed if their newest version was not from 2020. Additionally, NPM was removed due to the size of the data being too large. This left 13

package managers: CPAN, CRAN, Cargo, Dub, Haxelib, Hex, Maven, NuGet, Packagist, Pub, Puppet, Pypi and RubyGems.

RQ2: How do library dependency networks grow over time?

First, the growth of libraries was analysed within the libraries.io dataset. Most package managers had a stable growth trend. Yet, there were also package managers that had data read in chunks over a short period of time which was then followed by longer periods of time where few libraries were added. Such libraries were Maven and NuGet. Pypi also had minor jumps in the count of libraries. There were also a few package managers that had many libraries read in at the same time shortly after when libraries.io started tracking them. Such package managers were RubyGems, Maven, NuGet and CPAN. These trends can be observed on Figure 3

When it comes to the growth of libraries in package managers, then most have an exponential or linear growing trend. Yet, there were also a few that showed signs of slowing down in growth. Such package managers were Maven, RubyGems and CPAN. These trends can be observed on Figures 4 and 5.

The growth of versions showed similar trends to the growth of libraries with a difference in scale. In this case as well, most package managers showed a growing trend except for Maven, RubyGems and CPAN with Maven's growth coming near zero. These trends were observed on Figures 6 and 7.

Finally, monthly uploads of versions were analysed. Most package managers had an increasing number of versions uploaded monthly over time. Some outliers were found here as well. While Maven had a growing trend at the start, from 2018 there has been a steep decrease in new versions. Packagist also started with a growing trend but this trend has stopped since 2017. Starting from 2017 the number of new versions per month has stayed around the same magnitude. RubyGems grew steadily until 2015 after which there has been a steady decreasing trend in growth. CPAN also showed signs of slowing down in growth since 2015. While monthly uploads of versions may have spikes in some months, they do not affect the overall trend.

RQ3: How connected are libraries in different package managers through dependencies?

To answer this question, monthly snapshots were made to find the total number of dependencies in the newest versions of libraries over time. While most package managers showed a growing trend, there were also those that did not. As in the previous cases, Maven and RubyGems showed a trend of slowing down in growth. Meanwhile, for CPAN the number of dependencies started to decrease in 2017.

5. Discussion

This section examines changes in trends and gives reasons for them but also compares the results with real world data and previous research.

5.1 Libraries

The following subsections validate and discuss results on the number of libraries and compare them to findings from previous studies.

5.1.1 Validating results

For each package manager official statistics were searched. In most cases this included the use of the Wayback Machine [39] to get data from around 2020. This data was then compared to the results based on the libraries.io dataset.

Overall, the official statistics reported less libraries, but the magnitude of the number of libraries was comparable. The only exception to this was Maven where the official statistics reported significantly higher numbers. The following section discusses these observations for each package manager in detail.

Packagist's growth trend and total count of libraries in Figure 4 seem accurate when comparing it to Packagist's own statistics [40].

As for Pypi, it was recorded to have about 200 thousand libraries in the middle of December 2019, therefore the total counts match for Pypi as well[41].

According to libraries.io, NuGet has 199447 libraries. According to NuGet's homepage the number was approximately 182 thousand libraries so here the scale of total libraries is also accurate [42].

RubyGems has a growing trend in libraries until 2014. After that, growth has started to slow down. As this trend has been observed by others as well, then this data is most likely accurate [43] [44]. Both libraries.io and RubyGems homepage report about 160 thousand libraries at the end of 2019 [45].

Cargo has 35635 listed libraries on the libraries.io dataset. On Cargo's homepage around 33 thousand libraries are noted [46].

For CRAN the total number of libraries is 16695 in the libraries.io dataset and on their own homepage it is listed as 15818 in the middle of June 2020 [47].

Pub shows a growing trend on Figure 4 and others have also found that at least from 2019 there indeed is a steady growing trend [48].

Hex's final library count is 9452. According to Hex's homepage, at the end of 2019 there indeed was around 10 thousand libraries so this is also accurate [49].

Dub has 1903 libraries in total in the libraries.io dataset and on their own homepage they have 1708 libraries counted [50].

In the case of Haxelib and Puppet, there were no other data sources found to compare to.

