2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)

Distributed Software Dependency Management using Blockchain

Gavin D'mello, Horacio González–Vélez
Cloud Competency Centre, National College of Ireland
http://www.ncirl.ie/cloud
dmellogavin5000@gmail.com, horacio@ncirl.ie

***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 distributed infrastructure to provide immutability, and then a dependency graph is constructed with the help of smart contracts to trace the software provenance. Our system has been successfully tested with 4338 packages from NPM, 950 out of which are the top depended-upon packages.**

***Index Terms*—Software Reuse; Blockchain; Cloud Computing; Software packaging; Smart Contracts**

I. Introduction

Long considered a key practice in the industry, software reuse entails the creation of new software systems using existing software packages [1]. Typically composed of complementary software modules, most software packages are made available as common utility tools or frameworks which are used by millions of users which can in turn be used by other packages.

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 of wider range of interdependent software components [2].

Package managers provide a platform for code sharing. Reliable application package managers are of prime importance to software developers. Most packages need to be installed immediately 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 getting the package metadata or binaries could lead to build failures and hinder the development processes. An example of such a scenario is the left-pad problem discussed by [2] which led to many installation and build failures on NPM. Two percent of the transitive packages installations failed in this event.

While different software package managers offer mirrors and streaming, 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.

Widely considered immutable time-stamped data structures, blockchains implement peer-to-peer networks where participants can concurrently verify interactions using decentralised consensus protocols. Blockchain smart contracts allow us to store data and execute functions on them in such decentralised setup. Once a smart contract is deployed, transactions can be sent to the contract. In our case, transactions are the versions or new packages submitted by developers. The changes made by the transactions to our data structure are "mined" (verified) and broadcast to the entire network. A change once mined, cannot be reverted.

This paper is organised as follows. Section II discusses package managers and existing Blockchain systems. Section III outlines our proposed method to manage software packages with Blockchain, including the proposed algorithm for smart contracts. Section V 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 VI 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 modelled directly from the data coming from the blockchain node. Finally, Section VII presents some concluding remarks.

2377-5750/19/$31.00 ©2019 IEEE
DOI 10.1109/PDP.2019.00026

CPS Conference Publishing Services

Table I
LIST OF LANGUAGES AND THEIR PACKAGE MANAGERS

Language	Package manager
Java	Maven
Nodejs	NPM
Python	PyPI
C#	Nuget
PHP	Packagist
Ruby	Ruby gems

II. LITERATURE REVIEW

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 dynamic collaborative environments, rapid application development has increasingly become the norm for applicative environments [3]. 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 [4].

While software modularisation has been previously studied in the literature using different approaches [5], [6], the grassroots distributed administration of software packages remains an open problem. Code bases are constantly evolving over time and version control tools like Git and Subversion are widely used to manage versions and control changes.

Managing dynamic versions and software dependencies is complicated, as the program dependency graph for large programs is long known to be difficult to handle [7]. Microservice-based cloud architecture—where package dependencies can be linked to different packages—then increases the challenge at hand [8], 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 [9]. 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 [10], 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 [11], 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.

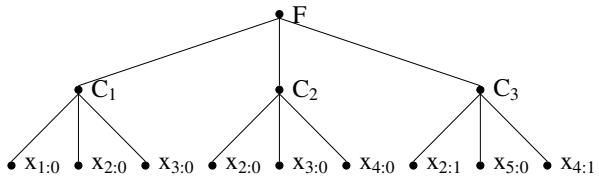


Figure 1. Transitive nature of packages.

Failure to understand these laws puts build systems at risk. Versions are divided into three parts major, minor and patch [12].

A security-oriented management framework, CHAINIAC has been used to verify integrity and authenticity for software-release processes based on decentralised nodes [13]. 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 big parts of the package are being rewritten. Minor versions are for new feature additions to the existing set without breaking changes. A patch is a bug fix which is backword compatible.

Table I 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 [14].

This work then 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 marshal the correct development and deployment for a given set of new packages. We have evaluated our work using 4338 packages from `NPM`.

