National College of Ireland

Automatic Software Dependency Management using Blockchain

Msc Research Project
Cloud Computing

Gavin D'mello
x17110483

School of Computing
National College of Ireland

Supervisor: Horacio González–Vélez

National College of Ireland
Project Submission Sheet – 2017/2018
School of Computing

Student Name:	Gavin D'mello
Student ID:	x17110483
Programme:	Cloud Computing
Year:	2018
Module:	Msc Research Project
Lecturer:	Horacio González–Vélez
Submission Due Date:	13/07/2018
Project Title:	Automatic Software Dependency Management using Blockchain
Word Count:	5649

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are encouraged to use the Harvard Referencing Standard supplied by the Library. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action. Students may be required to undergo a viva (oral examination) if there is suspicion about the validity of their submitted work.

Signature:	
Date:	16th September 2018

PLEASE READ THE FOLLOWING INSTRUCTIONS:

1. Please attach a completed copy of this sheet to each project (including multiple copies).
2. **You must ensure that you retain a HARD COPY of ALL projects**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on the computer. Please do not bind projects or place in covers unless specifically requested.
3. Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Automatic Software Dependency Management using Blockchain

Gavin D'mello
x17110483
Msc Research Project in Cloud Computing

16th September 2018

Abstract

Contemporary software deployments rely on cloud-based package managers for installation, where existing packages are installed on demand from remote code repositories. Usually, frameworks or common utilities, packages increase the code reusability within the ecosystem, whilst keeping the code base small. However, disruptions in the package management services can potentially affect development and deployment workflows. Furthermore, cloud package managers have arguably an ambiguous ownership model and offer limited visibility of packages to the users. This work describes the development of a blockchain-based package control system which is decentralised, reliable and transparent. Blockchain nodes are installed within the infrastructure to provide immutability, and then a dependency graph is constructed to trace the software provenance and package reliance. Our system has been tested with 4338 packages from NPM, 950 out of which are the top depended-upon packages.

Contents

1 Introduction

Long considered a key practice in the industry, software reuse entails the creation of new software systems using existing software packages Krueger (1992). Most software packages are made available as common utility tools or frameworks which are used by millions of users. With the advent of a microservices and cloud architectures, package reuse has increased as each service has its lifetime and own state to manage independently of other services. Each language and community tend to have a different operating package manager, and the proliferation of online version control tools such as Github and Bitbucket have led to the creation wider range of interdependent software components Decan et al. (2016).

Package managers provide a platform for code sharing and reliability. Reliable application package managers are of prime importance to software developers. Most packages need to be installed right before the deployment phase. Software packages tend to have direct and transitive dependencies on other packages, which make them vulnerable and/or prone to failure if any dependency is unpublished or compromised. Dependencies are not necessarily straightforward and can have multiple nesting levels. For example, the package `libcurl`, which is used for sending HTTP requests, depends on other packages like `zlib` which is used to compress data. Any failure in an application package manager could lead to build failures and hinder the development processes.

Different software package managers offer mirrors and streaming. Nonetheless, most package managers are heavily centralised in their architectures, which can present a single point of failure and, more relevant to this work, a source of inconsistencies when package components or versions change. It is therefore important to check if existing package managers can be decentralised to improve the reliability of the ecosystem.

Blockchain smart contracts allow us to store data and execute functions on them in a decentralised setup. Once the smart contract is deployed, transactions can be sent to the contract. The changes made by the transactions can be mined and broadcasted to the entire network. This work focuses on the application of smart contracts to maintain an immutable decentralised change control system for packages and versions. It can assure the provenance of a given set of packages to assure the correct development and deployment for a given set of new packages. We have evaluated our work using 4338 packages from `NPM`.

This paper is organised as follows. Section 2 discusses package managers and existing Blockchain systems. Section 4 outlines our proposed method to manage software packages with Blockchain, including the proposed algorithm for smart contracts. Section 5 shows the implementation of versions on the Smart contracts. Solidity was used to write the Smart contract and peer to peer storage was used to upload the packages. The version tree is described which shows how one version is different from another. The pattern used to store the contract also helps us to seamlessly change the processing logic from the storage. Section 6 presents our evaluation using 4338 packages along with some of their versions. These 4338 packages include packages which are the most depended-upon and key utilities. The section also outlines the bandwidth requirements of the blockchain node and the latency involved in pulling the packages from the system. It also shows a part of a network graph modeled directly from the data coming from the blockchain node. Finally, Section 7 presents some concluding remarks.

2 Related Work

Traditionally, software has been created using a waterfall model where every change goes through some pre-requisite number of stages. With the advent of open-source and rapid collaborative environments, rapid application development has increasingly become the norm for applicative environments Ruparelia (2010). Agile, Scrum, Extreme Programming, and other rapid application development methodologies have become more popular, since they allow changes to be added dynamically, leading to continuous implementation using a backlog. Developers pick software features from the backlog and releases are made at shorter durations compared to the traditional model, leading to adaptive software development Highsmith and Cockburn (2001).

Code bases are constantly evolving over time and version control tools like Git and Subversion are widely used to manage versions and control changes. Github and Bitbucket are widely employed in the software industry for hosting remote code repositories based on such services. These are centralised stores keep all the repositories published and also improve code discovery.

Github and Bitbucket encourage reuse of application code and, arguably, save resources and configuration efforts as the same package can be globally used by many different programs. However, Github and BitBucket package management can be complex. Packages have to be typically downloaded using source code and each client needs to explicitly keep version control on each package.

Cloud computing enables the provision of software on demand but requires applications to be delivered as a consistent lightweight service over the Internet. Consequently, monolithic software architectures are constantly getting divided into microservices Balalaie et al. (2016). The important decision developers have to make is what constitutes a service. The separation of concerns has made life difficult for developers because some code can be duplicated across services.

Managing dynamic versions and software dependencies is complicated, as the program dependency graph for large programs is long known to be difficult to handle Ottenstein and Ottenstein (1984). Microservice-based cloud architecture—where package dependencies can be linked to different packages—then increases the challenge at hand Toffetti et al. (2017), since the search space is significantly large to completely understand conflicts between dependencies.

