# Toward a distributed package management system

Fabien Dagnat, Gwendal Simon, Xu Zhang

# Toward Decentralized Package Management

Fabien Dagnat
Institut Telecom
Telecom Bretagne, France
fabien.dagnat@telecom-bretagne.eu

Gwendal Simon
Institut Telecom
Telecom Bretagne, France
gwendal.simon@telecom-bretagne.eu

Xu Zhang
Institut Telecom
Telecom Bretagne, France
xu.zhang@telecom-bretagne.eu

The mutation of the software economy toward crowdsourcing, the explosion of the number of devices and the increasing need for quickly-released software call for revisiting the way software are deployed and managed. The current approach adopted by most software package management systems is to rely on a single distributor, who collects packages from upstream sources, tests, releases and distributes them through a centralized channel, called *repository*. In this paper, we identify the major downsides of this centralized architecture and promote a distributed approach for software deployment. That is, in a network consisting of interconnected symmetric peers, all the developers are allowed to release and distribute software asynchronously. The discovery and retrieval of software is achieved through the communication among peers. We highlight the impacts of using such an approach and define a specific format of metadata that supports distributed package release.

## 1 Introduction

The size of the *free and open source software* community (two millions sourceforge users[1]) and the number of *application store* developers (90,000 so far[2]) illustrate the importance of *crowdsourced software*, *i.e.* software produced by a large number of loosely coordinated developers. At least two statements explain the sustained growth of crowdsourced software.

1. crowdsourced software has become a key economical argument. Apple typically takes advantage of the number of third-party applications available exclusively on its devices. The capacity to offer, as fast as possible, the largest and most diverse set of software is a critical need.
2. crowdsourcing enables fast and customized software evolution. Free software producers often distribute their programs in the form of source code, so that any developers having enough expertise can improve their functionalities according to users' need.

However, the deployment of crowdsourced software is a frustrating and error-prone task. Modern software is generally composed by a large number of inter-dependent components, called *packages*. A package can run correctly only when all its dependency requirements are satisfied and all the configurations are done properly. Installing or upgrading a package can break other packages or lead to unexpected performance of the system. The global consistency of a software distribution is extremely hard to maintain when the packages are independently produced. Facing this issue, the approach adopted by current

[1] http://en.wikipedia.org/wiki/SourceForge
[2] http://www.mobiledevhq.com/developers

Submitted to:
Lococo 2011

large systems such as the GNU/Linux distributions is to rely on a single distributor, who collects packages from upstream sources, approves them and publishes them together in a *repository*. Such centralized architecture enables the distributor to manage packages within a bounded space, so that the consistency of the composition of packages can be optimized [4].

In this paper, we argue the repository-based approach can not fulfill the need of crowdsourced software deployment. It exhibits some majors drawbacks that are intrinsically contradictory to the idea of crowdsourcing:

- managing a large repository for crowdsourced software is expensive. A recent analysis show that it costs Apple approximately $1.3 billion per year to run iTunes and its App Store. The investment is still increasing as more capacity is required to face the growth of software content.[3]
- relying on a single distributor is a supplier-side approach while software has become a fast moving consumer good (FMCG). Flexible and on-demand evolution is a highly desired feature offered by crowdsourced software. But the centralized administration leads to censorship which limits the degree of liberty developers can have. On the Web, it is common to see open platform developers complain about arbitrary software selection rules or unreasonable distribution fees.[4]
- approving and releasing third party software through a single channel is time consuming. It increases the time-to-market and degrades the availability of the newest products. Users are facing security risks due to delayed updates or bug fixes.[5] Many open source distributions have time-based release cycles, which are scheduled at slow pace. For Debian, the time between two consecutive releases is approximately $18-24$ months. Even packages in the unstable repository are continuously uploaded from the upstream sources, a large amount of users would prefer to install the stable distribution, which contains secure but obsoleted packages.
- repository-based approach can not fix the dependency hell. Maintaining the global consistency of the repository is not a scalable solution. To reduce the complexity of dependency resolution, GNU/Linux distribution repositories limit their number of packages (12,000 packages in average[6]). Mobile application stores have larger scale (200,000 for *Android Market* and 300,000 for the *App Store*[7]), but they do not handle package dependency at all. Packages on these platforms are generally self-contained small bundles. They are rarely allowed to declare dependency on others packages or no dependency check is done by the repository maintainer.
- repositories implies "boundaries" on software. Currently one repository (distributor) serves uniquely for one specific type of device or operating system. For instance, the iPhone App Store does not distribute applications for the Android Market. However, in the future Internet of Things, zillions of general hardware devices running heterogeneous software systems will be interconnected. Boundaries will disappear one day or another. It is important to have a uniform but scalable approach for deploying all kinds of packages.
- the dependence on a single point is an undesired property for both package storage and distribution. Users may be unable to upgrade their packages if the repository hosting the software stops providing the service.