CPAN shows signs of slowing down in growth. As others have come to the same conclusion, then this trend is most likely accurate [44] [51] [52]. The final number of libraries is around 37 thousand while on CPAN's homepage the number is around 41 thousand [53].

However, Maven's graph does not look accurate. According to libraries.io, Maven has 185 thousand total libraries and the growth of the number of libraries has stopped after 2019. Maven should have more total libraries and the growth trend should not be slowing down but continuing exponentially according to mavenreporistory.com [54]. According to the statistics on modulecounts.com, Maven should have about 300 thousand libraries in 2019 [44]. A possible reason for this discrepancy can be observed in Figure 3. In Figure 3 it can be seen that libraries.io reads in data for Maven mostly in chunks. The last time large amounts of data were read in, was in 2018. After this not much new data has been added. Therefore, since the growth trend for Maven in Figure 4 is accurate until 2018, it is possible that libraries.io does not have all data about newer libraries.

5.1.2 Comparison to previous results

The results are compared to the ones that Decan et al. got in their research [10]. The comparable package managers are Packagist, RubyGems, NuGet, Cargo, CPAN and CRAN.

In the Case of RubyGems, the magnitude of libraries matches overtime. Decan et al. [10] found that the growth trend was exponential. However, according to the data in the libraries.io dataset, RubyGems is slowing down in growth. In the beginning RubyGems did show an exponential growth in libraries, it has since been slowing down from around the year 2015.

The magnitude for NuGet matches as well. However, Decan et al. concluded that NuGet had linear growth in libraries [10]. Based on Figure 4, the newer dataset seems to suggest more of an exponential growth trend.

According to Decan et al. Cargo showed a linear trend in growth [10]. According to the libraries.io dataset, that was indeed the case between the publishing of Cargo and the end of 2016, but since then Cargo has shown more of an exponential trend in growth. The amount of libraries matches.

In both cases Packagist has very few libraries at the beginning of 2012. Yet at the end of 2016 the count of libraries has risen to about 100000. In both cases an exponential trend in growth was observed as well. The data matches for CPAN and CRAN as well.

5.2 Versions

The growth in total versions has similar trends to the growth in libraries. If growth in the number in libraries changes so does the growth in versions accordingly.

However, the number of monthly releases of new versions have some noticeable spikes that are worth investigating.

5.2.1 Validating results

This section discusses and validates results on library versions. Possible reasons for sudden spikes in the number of versions released per month are discussed and unusual general trends are validated based on other data sources.

Maven has two spikes in releases: in the first half of 2010 and in the middle of 2013. At around the time of the first spike, Maven 3.0 moved into beta which could be a possible explanation [55]. In the middle of 2013 the alpha for 3.1 was released and a month later 3.1 was out of alpha [56] [57]. In terms of the general growth trend, the sudden dropoff starting in 2018 is not accurate as the trend does not match the data on Maven's own statistics on mavencentral.com [54]. Mavencentral lists information on different repositories that can be used to index libraries. Taking the top 10 most popular library repositories for Maven and analysing how many new library versions are released each year reveals an exponential growth trend instead [54]. It is possible that libraries.io has simply not read in the data for newer versions as is the case for libraries for Maven.

NuGet has a high spike in version uploads at the beginning of 2016. This could be influenced by the release of a new NuGet version at the end of 2015 [58]. There was also a tutorial published at the beginning of the next year on how to use a new feature of the update [59]. On the other hand they also updated their folder structure for NuGet.Server at the beginning of 2016 [60]. As NuGet.Server is used to host and share libraries internally, it could cause users to update their libraries accordingly [60]. However, it is difficult to say whether the data on these spikes is accurate as there is no official data to compare to.

Pypi has a spike in version releases in the first half of 2017. One possible reason could be that the homepage URL for Pypi was changed in the beginning of July that year [61]. With this the software that powers Pypi changed [62]. However, it is unclear whether there is any connection between this update and the spike in version uploads.

RubyGems has two spikes: one in 2019 and another in 2014. In July of 2009 there was only a minor version update to RubyGems so it is difficult to find the reason for this spike [63]. At the time of the 2014 spike there was internal maintenance done on the homepage of RubyGems and also a new version 2.4 was published [64] [65]. The subsequent decline in new versions shortly afterwards is observed in libraries and versions as well on figures 4 and 6.