The term 'DLL hell' has been coined to describe many different versions of the same library Tucker et al. (2007). Programs using the different versions of the same library tend to break in case there are major changes in the package, so package managers are expected to be 'intelligent enough' to handle different versions of the same package. A CUDF (Common Upgrade Description Format) document was proposed to keep a track of the package definition and its dependencies Abate et al. (2012), similar to PyPI's `requirement.txt` file or NPM's `package.json`. However, modern-day application package managers such as NPM can have multiple versions of the package Schlueter (2010), and common version requirements are kept in a common directory and alternate versions are kept local to the package which helps to eliminate collisions of different versions. Some package managers use semantic versioning or 'semver' principles. These principles should be clearly understood by the authors and users. Authors must understand that the breaking changes must always be released as major changes. Users must carefully review the changes in the packages to avoid build errors. Failure to understand these laws puts build systems at risk. Versions are divided into three parts *Semantic Versioning*

Table 1: List of languages and their package managers

Language	Package manager
Java	Maven
Nodejs	Npm
Python	Pypi
C#	Nuget
PHP	Packagist
Ruby	Ruby gems

user guide (2013): Major.Minor.Patch

A security-oriented management framework, CHAINIAC has been used to verify integrity and authenticity for software-release processes based on decentralised nodes Nikitin et al. (2017). However, CHAINIAC does not appear to address the immutability issue as changes in versions may eventually break compilation and software components. Major version bumps are for breaking changes in the API or when the package big parts of the package are being rewritten. Minor versions are for new features additions to existing set without breaking changes. A patch is bug fixes without breaking changes.

Every programming language has its own package manager. Table 1 has a list of selected languages along with their package managers

Having openly released their architecture for dependency management, NPM is the package manager for JavaScript and is also widely considered among the largest code repositories in the world Wittern et al. (2016).

3 Contribution

This is a brief comparison between the existing systems and the proposed method.

Table 2: Comparison of existing systems and the proposed method

Variable name Head	**Different Package Managers**			
	Maven	*Nuget*	*NPM*	**Proposed Method**
Decentralized	No	No	No	Yes
Write Throughput	High	High	High	Low
Read Throughput	High	High	High	High
Immutable	No	No	No	Yes

The proposed method involves using a blockchain-based smart contract to decentralise the package management. The idea is to propose a model which can work well with both centralized and decentralized systems. Decentralisation is offered by a blockchain network, in our case Ethereum. Ethereum uses Solidity for creating smart contracts. The language has similar syntax to Javascript. The contract is compiled by a Solidity compiler. The Solidity compiler returns the bytecode which needs to be deployed to the Ethereum node. language is yet under heavy development. There are several features like the dynamic passing of structures and mappings which are experimental under the current version.

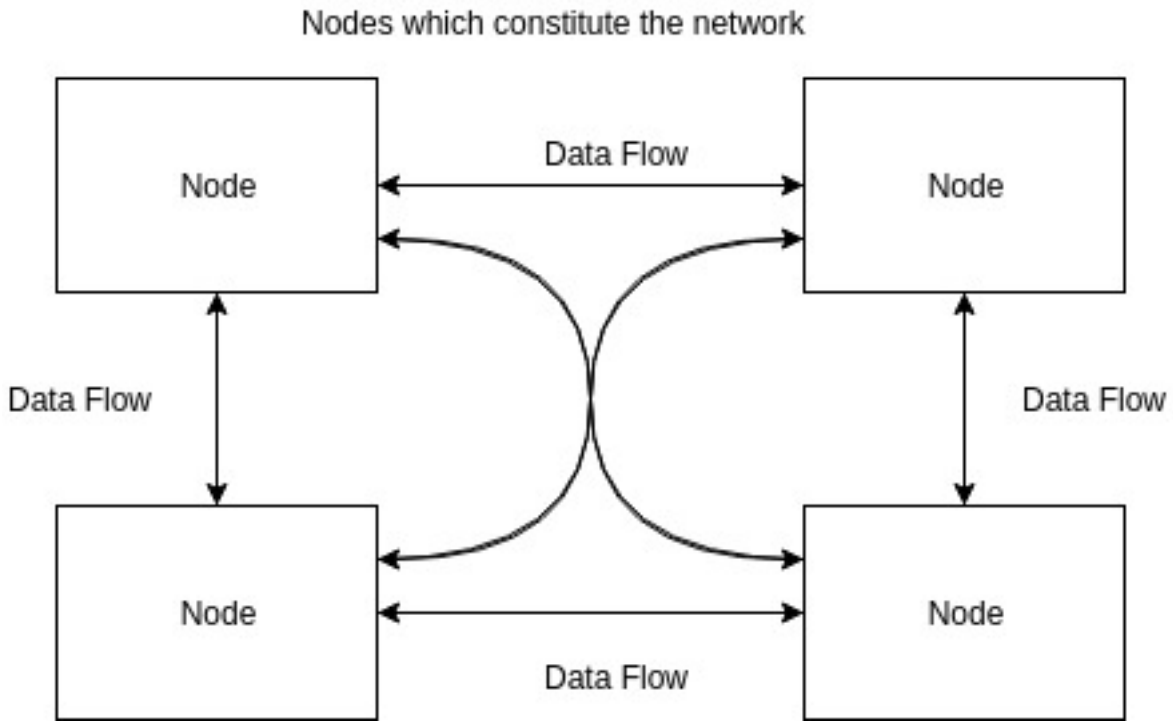

Figure 1: Blockchain data flow.

A blockchain can be construed as a network rather than a technology or system. The smart contract storage is essentially a public database which is immutable and decentralized. It has primarily been used in financial systems. It only came into prominence when Nakamoto released the Bitcoin paper Nakamoto (2008). The system did not have a central authority for verifying transactions and involves having a group of miners to which verify transactions. The miners are paid for the electricity and CPU spent in verifying the transactions. Figure 1 shows how data flows between different nodes in a blockchain network. The work was substantial enough to gain traction. Bitcoin is not Turing complete and potential blockchain adopters could not do anything but get inspired by the system and reinvent the wheel by building something similar.