III. METHOD

Every package manager has to deal with direct and transitive dependencies. The transitivity can be seen from 1. 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 [15]. If the x packages were to be removed, it would affect F and all C packages. The removal of x would lead to situation similar to left-pad explained by [2].

A. Data structure

The data schema can be best seen as a tree shown in 2. The root node acts the package name and the leaf node is the actual package information. The nodes are version numbers. The data structure provided in Figure 3 gives a summary of the data structure. Every package would have its own package

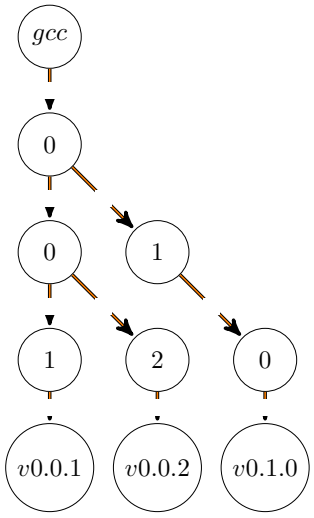


Figure 2. Placement of new versions on version tree

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

Figure 3. Crypto Assets stored for each version.

tree. This tree would evolve over time. The data structure is flexible to both the minor major versioning technique as well as semantic versioning. An oversimplification of this data structure would be to have to nested map pointing to a resource. Using an 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 binary, which will be stored using the Inter Planetary File System (IPFS).

Developers can host the Ethereum nodes in their environment or use a centralised service which gives them access to a node. 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.

B. 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 is a peer to peer storage solution. While uploading the package file to IPFS we receive an immutable hash. This 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.

We use a Coral Distributed hash table to store the data which concentrates more on locality [16]. 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 [17]. 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.

C. Algorithms

The algorithm outlined is used for storing a 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.

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 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.

D. Diagrams

Class diagram

Contracts are immutable in nature. In a centralized setup, we just update the code and deploy to all the servers. 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. All the package metadata would be stored within

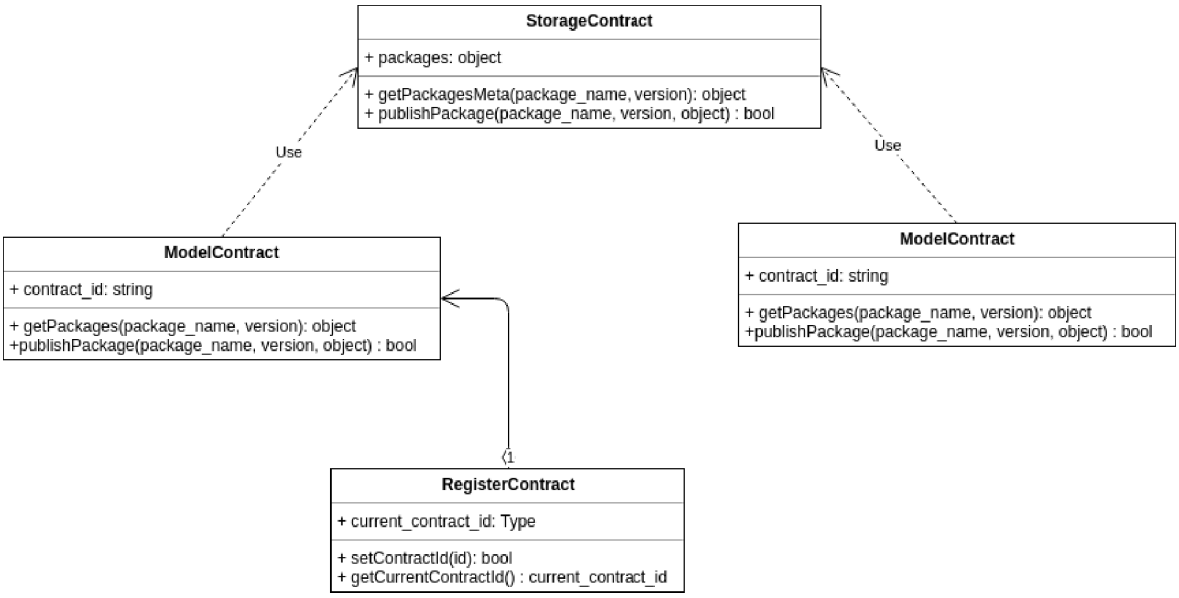


Figure 4. Class Diagram for the Contract Deployment Pattern.