Pub has a very high number of versions uploaded at the beginning of 2015. Interestingly, the Pub GitHub repository shows that there were no commits between April 19. 2013 and May 15. 2015 [34]. It is possible that there might be a connection between the spike in new versions and the sudden start in commits to the Pub repository. However, since the first recorded versions for Pub are from 2015, then it is likely that at that time a sizable chunk of data was read in one go, causing the spike. Though Pub was published in 2011, there seems to be no data on versions for Pub before 2015.

Cargo has a spike right after its release. This is most likely libraries migrated from other languages or other package managers as Cargo is the newest package manager on the list.

5.2.2 Comparison to previous results

The results for Packagist, RubyGems, NuGet, Cargo, CPAN and CRAN are compared. The magnitude of versions added per month matches for every package manager with the results

that Decan et al. got [10]. While Decan et al. [10] concluded that all package managers remain steady or grow over time, that has not been the case for RubyGems or CPAN. While RubyGems showed a growing trend until 2015, it has been steadily slowing down ever since. CPAN on the other hand, remained mostly steady with a slight increase over time until 2017. From there, CPAN has shown a downward trend in the number of new versions per month.

5.3 Dependencies

The following subsections validate and discuss results on the count of dependencies and compare them to findings from previous studies.

5.3.1 Validating results

RubyGems shows signs of slowing down in growth for the amount of dependencies in monthly snapshots. Considering that in libraries and versions the same trend is seen then this is expected.

Maven's growth has levelled off by the end of 2019. This trend is seen in libraries and versions as well and is caused most likely by libraries.io not reading in data about Maven as seen on figure 3.

CPAN's growth has started to drop from 2017. Upon further inspection the newest dependencies that could be connected to a version are from 31.05.2017. Later versions could not be connected with any dependencies. Querying directly from PostgreSQL gives the same result. Considering that two different methods give the same result, it means that the libraries.io database most likely does not have data on dependencies that connect to any versions after May of 2017.

Pypi, compared to others, ranked high in the number of libraries and versions. Yet, when it comes to dependencies, they have a relatively small amount of them. One option is that libraries.io is missing some information about dependencies for Pypi as Pypi does not tend to know the dependencies for their libraries due to issues with or even lack of dependency management [66] [67]. Dependencies can be declared in documentation or install scripts. Wiener et al. [67] analysed 550000 python repositories and found that 410000 repositories imported third party libraries but did not include any dependency management files. This can lead to difficulties in figuring out the correct dependencies for each library. Which again can lead to missing data in libraries.io. This could also be the cause for the two decreases in dependencies for Pypi.

Pub has a spike at the beginning of 2015 and has been growing steadily since. The spike is most likely caused by the fact that the first recorded versions are from that time as seen on Figure 7.

The rest of the package managers have a steady growing trend without any notable changes.

5.3.2 Comparison to previous results

The overlapping package managers are compared again with the results by Decan et al. [10]. The magnitude and trend of growth match with the result that Decan et al. [10] got. However, they concluded that all package managers show a growing trend. After 2017 RubyGems has

been steadily slowing down in growth. In CPAN's case the trend is difficult to determine due to incorrect data.

5.4 Threats to validity

One of the threats to validity is the contents of the libraries.io dataset. As it is a reputable dataset which has been in other research as well, an assumption was made that it contains credible information. Yet, there were cases which implied that the libraries.io database did not have all the available information. This was the case for example with the Maven package manager. According to the data in the dataset, Maven showed a trend of slowing down in growth. Comparing this data to their own did not match however. To make sure that the database indeed does show this trend, the libraries count was queried in two different ways. Once with the Jupyter Notebook code and then a direct query was made to the data in PostgreSQL.

Reading in the data from the libraries.io dataset was difficult due to its size. Versions and dependencies were read in with a custom script. The analysis was written as a Jupyter notebook. The author made sure to check for any mistakes in the code, but issues stemming from the database size could still affect the results. To check for such mistakes all notable trends were checked against additional data sources.