Ethereum is considered a natural extension of Bitcoin which is also a cryptocurrency. Ethereum is a Turing complete solution where the user can program and deploy bytecode to the Ethereum nodes Buterin et al. (2014). This led to interesting opportunities where users could actually harness the actual strength of the network. The concept of smart contracts allowed the user to add custom logic to the Ethereum nodes. The smart contract is written in Solidity which is a general-purpose programming language. The smart contract functions can be called from thin clients like browsers which we call decentralized applications. Any change to data on the Blockchain is mined by the Ethereum community miners and there is a gas price which the miner gets for blocks that they add to the system. Ethereum uses a distributed ledger for transactions and provides a decentralized architecture for smart contract execution. After the smart contract is deployed onto the Ethereum node, the contract is executed via RPC calls from thin clients.

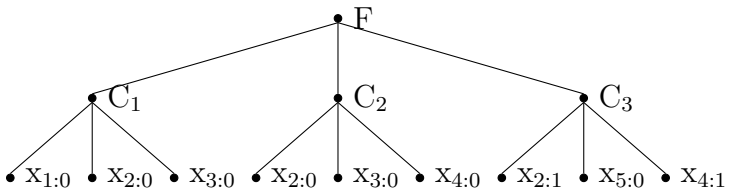

Figure 2: Transitive nature of packages.

4 Methodology

Equation (1) presents the clauses which act as dependencies of a package to be installed.

$$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_5 \vee x_3) \tag{1}$$

Each clause has three versions which have all literals as sub-dependencies of dependencies. Each literal has two versions $x_{1:0}$ and $x_{1:1}$, where $x_{1:0}$ is given as the negation in the package. To install a package F, we need to install all its sub packages given by clauses in the equation. Assuming we can install only one version of the package at a time, we need to find a satisfying assignment to Equation (1) such that we get a successful response where all clauses can be installed for the main package to be installed. It has been shown that the problem is NP-complete by converting the 3SAT instance to a dependency problem instance Cox (2016). The NP-completeness can be avoided by attacking some assumptions like only a single package can be installed on the instance at a time and having a versioning system which does not specify package range.

Every package manager has to deal with direct and transitive dependencies. The more the transitivity, the more the complicated the network. The transitivity can be seen from 2. Here, all the nodes are packages and the edges represent the dependence. We can see that package F depends on C_1, C_2 and C_3. Also, packages C_1, C_2 and C_3 directly depend on other packages.

Here, each C clause has three versions each of which depends on different packages. For example, $C_{1:0}$ depends on $x_{1:0}$ and $C_{1:1}$ depends on $x_{2:0}$. A dependency chart was used for the 3SAT reduction as shown by Cox (2016). If the x packages were to be removed, it would affect F and all C packages. The 'left-pad' problem was pointed out by Decan et al. (2016), where a package named 'left-pad' was used by major packages like Atom and Babel. The owner of 'left-pad' unpublished this package which caused all installations and builds to fail. Two percent of the transitive packages installations failed because of this event. Developers started rethinking their dependency structures and including only which they dire needed.

Smart contracts are similar to stored procedures in SQL. The bytecode of the smart contract is deployed to all the network nodes. The functions of the bytecode are then invoked via RPC calls on the client. New data to be mined requires proof of work from the Ethereum node. Proof of work here means the node will have to perform some computational work before adding a block to the network. This reduces the number of the blocks a node can add to the network and gets fairness to the system.

4.1 Data structure

The data schema can be best seen as a tree. The root node acts the package name and the leaf node is the actual package information. The nodes are version numbers. The

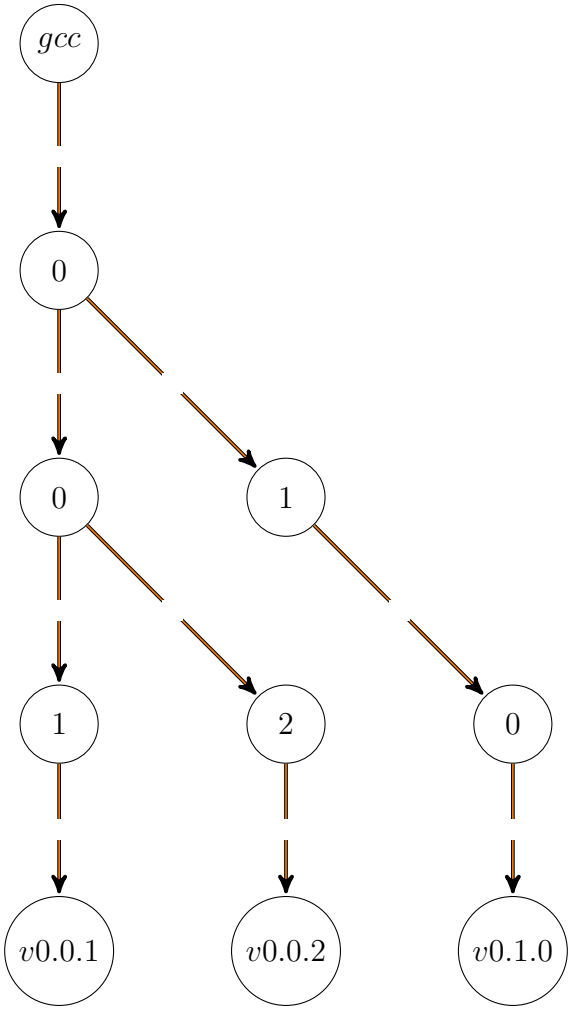

Figure 3: Placement of new versions on version tree

```
{
    owner : 'example_ownerid',
    dependencies : {
        'Package1' : '1.1.2',
        'Package2' : '1.2.1'
    },
    link : 'example_link',
    checksum : 'example_checksum'
}
```

Figure 4: Data structure for package versioning.