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
```

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
```

the storage contract. The relationships with other packages are stored in the contract as well. Whenever new contracts are deployed, the client needs to be notified with the address of the new contract. This becomes an issue because an update would have to be released on every new contract change. 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.

Sequence diagrams

The installation sequence diagram shows the installation of a package. Figure 5 shows the interaction between the client, the node, and the IPFS server during an installation. 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 [18]. 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.

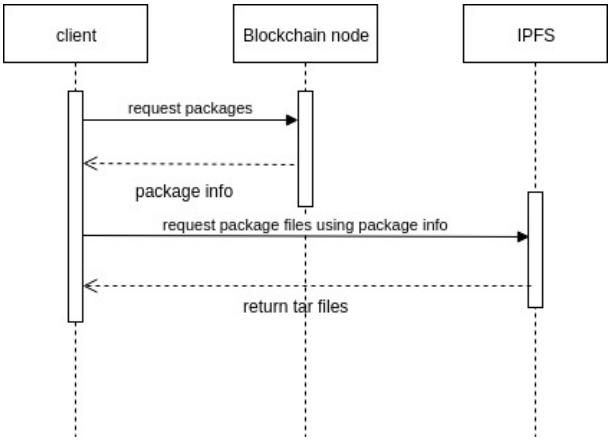


Figure 5. Package installation.

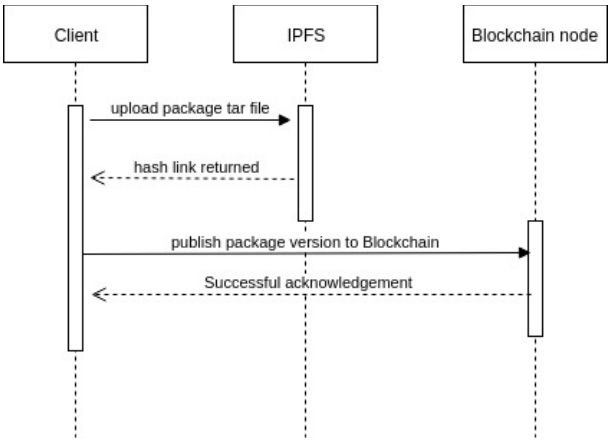


Figure 6. Publishing a package.

Figure 6 shows the interaction between the client, IPFS and the Blockchain node. The package to be published is converted to a tar file and 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.

IV. Proposed system

The proposed system employs blockchain-based smart contracts to decentralise the package management. Blockchain decentralised systems have demonstrated not to have a single point of failure and also improve resource utilization [19]. In our case, we have employed the blockchain network Ethereum, which uses Solidity for the creation of smart contracts.

The smart contract storage is used for the metadata of the packages. The contract is immutable and once the package is published it is marked in cryptographic stone. IPFS gives us peer to peer storage which is used for storing the binaries. The IPFS cluster module can be used over the IPFS server to ensure there are replicas of the binaries that are published. Table II shows a comparison of existing systems and proposed systems.

Table II
Comparison of existing systems and the proposed method

Variable name	**Different Package Managers**			
Head	*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

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 [20]. The system does 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.

Ethereum is considered as a natural extension of Bitcoin which is also a cryptocurrency. It is a Turing complete solution where the user can program and deploy bytecode to the Ethereum nodes [21]. 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 functions can be called from thin clients like browsers which we call decentralized applications. Transactions sent to the Blockchain are 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. The usage of distributed ledger ensures that reliability as changes are replicated across all the nodes. The downtime of some nodes does not affect the other nodes.

V. Implementation

Table III 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 OpenStack to run the Geth binary and also the IPFS server.