The code is available at: https://github.com/Karinase/Analysis-of-dependency-graphs. It contains the script for reading data into PostgreSQL and a Jupyter Notebook with the code that was used to create the tables in this research. Therefore, it is possible to check the correctness of the analysis.

5.5 Usage of the libraries.io database

As the 2020 dataset is large, then a lot of space is needed to store it. Figuring out the structure of the dataset is simple with the help of the detailed documentation which gives an overview of every table and the data within them. However, as the tables are large, then there might be difficulty in reading the data into a program for analysis, especially if the programs have set upper limits for size. In this case, a viable option is to read the data in chunks. Writing a separate script for reading the data in takes extra work however. Due to the size of data and the time it takes to read it, the information was first read into PostgreSQL. This was done so that all data could be read in once and then accessed as needed in the Jupyter Notebook file. As it was not possible to read the entirety of the dependencies into the processing file in one go, this was necessary. Since the entries in the dependencies table all had a parameter which showed to which package manager they belonged to, then it was possible to just query for one package manager at a time. Since the entries in the repositories did not have any data on the package manager, reading in the data would have been much more difficult as there was not enough memory to read everything in.

Not all data in the libraries.io dataset can be fully trusted. A sanity check should be conducted for the data. For example, publishing dates in the year 1900 are not valid. Also, results acquired from this dataset should be compared with other available data to check for correctness. In this way it was possible to determine that Maven's growth is not actually slowing down but that the dataset itself does not have sufficient data.

Reading the data from the CSV files into PostgreSQL took a long time. Reading the dependencies table took more than 20 minutes. Reading the data into the Jupyter Notebook file also takes time. Reading in the larger tables can take several minutes. Dependencies queried selectively by their package manager take less time. Still the more entries a package manager has the more time it takes to read all the data in. The difference can vary in minutes depending on which package manager's dependencies are queried. Additionally, since the dataframes in the Jupyter Notebook are large, then any changes made to them also take a long time to process. To read in all the data and run the processes to get the tables for dependencies took around an hour.

In conclusion, when working with the 2020 dataset from libraries.io, space management and working around size limitations would be the biggest challenge. On the other hand, the structure and contents of the dataset are well organised, easy to understand and well documented.

6. Conclusions

The 2020 dataset from libraries.io was used to analyse how package managers grow over time. First, the dataset was downloaded and unpacked. Then, the relevant tables were read into PostgreSQL to simplify the process of loading the data into the analysis file. The data was then read into a Jupyter Notebook file progressively as needed with the aid of the Psycopg2 library.

Once the data was in the Jupyter Notebook file, the libraries, versions and dependencies tables were analysed in detail to find out exactly what kind of data was available. From this, it was possible to find out what kind of data each package manager would need to have to conduct further research on. The selection process removed 25 package managers from comparison due to lack of data. The final selection consisted of 13 package managers: CPAN, CRAN, Cargo, Dub, Haxelib, Hex, Maven, NuGet, Packagist, Pub, Puppet, Pypi and RubyGems.

The growth of libraries was analysed first. Most libraries were analysed by libraries.io continuously. There were also a couple of cases where data seemed to be analysed in bigger chunks after which there was a period where very few libraries were added to the database.

Then the libraries and versions tables were connected and the growth in the number of libraries was found. Most package managers showed a stable or increasing trend. However, Maven, RubyGems and CPAN showed signs of slowing down in growth instead. In the case of Maven it was due to incorrect data in the libraries.io dataset. In the cases of RubyGems and CPAN it was a trend that had been observed by others as well. Decan et al. [10] had previously observed that all package management ecosystems either stayed stable or grew, but this did not seem to be the case anymore.

After that, the growth of versions was analysed for each package manager. The growth in the number of versions showed similar trends to the growth in libraries. Here the same three package managers were the only ones that did not show a growing trend. Again, in the case of CPAN and RubyGems, it was deemed accurate and in the case of Maven, it was due to incorrect data.

While analysing the count of new versions per month, some notable spikes in trends were analysed. In terms of general trends, there were no unexpected results. The package managers that showed a growing trend previously, did so in this table as well.

Finally, the growth in dependencies was analysed. Most package managers grew over time. Maven and RubyGems were slowing down in growth. CPAN's amount of dependencies was going down. In the case of dependencies, CPAN had a lack of data as the newest dependency that could be connected to a version was from 2017.