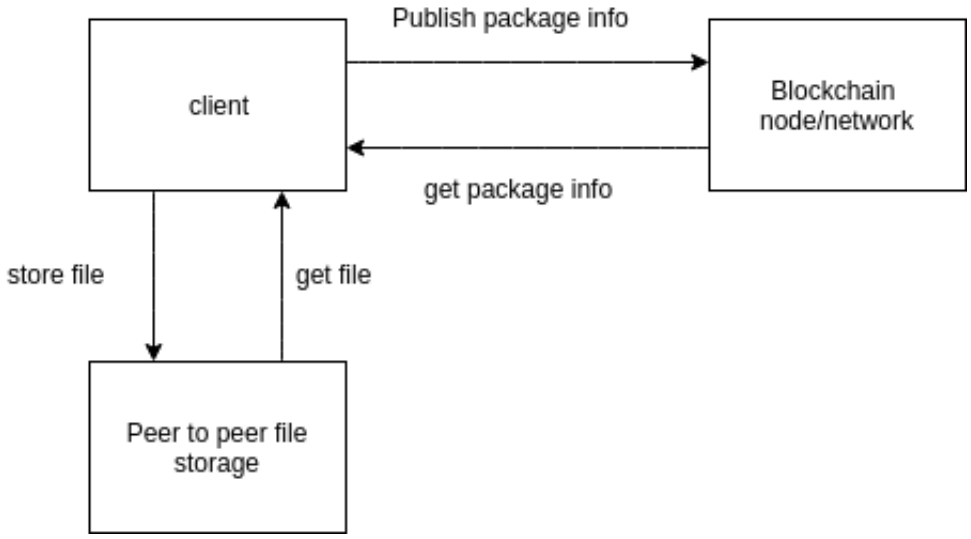

Figure 5: Smart contract interaction.

data structure provided in Figure 4 gives a gist of the data structure. Every package would have its package tree. The data structure is flexible to both the minor major versioning technique as well as semantic versioning where we have three versions. An oversimplification of this data structure would be to have to nested map pointing to a resource. Using as hashmap keeps the time complexity to $O(1)$.

The package information will contain information which is important to install the package like dependencies, link, dependents. The package dependencies are the packages which will be installed with the package. The version of the dependencies is important here to exactly install the dependency the package needs. The data can be fed to the client either in a single go or differently for every package. The ownerId helps us to identify the owner of the package. The package ecosystem can be looked upon as a giant graph where each can depend on other packages or other packages depend on it. The link is the pointer to the package which will be stored with IPFS.

The developers can host the Ethereum nodes in their environment or use the network. Some users would like to keep a copy of the blockchain in the event the network goes down. The usage of the node or the network will be configurable via the package manager client.

4.2 Storage solution

We need to keep the bare minimum information on the ledger so that the intermediation of metadata on the peer to peer network is fast. A compressed version of the package would be kept on the IPFS. IPFS(Interplanetary file system) is peer to peer storage solution. While uploading the package file to IPFS we receive an immutable hash. This is hash can be used to retrieve the file in the future. IPFS uses a Distributed hash table to store the hash and the data is stored locally in the node where it is published. Any other node which requests a file has to download the file from the nearest node and keep it locally with its self for a definite period of time.

We use a Coral Distributed hash table to store the data which concentrates more on locality Benet (2014). A replication factor can be added to have some replication of blocks. The hash returned is stored on the Smart contract. IPFS internally uses the Bit

Torrent protocol where it opens up many connections to the different peers and downloads bits and pieces of the file simultaneously. By using IPFS we make sure that no part of the architecture is centralised. IPFS is used for storing web archival records where the payloads are stored in IPFS and the indexes are stored in a format called CDXJ Alam et al. (2016). CDXJ is an extension to CDX which has JSON support. CDXJ plays a similar role to the blockchain node in our system except that it is mutable.

4.3 Algorithms

The algorithm is for storing package which the developer wants to make available. The client side would have to provide the owner name, package name, version, and dependencies. The package version, name, and dependencies are extracted from the dependency file on the client. The package to be published is stored in the data structure mentioned above which will contain trees for all packages. The complexity of this algorithm is $O(1)$. The package Info is the same as the mentioned above. It will contain the dependencies, dependency versions and link to the current package.

Algorithm 1 Publish algorithm

```
1: procedure publishPackage
   ▷ *Smart contract storage
2:     packages ← package map
   ▷ *inputs
3:     pn ← name of the package
4:     packageInfo ← packageInfo
5:     v ← version
6:     major ← version major
7:     minor ← version minor
8:     patch ← version patch
9:     if packages[pn] then
10:        if packages[pn][major][minor][patch] == null then
11:            packages[pn][major][minor][patch] ← packageInfo
12:            success ← true
13:        else
14:            success ← false
15:        end if
16:    end if
       return success
17: end procedure
```

The algorithm for downloading an installed package is given. The asymptotic complexity of installing the entire package depends on the number of dependencies in the package. Requesting a package with its version from the storage layer would have a time complexity of O(1). Web3, which is the client used to connect to Ethereum node is does not support passing structures downstream yet. As and when support is added for dynamic structures being returned downstream, the model contract can have the processing logic to collect the package with all its dependencies. This would decouple the storage and the processing and the model contract can be changed with time.

Algorithm 2 Install package algorithm

```
1: procedure GETPACKAGE
                                                    ▷ *Smart contract storage
2:     packages ← map of packages
                                                    ▷ *inputs
3:     major ← version major
4:     minor ← version minor
5:     patch ← version patch
6:     result ← []
7:     if packages[pn][major][minor][patch] != null then
8:         packageInfo ← packages[pn][major][minor][patch]
9:         dependencies ← packageInfo['dependencies']
10:        result ← packageInfo, dependencies
11:        success ← true
12:    else
13:        success ← false
14:    end if
       return success, result
15: end procedure
```

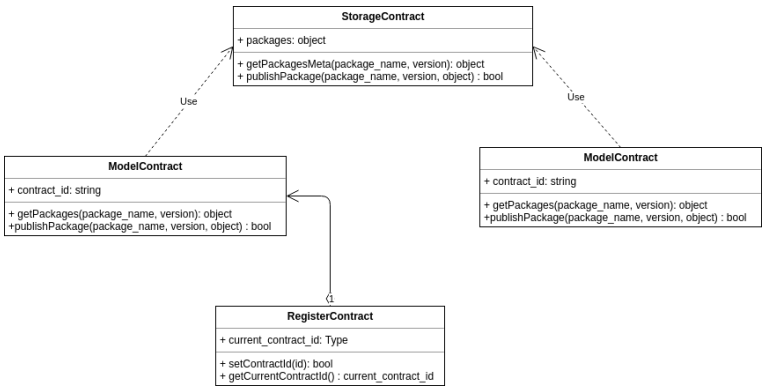

Figure 6: Contract Deployment pattern.