[3] http://www.asymco.com/2011/06/13/itunes-now-costs-1-3-billionyr-to-run/
[4] http://www.southsearepublic.org/article/2438/read/apples_arbitrary_app_store_process/
[5] http://www.macworld.com/article/159978/2011/05/mac_app_store_security_updates.html
[6] http://oswatershed.org/
[7] http://en.wikipedia.org/wiki/List_of_digital_distribution_platforms_for_mobile_devices

As a conclusion, the central repository is a key limitation in the distribution and management of crowdsourced packages. To tackle the drawbacks listed above, this paper promotes using distributed approach for package management. We imagine a network consisting of interconnected symmetric peers, atop which all the developers are allowed to publish and distribute new packages and updates at anytime. Both the release and delivery of packages should be handled in a decentralized way. As a beginning of our work, this paper only describes the original idea. We identify the major changes that should be made to support distributed package release and propose a new metadata format needed for decentralized upgrade. At last, a blueprint of our future work is presented.

## 2 Impacts of Decentralization

Building a repository-less package management system (PMS) raises a number of open issues, as all the tasks of developing, managing, and deploying software are distributed across *peers* in the network. We define a peer as any device with a computational capacity, including servers, personal computers, mobile handsets, wireless sensors and any programmable object in the Internet of Things. In this paper, despite the heterogeneity of the population, we consider that all peers are symmetric in terms of functionalities and the roles they play in the software production life-cycle. A peer is not only a producer who creates, tests, releases and distributes packages, but also a consumer who uses software build from packages. Compared to conventional PMS, a repository-less approach differs in the following concerns.

- decentralized software evolution: the idea we would like to explore is that any peer should be able to release new versions of packages at any time. This statement will probably launch a wide debate among developers. Indeed, in component-based software, the communication between interdependent packages is hard coded into the source code and the binding is based on the component's name. Even if packages can explicitly specify dependency in their metadata, we can't guarantee the completeness of this information provided by untrusted third parties. An elegant method for dependency management is still lacking. But anyway, this issue also exists in repository-based PMS. As we will refer later, some research works has already been initiated to address it.
- upgrade without total order: in a centralized PMS, the version number of a package is conventionally encoded using an integer or a date denoting the total order among versions. This no longer works for distributed PMS as packages are produced independently and asynchronously, creating distinct branches in the evolution history. That is, neither the order is total, nor the version identifier uniqueness can be guaranteed. Hence, a new version encoding scheme must be proposed. It needs to be mentioned that, in current FOSS distributions, packages already have tree-like evolving histories. Once a package is distributed in an official release, it needs to be maintained independently from the development line. At this point, a distinct upgrading branch is derived. In the Debian bug tracking system, the whole evolution history of a package is recorded in the change log so that the branches of versions can be tracked. However, this information is not contained in the package metadata and the meta-installer has no mean to distinguish versions in different branches.
- policy free release management: releasing software independently is easy when dealing with self-contained small packages maintained by individual or small group of developers. However for larger projects like Debian, GNOME and OpenOffice, the production life-cycle relies on the coordination of a large number of volunteers[8]. Releases continually produced by untrusted outsiders of the projects will inevitably impact the project's image, even under the assumption that they are

[8] `http://www.cyrius.com/journal/`

all certified to have enough quality and do not violate any dependency on the official packages. Hence, they should be signed and distinguished from the official releases. A project can still maintain its own repository and release policy. The repository is however considered as a powerful *peer*, which has no privileged authority over other peers.

- distributed data storage and retrieval: packages are stored on peers across the network. Searching and delivering packages should be achieved through the coordination of peers. Packages should not be removed immediately after being unpacked as done in most Linux operating systems. To guarantee fast and reliable retrieval, non-installed packages need also to be cached. No repository means no central point to consult for new package upgrades. The discovery (notification) of new product should be driven by user's interest. In this content, social networks of users and products should arise and become the support of modification propagations.