The setup can be used as a decentralised system or in collaboration with another package manager as shown in Figure 7. The decentralised system would require users identifying themselves to the network and would require sufficient funds to submit transactions. Every user would need an Ethereum address and would publish the packages with their own addresses. The collaboration with other package managers would allow anyone to submit transactions without the need of Ethereum addresses. The publishing would be done by a single user address which will be shared by the services. The collaboration mode does not require an Ethereum address from the user.

Table III
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 ownership model of the packages in a decentralised system would have every user owning their packages. This means that only the owner could add new versions to the packages. In the collaborative system, we would have a single user owning many packages. The collaborative system shown in Figure 7 can potentially be used where users need high reliability, but do not want create wallets to maintain packages.

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. A queue was placed to improve the flow control between the workers and the listener. This also improved the package publishing throughput of system.

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. The listener is a single point of failure in this architecture.

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. A model contract as shown in figure 4 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 has 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.

VI. RESULTS

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 [22].

It has been seen that if the Ethereum node is bombarded with low price transactions, these get stuck on the node as pending transactions. These transactions are removed form the pending transactions buffer once the transactions have been verified by the network.

Consequently, we have to set the gas price appropriately in order for the transactions to be mined quickly. If the price offered is low, the transactions will not be mined immediately 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 due to insufficient funds. As the funds come from the faucet, which happens to be rate limited, we decided to slowly submit transactions and keep adaptively funding the address with more ether.

Bandwidth monitoring was performed on the instance where the blockchain node was installed. The results are plotted in Figure 8. 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.

Nload was used to track the bandwidth of the instance. 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 Tables IV and V.

It was noticed that the mean latency to get the package metadata from the Ethereum node is approximately 148 ms. This test includes file I/O time. The test was done with the synchronous API. This approach is not scalable as the client libraries are run on user's operating system. A test was conducted with the asynchronous API which reduced the mean latency to under a millisecond. 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.

Table IV
LATENCY TEST FOR GETTING PACKAGE FROM THE SYSTEM.

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

Table V
LATENCY TEST FOR GETTING PACKAGE FROM THE SYSTEM USING ASYNCHRONOUS APPROACH.