4.4 Diagrams

Class diagram

Contracts are immutable in nature. In a centralized setup, we just update the code with and deploy to all the servers. While using contracts cannot be updated, new contracts need to be deployed. We need to make sure we do not lose references to our storage layer. The storage layer is separated to ensure a reference is maintained. Every change to the contract would require an update to the client side binaries, this would be highly inefficient. To overcome this issue, we propose a register-contract which contains a reference to the main contract. The client function will have to execute the register-contract and get the address of the main contract. Once the address is received, it will be able to get the latest code of the contract. Separation of concerns is of prime importance in the blockchain.

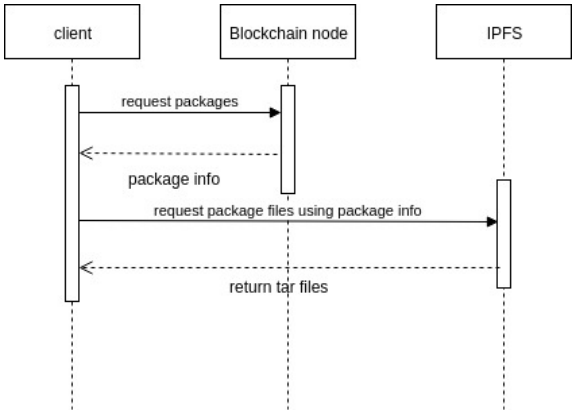

Figure 7: Package installation.

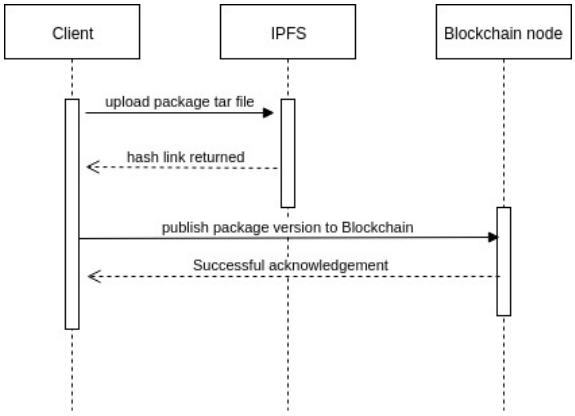

Figure 8: Publishing a package.

Sequence diagrams

The installation sequence diagram shows the installation of a package. The diagram shows the interaction between the client, the network, and the IPFS server. In HDFS, the blocks do not flow through the name-node. However, the name-node does keep a log of all the blocks on the data nodes Shvachko et al. (2010). Similar to HDFS, the file data does not go through the blockchain nodes. The file data is fetched from the IPFS servers. This would increase the throughput and also avoid the blockchain intermediation issue.

The package to be published is uploaded is converted to a tar file. Once the package is compressed, it is uploaded to IPFS. IPFS returns a hash link which helps us uniquely identify the file when needed. We take this hash link and send it to the blockchain node which stores the package information.

The client state diagram shows the flow of events on the client side library. The client-side library will not installed packages that are already installed unless the developer tries to install a different version of the same package. By doing this we save a considerable amount of bandwidth. The client maintains a local state which keeps a track of the packages that are installed. Top level packages will directly replace the other version by design. The packages and their dependencies will be installed in a shared space to avoid duplication. Dependencies of the different versions will be kept local to the package that requires them and not in a shared space.