The results provide valuable insight into how different package managers grow over time. It also showed that not all conclusions that were made from previous research done by Decan et al. [10] still applied. These results can be useful to anyone who uses or develops libraries or package managers.

Libraries.io had some incorrect data in its database. Furthermore, due to the size of data it was difficult to manage. However, the structure and data was easy to understand.

7 References

[1] ‘Third-party software component’, *Wikipedia*. Sep. 05, 2021. Accessed: Oct. 11, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Third-party_software_component&oldid=1042627225

[2] ‘NumPy’. https://numpy.org/ (accessed Oct. 11, 2021).

[3] ‘The Ultimate NumPy Tutorial for Data Science Beginners’, *Analytics Vidhya*, Apr. 27, 2020. https://www.analyticsvidhya.com/blog/2020/04/the-ultimate-numpy-tutorial-for-data-science-beginners/ (accessed Oct. 11, 2021).

[4] ‘jsoup: Java HTML parser, built for HTML editing, cleaning, scraping, and XSS safety’. https://jsoup.org/ (accessed Oct. 12, 2021).

[5] ‘Package manager’, *Wikipedia*. Oct. 09, 2021. Accessed: Oct. 12, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Package_manager&oldid=1049088705

[6] M. D. Says, ‘JavaScript package managers compared: Yarn, npm, or pnpm?’, *LogRocket Blog*, Nov. 24, 2020. https://blog.logrocket.com/javascript-package-managers-compared/ (accessed Oct. 12, 2021).

[7] V. Amilkanthawar, ‘Which Python Package Manager Should You Use?’, *The HumAIn Blog*, Jun. 14, 2019. https://medium.com/in-pursuit-of-artificial-intelligence/which-python-package-manager-should-you-use-150d9696f9db (accessed Oct. 12, 2021).

[8] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, ‘An Empirical Analysis of Technical Lag in npm Package Dependencies’, in *New Opportunities for Software Reuse*, Cham, 2018, pp. 95–110. doi: 10.1007/978-3-319-90421-4_6.

[9] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, ‘On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm’, in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2019, pp. 589–593. doi: 10.1109/SANER.2019.8667997.

[10] A. Decan, T. Mens, and P. Grosjean, ‘An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems’, *ArXiv171004936 Cs*, Oct. 2017, Accessed: Oct. 14, 2021. [Online]. Available: http://arxiv.org/abs/1710.04936

[11] A. Decan, T. Mens, and M. Claes, ‘On the topology of package dependency networks: a comparison of three programming language ecosystems’, in *Proccedings of the 10th European Conference on Software Architecture Workshops*, New York, NY, USA, Nov. 2016, pp. 1–4. doi: 10.1145/2993412.3003382.

[12] ‘Libraries - The Open Source Discovery Service’, *Libraries.io*. https://libraries.io (accessed Nov. 29, 2021).

[13] J. Katz, ‘Libraries.io Open Source Repository and Dependency Metadata’. Zenodo, Jan. 12, 2020. doi: 10.5281/zenodo.3626071.

[14] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, ‘Technical Lag of Dependencies in Major Package Managers’, in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2020, pp. 228–237. doi: 10.1109/APSEC51365.2020.00031.

[15] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, ‘Structure and Evolution of Package