Number of packages Head	**Statistics (ms)**
	Mean
250	0.748
500	0.67
1000	0.559

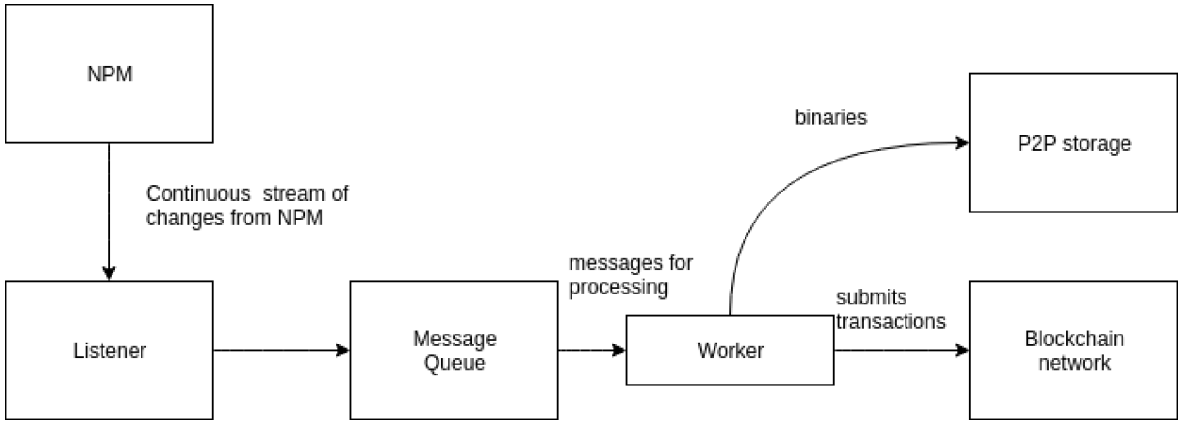


Figure 7. System architecture for publishing new packages

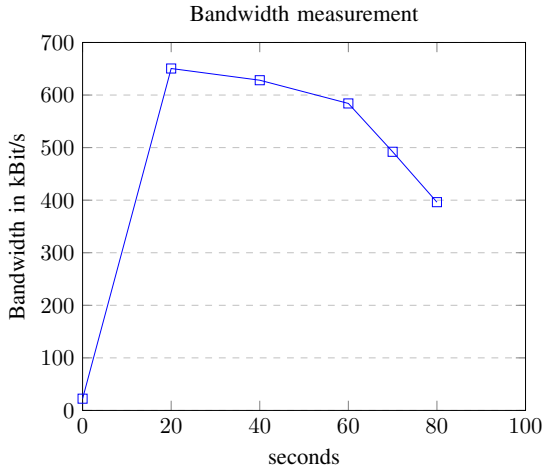


Figure 8. Empirical evaluation of Bandwidth usage

The block synchronisation of Ethereum nodes depends on other peers. The Ethereum node should be fully synchronized in order to serve data correctly. 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 and is experimental. The fast synchronization mode just downloads the block headers instead of the entire block and it also has better support from the community compared to the light mode.

The testing was done with single IPFS node. The IPFS cluster feature can be used to make sure there are replicas distributed across cluster peers in the network. An IPFS node which does not participate in the cluster can also request a binary from the IPFS network on an on-demand basis where it will have only a selected number of package binaries. A replication factor can be decided upon to replicate the packages to a minimum number of cluster peers.

Figure 9 presents a directed graph with all first level dependencies of the NPM repository. It is noted that a number of packages in this Javascript ecosystem can have multiple levels of dependencies. The graph is directly derived from the data received from the Ethereum node using first level dependencies only. 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 increasing number of first level dependents. This number is likely to grow as previously shown [22]. These packages are critical to the community as they form the basis of any sort of development.

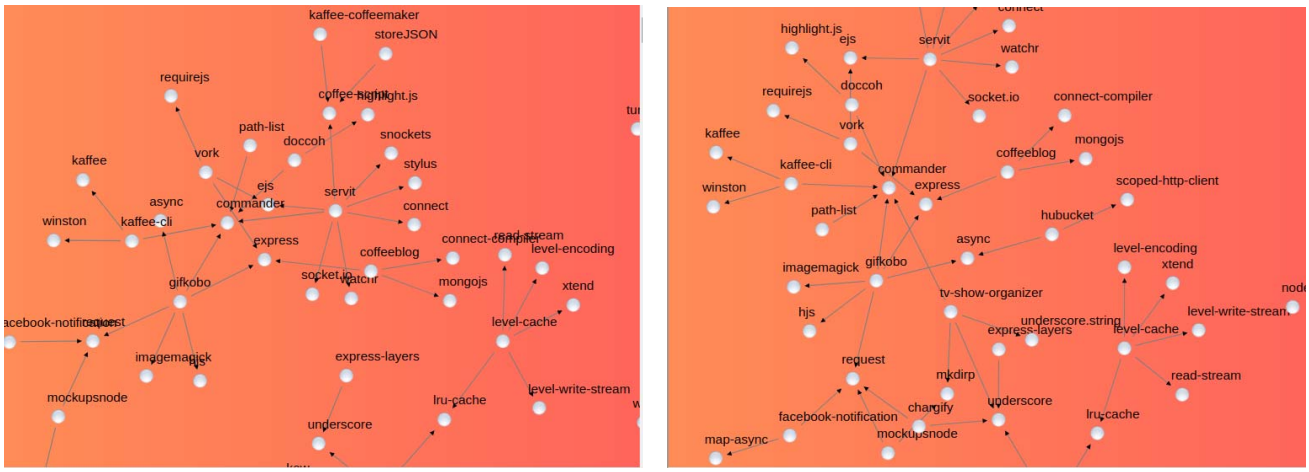
Some packages recorded in the ledger have cyclic dependencies. The impact of the cyclic nature of the dependencies has not been fully studied, but we argue that it may be detrimental to the overall traceability and version control for a repository.