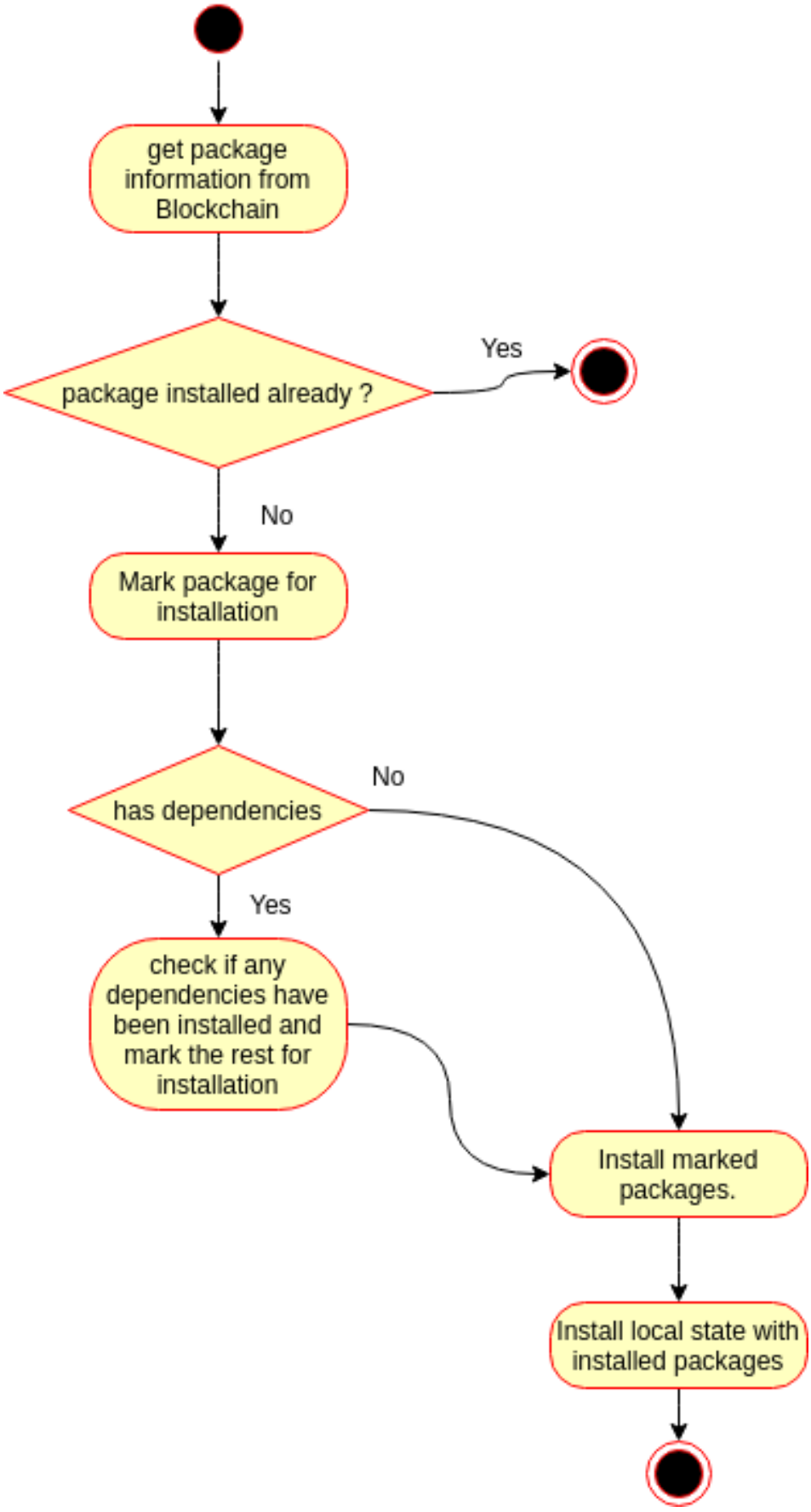

Figure 9: State diagram for the client.

5 Implementation

The proposed solution uses Ethereum smart contracts to store package metadata and binaries in IPFS. Table 3 shows the software requirements for the implementation. The Geth binary was used to run the Ethereum node. The web3 package was used for the client side implementation. Solidity was used to write the smart contract. The web3 client gives a nice interface to interact with Ethereum based smart contract. A CentOS instance was created on Open stack to run the Geth binary and also the P2P storage server. The Geth binary is built in Go which means it is compatible with operating system.

The setup can be used as a standalone system or in collaboration with another package manager as shown in figure 10. The standalone system would require users identifying themselves to the network and would require sufficient funds to submit transactions. The collaboration with other package managers would allow anyone to submit transactions as there would only group of services publishing for everyone. The copies of the transactions would be maintained in both the centralized and the decentralized systems.

The NPM listener pushes metadata to interested services. The metadata consists of the name, version, license etc. A daemon was created to get the metadata from NPM. This daemon also excludes packages which don't have a valid version. It was noticed that the speed at which the packages arrived was too fast. This would cause transaction clouding on the Ethereum node. A queue was placed to improve the flow control between the workers and the listener.

Another CentOS instance was used to run the workers and the service. The messages are given to the workers in a round robin fashion. The workers are the processes which upload the binaries to the peer to peer storage and push the metadata on the Ethereum network. The workers can be horizontally scaled out to improve job throughput. The workers are decoupled from the listener.

Table 3: Software requirements.

Software	Version
Geth	v1.8.13
Solidity	v0.4.24
Node.js	v6.9.1
IPFS	v0.4.17
web3	v0.18.4

The smart contract models the dependencies graph of each package. A graph network can be visualized with contract storing all the dependencies and the dependents on the package. The public nature of the Ethereum makes it well suited for open source packages. The storage contract has data operations and is primarily the data layer of the system. As of now, web3 does not have a stable way of handling dynamic structures. Once the support for the this is stable a model contract can be used to leverage this feature. A model contract as shown in figure 6 can be used to improve the algorithm or add other interesting features on top of the storage layer. This pattern is also used to ensure that we do not lose the reference to our storage data as contracts are immutable and structures cannot be changed once the contract is deployed.

Web3 v0.20 was chosen over v1.0 beta because it's been around longer. Also, some of the experimental solidity features on string arrays were tried. The bytes array was found to be more stable than the string array in solidity. This, however, has an additional cost of converting all the byte elements to string elements after getting them from the contract.

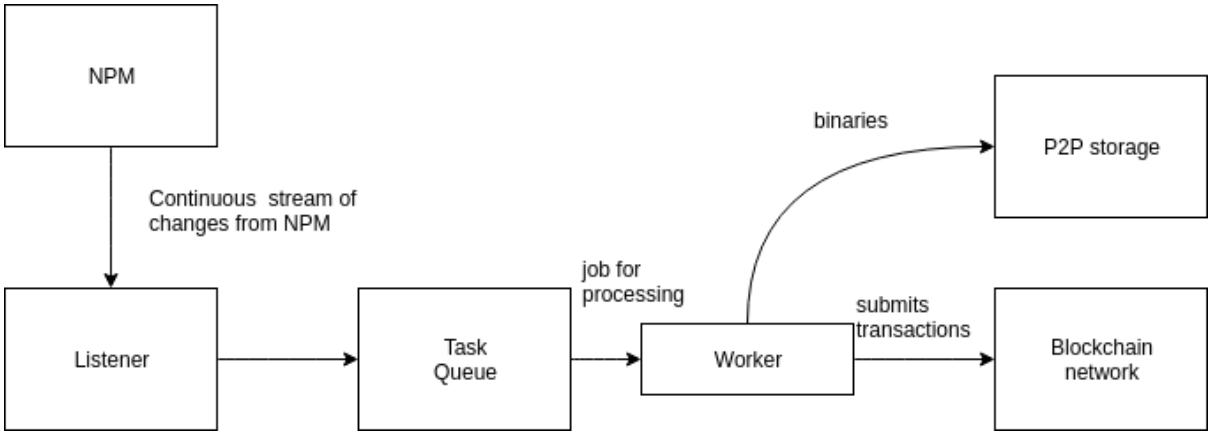

Figure 10: System Architecture for publishing new packages

A test simulator was used for the creation of the contract and initial testing. The test simulator has fake accounts and fake Ethereum which can be easy to check the functionality before deploying the main or any test Ethereum network.