Dependency Networks', in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 102–112. doi: 10.1109/MSR.2017.55.
[16] Q. Li, J. Song, D. Tan, H. Wang, and J. Liu, 'PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities', in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2021, pp. 161–173. doi: 10.1109/DSN48987.2021.00031.
[17] M. Zimmermann, C.-A. Staicu, and M. Pradel, 'Small World with High Risks: A Study of Security Threats in the npm Ecosystem', p. 17.
[18] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, 'A look in the mirror: attacks on package managers', in *Proceedings of the 15th ACM conference on Computer and communications security*, New York, NY, USA, Oct. 2008, pp. 565–574. doi: 10.1145/1455770.1455841.
[19] 'postgresql - ERROR: could not stat file "XX.csv": Unknown error', *Stack Overflow*. https://stackoverflow.com/questions/53523051/error-could-not-stat-file-xx-csv-unknown-error (accessed Oct. 18, 2021).
[20] 'CPAN', *Wikipedia*. Jan. 10, 2022. Accessed: May 05, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=CPAN&oldid=1064923703
[21] 'ANNOUNCE: CRAN'. https://stat.ethz.ch/pipermail/r-announce/1997/000001.html (accessed May 05, 2022).
[22] J. Ooms, 'Possible Directions for Improving Dependency Versioning in R', *R J.*, vol. 5, no. 1, pp. 197–206, 2013.
[23] *Cargo*. The Rust Programming Language, 2022. Accessed: May 05, 2022. [Online]. Available: https://github.com/rust-lang/cargo
[24] 'Packages and Crates - The Rust Programming Language'. https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html (accessed May 08, 2022).
[25] *dub package manager*. D Programming Language, 2022. Accessed: May 05, 2022. [Online]. Available: https://github.com/dlang/dub
[26] *Haxelib: library manager for Haxe*. Haxe Foundation, 2022. Accessed: May 05, 2022. [Online]. Available: https://github.com/HaxeFoundation/haxelib
[27] *Hex*. Hex, 2022. Accessed: May 05, 2022. [Online]. Available: https://github.com/hexpm/hex
[28] 'Apache Maven', *Wikipedia*. Apr. 14, 2022. Accessed: May 05, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Apache_Maven&oldid=1082740778
[29] 'What is Maven: Here's What You Need to Know [Updated]', *Simplilearn.com*. https://www.simplilearn.com/tutorials/maven-tutorial/what-is-maven (accessed May 08, 2022).
[30] 'NuGet', *Wikipedia*. Mar. 15, 2022. Accessed: May 05, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=NuGet&oldid=1077289468
[31] JonDouglas, 'What is NuGet and what does it do?' https://docs.microsoft.com/en-us/nuget/what-is-nuget (accessed May 08, 2022).
[32] *Packagist*. Composer, 2022. Accessed: May 05, 2022. [Online]. Available: https://github.com/composer/packagist
[33] 'Packagist'. https://packagist.org/ (accessed May 08, 2022).
[34] *Contributing to pub*. Dart, 2022. Accessed: May 05, 2022. [Online]. Available: https://github.com/dart-lang/pub
[35] 'How to use packages'. https://dart.dev/guides/packages (accessed May 08, 2022).

[36] *Puppet*. Puppet, 2022. Accessed: May 05, 2022. [Online]. Available: https://github.com/puppetlabs/puppet
[37] 'Python Package Index', *Wikipedia*. May 01, 2022. Accessed: May 05, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Python_Package_Index&oldid=1085631345
[38] *RubyGems*. RubyGems, 2022. Accessed: May 05, 2022. [Online]. Available: https://github.com/rubygems/rubygems
[39] 'Internet Archive: Wayback Machine'. https://archive.org/web/ (accessed May 17, 2022).
[40] 'Install Statistics - Packagist'. https://packagist.org/statistics (accessed May 13, 2022).
[41] 'PyPI Download Stats', Dec. 16, 2019. https://web.archive.org/web/20191216073755/https://pypistats.org/ (accessed May 13, 2022).
[42] 'NuGet Gallery | Home', Dec. 31, 2019. https://web.archive.org/web/20191231070938/https://www.nuget.org/ (accessed May 13, 2022).
[43] 'Here to Stay Analyzing RubyGems Stats for 2018', *Infinum*, Jan. 22, 2019. https://infinum.com/blog/analyzing-rubygems-stats-v2018/ (accessed May 13, 2022).
[44] 'Modulecounts'. http://www.modulecounts.com/ (accessed May 13, 2022).
[45] 'Stats | RubyGems.org | your community gem host', Jan. 07, 2020. https://web.archive.org/web/20200107223958/https://rubygems.org/stats (accessed May 13, 2022).
[46] 'crates.io: Rust Package Registry', Jan. 01, 2020. https://web.archive.org/web/20200101103617/https://crates.io/ (accessed May 13, 2022).
[47] 'CRAN - Contributed Packages', Jun. 21, 2020. https://web.archive.org/web/20200621022812/https://cran.r-project.org/web/packages/index.html (accessed May 15, 2022).
[48] C. Sells, 'Flutter Package Ecosystem Update', *Flutter*, May 21, 2020. https://medium.com/flutter/flutter-package-ecosystem-update-d50645f2d7bc (accessed May 13, 2022).
[49] 'Hex', Dec. 24, 2019. https://web.archive.org/web/20191224132419/https://hex.pm/ (accessed May 13, 2022).
[50] 'Find, Use and Share DUB Packages - DUB - The D package registry', Jan. 19, 2020. https://web.archive.org/web/20200119143341/http://code.dlang.org:80/ (accessed May 13, 2022).
[51] PERLANCAR, 'Dwindling CPAN releases', *perlancar's blog*, Aug. 01, 2019. https://perlancar.wordpress.com/2019/08/01/dwindling-cpan-releases/ (accessed May 13, 2022).
[52] 'CPAN Report 2021'. https://neilb.org/2022/02/07/cpan-report-2022.html (accessed May 13, 2022).
[53] 'The Comprehensive Perl Archive Network - www.cpan.org', Dec. 31, 2019. https://web.archive.org/web/20191231123014/https://www.cpan.org/ (accessed May 13, 2022).
[54] 'Maven Repository: Repositories'. https://mvnrepository.com/repos (accessed May 13, 2022).
[55] '[ANN] Apache Maven 3.0-beta-1 Released-Apache Mail Archives'. https://lists.apache.org/thread/46y4mlq13vpjsx0fwr5f5xhb2xhh7b5k (accessed May 14,