VII. Conclusions

The proposed system is arguably a reliable decentralised architecture for package management. The system relies on the Ethereum to keep systematic track of software dependencies and provenance preservation. Specifically, smart contracts have been systematically used for integrating our logic onto the Ethereum network, where IPFS has been proposed for storing actual binaries. We strongly believe this work should eventually have some impact into the open-source ecosystem as it can be coupled into existing repositories to maintain a cohesive chain of version and dependencies in software packages.

The Ethereum 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. Ethereum blockchain network has a good read throughout compared to the write throughput which could be ideal for dependency management as the current state of the art receives billions of installation requests.

Further work should 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



(a) Step 1 (b) Step 2

Figure 9. 2-step sequence of the directed graph of packages generated by our system. The centrality of a number of packages (e.g. lodash, commander) identifies their importance within a specific repository, in this case NPM.

styles, and other useful traits. One distinct possibility is to build upon our previous work on software migration (pattern-based [23] and document to graph NoSQL databases [24]) to enable multi-level, multi-dependency component migration.

REFERENCES

[1] C. W. Krueger, "Software reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, Jun. 1992.

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

[3] N. B. Ruparelia, "Software development lifecycle models," *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 3, pp. 8–13, May 2010.

[4] J. Highsmith and A. Cockburn, "Agile software development: the business of innovation," *Computer*, vol. 34, no. 9, pp. 120–127, Sep 2001.

[5] D. Qiu, Q. Zhang, and S. Fang, "Reconstructing software high-level architecture by clustering weighted directed class graph," *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 04, pp. 701–726, 2015.

[6] R. Naseem, O. Maqbool, and S. Muhammad, "Cooperative clustering for software modularization," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2045–2062, 2013.

[7] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *SIGPLAN Not.*, vol. 19, no. 5, pp. 177–184, Apr. 1984.

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

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

[10] P. Abate, R. D. Cosmo, R. Treinen, and S. Zacchiroli, "Dependency solving: A separate concern in component evolution management," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2228–2240, 2012.

[11] I. Schlueter. (2010) The node package manager and registry. (Last accessed:1/Dec/2018). [Online]. Available: https://www.npmjs.org

[12] *Semantic Versioning user guide*, 2013, (Last accessed:1/Dec/2018). [Online]. Available: https://semver.org/

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

[14] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *MSR'16*. Austin: ACM/IEEE, May 2016, pp. 351–361.

[15] R. Cox, "Version SAT," 2016, (Last accessed:1/Dec/2018). [Online]. Available: https://research.swtch.com/version-sat

[16] J. Benet. (2014) IPFS-content addressed, versioned, P2P file system. (Last accessed:1/Dec/2018). [Online]. Available: https://arxiv.org/pdf/1407.3561.pdf

[17] S. Alam, M. Kelly, and M. L. Nelson, "Interplanetary wayback: The permanent web archive," in *JCDL '16*. Newark: ACM, 2016, pp. 273–274.

[18] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST'10*. Lake Tahoe: IEEE, May 2010, pp. 1–10.

[19] F. Hawlitschek, B. Notheisen, and T. Teubner, "The limits of trust-free systems: A literature review on blockchain technology and trust in the sharing economy," *Electronic Commerce Research and Applications*, vol. 29, pp. 50–63, 2018.

[20] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. (Last accessed:1/Dec/2018). [Online]. Available: https://bitcoin.org/bitcoin.pdf

[21] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," Tech. Rep., 2014, (Last accessed:1/Dec/2018). [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[22] A. Kashcha. (2018) Npm rank. (Last accessed:1/Dec/2018). [Online]. Available: https://gist.github.com/anvaka/8e8fa57c7ee1350e3491

[23] S. Boob, H. González-Vélez, and A. M. Popescu, "Automated instantiation of heterogeneous fast flow CPU/GPU parallel pattern applications in clouds," in *PDP 2014*. Torino: IEEE, Feb. 2014, pp. 162–169.

[24] A. Bansel, H. González-Vélez, and A. E. Chis, "Cloud-based NoSQL data migration," in *PDP 2016*. Heraklion: IEEE, Feb. 2016, pp. 224–231.