The Rinkeby test network was used for the actual execution of the smart contracts. Rinkeby uses Proof of Authority, which means that there are authorized set of miners who verify transactions and add blocks. Ether needs to be expended while submitting transactions to the Smart contract. This Ether was received from the Ethereum Rinkeby faucet.

The block synchronisation of Ethereum nodes depends on other peers in the test network. The Ethereum node first synchronizes all the blocks and then starts accepting contracts and transactions. The full synchronization mode requires a lot of time as the chain is very large. The light mode, on the other hand, is the fastest option but does not have much support from peers. The fast synchronization mode just downloads the block headers instead of the entire block and it also has enough support from the community so the synchronization in this mode is fast.

The main network uses the Proof of work concept where the miners need to expend CPU to solve a computation problem and then add blocks to the network. The miners are rewarded with Ether once they solve the problem. The test network instead works on a proof of authority concept. It means there are predefined nodes which are used to verify transactions and adding blocks on the network. There are no miners on the test Rinkeby network. The implementation of proof of authority reduces the risk of the chain becoming big quickly because the authorized nodes include blocks at a fixed rate.

In all its essence, the Ethereum network is similar to a PaaS system. The contracts are distributed to all the miners, who execute the function with the transaction parameters for us and broadcast the results over the network. The clients pay for submitting transactions and deploying contracts, similar to how we pay the cloud service providers for deploying services on their platform. We do not control the environment in which the transaction is mined. In case we need more control on the mining activity, it would be good to opt for a private network rather than a public one.

6 Evaluation

Our evaluation has been performed using 4338 packages, including the top 950 packages which are used directly and transitively by other packages as documented in Kashcha (2018).

It has been noticed that if the Ethereum node is bombarded with transactions, the node does not broadcast those transactions immediately and these get stuck on the node as pending transactions. Miners get an incentive for mining blocks on the main network, so the main network has more miners. The Rinkeby network does not have any miners as it works on a Proof of Authority system. Only authorized nodes can verify transactions on the Rinkeby network. The authorized nodes ensure that the Rinkeby network grows at a steady pace.

Consequently, we have to set the gas price appropriately in order for the transactions to be mined quickly. Had the gas price been low, the transactions would not have been picked up for mining because the miners would have found other more lucrative transactions. Consequently, our initial gas price was 1 gwei for transactions and, by using a simple demand-supply iterative function, it was finally increased to 5 gwei.

Our tests ran for a week. It was initially noticed that the workers would crash because of insufficient funds. As the funds come from the faucet, which happens to be rate limited, we decided to slow submit transactions and keep adaptively funding the address with more ether.

Bandwidth monitoring was kept on the instance where the blockchain node was installed. The results are plotted in Figure 11. The evaluation involved requesting a list of packages along with their versions present on the node. The metadata for these packages was then requested individually.

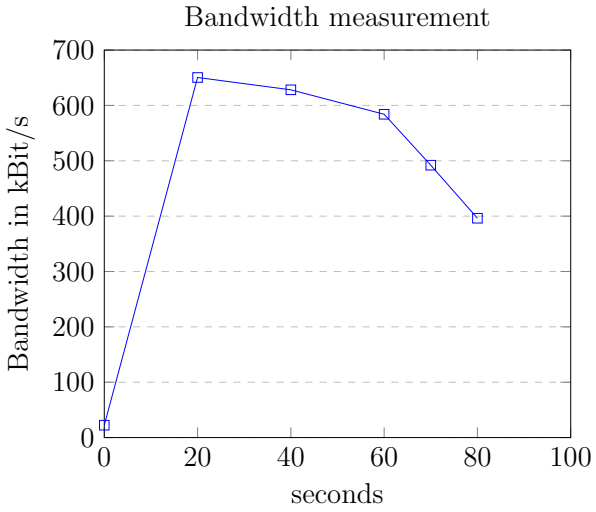

Figure 11: Empirical evaluation of Bandwidth usage

Nload was used to track the bandwidth of the instance. As seen in the figure, there is a sharp increase in the outgoing bandwidth, as much as 650.48 Kbit/s. The event happens because all the events are requested from the node. Requesting the events for the packages caused the bandwidth to spike up. The outgoing bandwidth gradually trickled down to 396.32 Kbit/s as other package metadata was requested.

Latency test was conducted on the Blockchain node. The results for the test are given in 4.

Table 4: Latency test for getting package meta data from the node.

Number of packages Head	**Statistics (ms)**		
	Mean	*Median*	**standard deviation**
250	148.3	146	8.73
500	149.578	147	9.23
1000	148.991	147	9.05

It was noticed that the mean latency to get the package metadata from the Blockchain node is approximately 148 ms. The cost to get the overall package would be higher as there is no handling for parsing dynamic structures on the web3 client. Once there is a stable support for dynamic structures, metadata for many packages can be requested at once.