2022).
[56] ‘[ANN] Maven 3.1.0-alpha-1 Release-Apache Mail Archives’. https://lists.apache.org/thread/zhmprt43yco8n6oh5qh6dop2txtxvzgl (accessed May 14, 2022).
[57] ‘[ANN] Maven 3.1.0 Release-Apache Mail Archives’. https://lists.apache.org/thread/xwzdplqpqs9l7dnxkfl2h8o72zy7kqzf (accessed May 14, 2022).
[58] ‘NuGet 3.3 Release’, *The NuGet Blog*, Nov. 18, 2015. https://devblogs.microsoft.com/nuget/nuget-3-3/ (accessed May 14, 2022).
[59] ‘NuGet ContentFiles Demystified’, *The NuGet Blog*, Jan. 26, 2016. https://devblogs.microsoft.com/nuget/nuget-contentfiles-demystified/ (accessed May 14, 2022).
[60] ‘Accelerate your NuGet.Server’, *The NuGet Blog*, Jan. 13, 2016. https://devblogs.microsoft.com/nuget/accelerate-your-nuget-server/ (accessed May 14, 2022).
[61] ‘Migrating to PyPI.org — Python Packaging User Guide’. https://packaging.python.org/en/latest/guides/migrating-to-pypi-org/ (accessed May 14, 2022).
[62] ‘Integration guide — Warehouse documentation’. https://warehouse.pypa.io/api-reference/integration-guide.html (accessed May 14, 2022).
[63] ‘1.3.5 Released - RubyGems Blog’. https://blog.rubygems.org/2009/07/21/1.3.5-released.html (accessed May 14, 2022).
[64] ‘Maintenance window on July 14th - RubyGems Blog’. https://blog.rubygems.org/2014/06/30/maintenance.html (accessed May 14, 2022).
[65] ‘2.4.0 Released - RubyGems Blog’. https://blog.rubygems.org/2014/07/16/2.4.0-released.html (accessed May 14, 2022).
[66] D. Ingram, ‘Why PyPI Doesn’t Know Your Projects Dependencies’. https://dustingram.com/articles/2018/03/05/why-pypi-doesnt-know-dependencies/ (accessed May 15, 2022).
[67] ‘Why so many Python projects lack dependencies management in git, and what you can do about it’, *The Official Tabnine Blog*, Feb. 15, 2022. https://www.tabnine.com/blog/why-so-many-python-projects-lack-dependencies-management-in-git-and-what-you-can-do-about-it/ (accessed May 15, 2022).

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Karina Sein,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

"Analysis of dependency graphs of third party libraries in different package managers",

(*title of thesis*)

supervised by Kristiina Rahkema.

(*supervisor's name*)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Karina Sein

17/05/2022