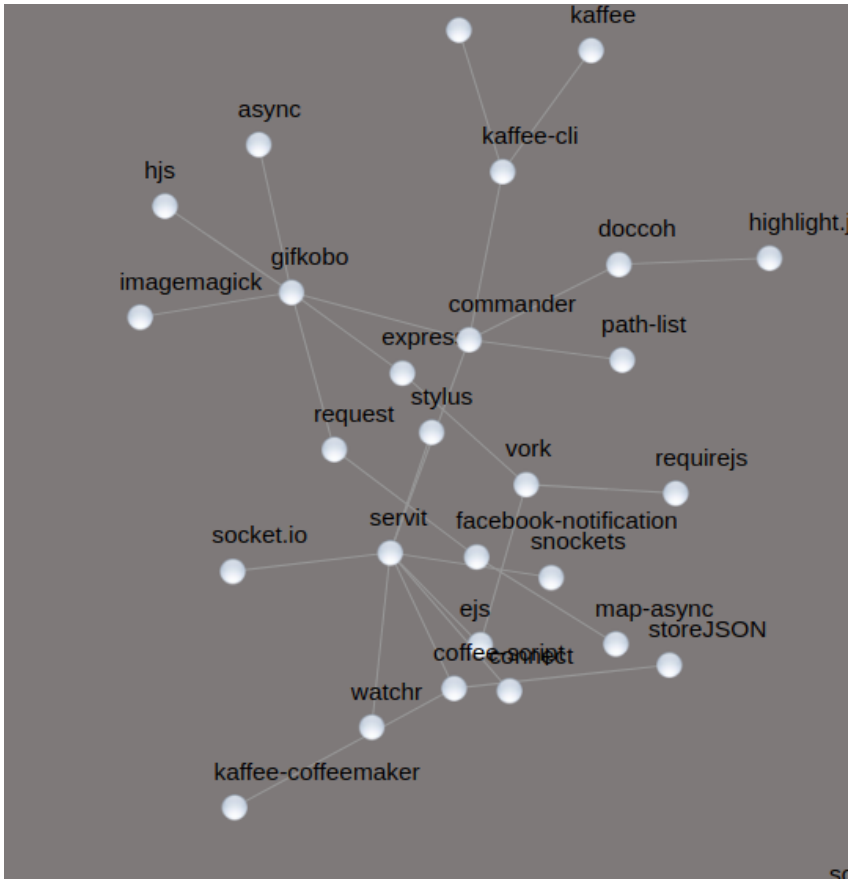

Figure 12: Network graph of packages (partial view).

Referring to 12, we can see at many packages in the Javascript ecosystem can have multiple levels of dependencies. The graph is modeled directly from the data received from the Blockchain node. It was noticed that some packages have cyclic dependencies. The impact of the cyclic nature of the dependencies has not been fully studied, but we argue that it may be detrimental to the overall traceability and version control for a repository.

The highest number of direct dependents observed in the subset which was tested is 361. Some packages such as `lodash`, `commander` and `body-parser` show an in-

creasing number of first level dependents. This number is likely to grow as previously shown Kashcha (2018). These packages are critical to the community as they form the basis of any sort of development.

7 Conclusion and Future Work

The proposed system is a reliable decentralised architecture for package management. The system relies on the Ethereum for functioning. Smart contracts have been used for integrating our logic onto the blockchain network, where IPFS has been proposed for storing actual binaries.

The blockchain network can have any number of nodes in the network at a time. The users can install the Ethereum node on their infrastructure or use it as a service in order to keep track of the dependencies. Also, the underlying architecture can be also be used for any kind of dependency management with tweaks to the client and version changes in the contract. The native client will, however, change according to the platform. Blockchain networks have good read throughout compared to the write throughput which could be ideal for dependency management.

Further work is envisioned to study dependencies at multiple levels as our current work has only taken into account single-level. Ideally, it may be useful to study in detail cliques which can uncover not only functional software properties, but also non-functional characteristics such as developer relationships, code styles, and other useful patterns.

References

Abate, P., Cosmo, R. D., Treinen, R. and Zacchiroli, S. (2012). Dependency solving: A separate concern in component evolution management, *Journal of Systems and Software* **85**(10): 2228–2240.

Alam, S., Kelly, M. and Nelson, M. L. (2016). Interplanetary wayback: The permanent web archive, *JCDL '16*, ACM, Newark, pp. 273–274.

Balalaie, A., Heydarnoori, A. and Jamshidi, P. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture, *IEEE Software* **33**(3): 42–52.

Benet, J. (2014). Ipfs-content addressed, versioned, p2p file system. Accessed on 19.2.2018.
URL: *https://arxiv.org/pdf/1407.3561.pdf*

Buterin, V. et al. (2014). A next-generation smart contract and decentralized application platform, *Technical report*. Accessed on 19.2.2018; Cited by 215.
URL: *https://github.com/ethereum/wiki/wiki/White-Paper*

Cox, R. (2016). Version SAT. Accessed on 19.2.2018.
URL: *https://research.swtch.com/version-sat*

Decan, A., Mens, T. and Claes, M. (2016). On the topology of package dependency networks: A comparison of three programming language ecosystems, *ECSA '16*, ACM, Copenhagen, pp. 21:1–21:4.

Highsmith, J. and Cockburn, A. (2001). Agile software development: the business of innovation, *Computer* **34**(9): 120–127.

Kashcha, A. (2018). Npm rank. Accessed on 25.07.2018.
URL: *https://gist.github.com/anvaka/8e8fa57c7ee1350e3491*

Krueger, C. W. (1992). Software reuse, *ACM Computing Surveys* **24**(2): 131–183.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Accessed on 19.2.2018; Cited by 3059.
URL: *https://bitcoin.org/bitcoin.pdf*

Nikitin, K., Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Gasser, L., Khoffi, I., Cappos, J. and Ford, B. (2017). CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds, *USENIX Security 2017*, USENIX Association, Vancouver, pp. 1271–1287.

Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment, *SIGPLAN Not.* **19**(5): 177–184.

Ruparelia, N. B. (2010). Software development lifecycle models, *SIGSOFT Softw. Eng. Notes* **35**(3): 8–13.

Schlueter, I. (2010). The node package manager and registry. Accessed on 19.2.2018.
URL: *https://www.npmjs.org*

Semantic Versioning user guide (2013). Accessed on 10.3.2018.
URL: *https://semver.org/*

Shvachko, K., Kuang, H., Radia, S. and Chansler, R. (2010). The hadoop distributed file system, *MSST'10*, IEEE, Lake Tahoe, pp. 1–10.

Toffetti, G., Brunner, S., Blchlinger, M., Spillner, J. and Bohnert, T. M. (2017). Self-managing cloud-native applications: Design, implementation, and experience, *Future Generation Computer Systems* **72**: 165 – 179.

Tucker, C., Shuffelton, D., Jhala, R. and Lerner, S. (2007). OPIUM: Optimal package install/uninstall manager, *ICSE '07*, IEEE, Minneapolis, pp. 178–188.

Wittern, E., Suter, P. and Rajagopalan, S. (2016). A look at the dynamics of the JavaScript package ecosystem, *MSR'16*, ACM/IEEE, Austin, pp. 351–361.