## 3 Related Works

To the best of our knowledge, no previous studies have discussed the overall approach for distributed package management. But each of the aspects mentioned above has been addressed seperately. [4] describes a repository-based dependency management approach. The authors formalized the problem of testing whether a package is installable within a collection of packages. They model each package dependency as boolean variable whose value specifies whether it is installed. The installability of a package therefore can be verified by testing the conjunction of all its dependencies. The complexity of solving such a SAT problem is NP-complete and is time consuming when the repository is large. Nix [3] is a purely functional component deployment tool, it allows different versions of a same component to be installed side-by-side, thus supports non-destructive upgrade. Components are unpacked into isolated directories, which are indexed using the cryptographic hashes of all the inputs needed to build and run them. By doing so, Nix guarantees the completeness of dependencies specification and avoids interference between components sharing common dependency. However, the problem of decentralized upgrade is not covered in this work. In [5], the authors propose an architectural-based approach to enable decentralized software evolution. They use connectors as communication routers to bind components, which enables changing the binding decisions without altering the components. As architectural descriptions only focus on the high level structural view of component, they are not expressive enough to reflect physical constraints on components and the environment. At last, [6] highlights the importance and difficulty of continuously releasing updates to users. It presents an algorithm that automatically generates incremental updates from build and testing processes. This approach requires all the dependencies and reverse dependencies of the packages to be known, thus can't be extended to repository-less deployment.

## 4 A Metadata Format for Decentralized Upgrade

In this section, we formally define the essential metadata needed for distributed package management. Instead of using a pair of name and version number, we use the cryptographic hash of the package data as its identifier. The package's entire evolution history is contained in the metadata. By using *Bloom filters*, two package histories can be compared efficiently to decide whether one package is an upgrade of the other.

**Definition 1.** *A* package *$p$ is a pair $(pid, vid)$ where $p.pid$ encodes the family of the package and $p.vid$ identifies its version.*

A sufficiently long hash (*e.g.* the 128 bit MD5 cryptographic hash) of the binary package data can be used as the *vid* to ensure its uniqueness. A package can be identified by using only the *vid*; the *pid* is however needed for denoting a group of packages derived from a single root. Without version identifier in a total order, the inheritance relation among packages can't be explicitly extracted from the *vid*.

**Definition 2.** *The* replace *relation between two packages p and q, noted $p \succ q$, means that the package p (respectively q) is a direct successor of q (resp. ancestor of p).*

A given package can have several successors if new packages are concurrently released. In order to prevent an explosion of release branches, we also support *merging* several package versions, *i.e.*, having several *replace* relations in the same package. Note also that *replace* can also implement a *subsumption* relation, that is a package $p$ can replace a package $q$ with $p.pid \neq q.pid$. The evolution history is thus represented by a *Directed Acyclic Graph* (DAG).

**Definition 3.** *The* history DAG *of a package family P is the directed graph $G^P = (V,E)$ such that V contains $(i)$ all packages p with $p.pid = P$ and $(ii)$ all packages q with $p \in V$ and $p \succ q$. A directed edge $(q \rightarrow p)$ in E is equivalent to $p \succ q$.*

With the subsumption relation, the history DAG can have several roots, and packages with different *pid* may belong to the same history DAG. A package $p$ may exist in several history DAG, but there exists a unique history DAG $G^P$ such that $p.pid = P$. We call it the history of $p$. We can then deduce the set of ancestors of a package $p$ from the history of $p$.

**Definition 4.** *The* ancestors *of a package p are all packages q such that there exists a path from q to p in the history of p.*

Hence it is possible to check whether a package $p$ upgrades an installation by testing whether an installed package is ancestor of $p$.

The size of the ancestor set grows constantly. Transmitting this information requires extra bandwidth and looking up in large data set is time-consuming. A scalable solution is to associate with each package a compressed metadata, called *ancestor summary* (or summary in short), from which it is possible to know whether an installed package belongs to the ancestor set. A *Bloom filter* is a good candidate: it has constant length and it supports efficient membership tests. A Bloom filter may yield a false positive suggesting that an element is in a set even though it is not. This does not induce serious problem as before really upgrading, a peer can verify the replace relationship using the complete history. Hence false positive only causes useless download but not installation of package.

By definition, a *Bloom filter* representing a set $S$ of $n$ elements is a vector of $m$ bits, initially all set to 0. For each element $x \in S$, $k$ independent hash functions $h_1, h_2, \ldots, h_k$ are used to encode the element (string) into an integer over the range $\{1, \ldots, m\}$. All bits $h_i(x)$ are set to 1, $1 \leq i \leq k$. To check whether an element $y$ is in $S$ is checking whether all the bits $h_i(y)$ are set to 1, $1 \leq i \leq k$. By choosing appropriate values of $k$ and $m$, the false positive rate can be limited to a negligible small value.

Note also that *augmenting* the *replace* relation with information is possible, *e.g.* developer updates, minor updates and major upgrades [**?**].

Finally, release management depends on user behavior and community rules. The emerging usages are unpredictable: exponential growth of the number of versions, development of merge bots, encapsulation of dependent packages in order to prevent incompatibility with future versions, and so on.

## 5 Perspective Works

In order to test our proposal, we are currently implementing a prototype of decentralized package manager. More precisely, a peer-level simulator have been developed. It emulates the logic of all the opera-

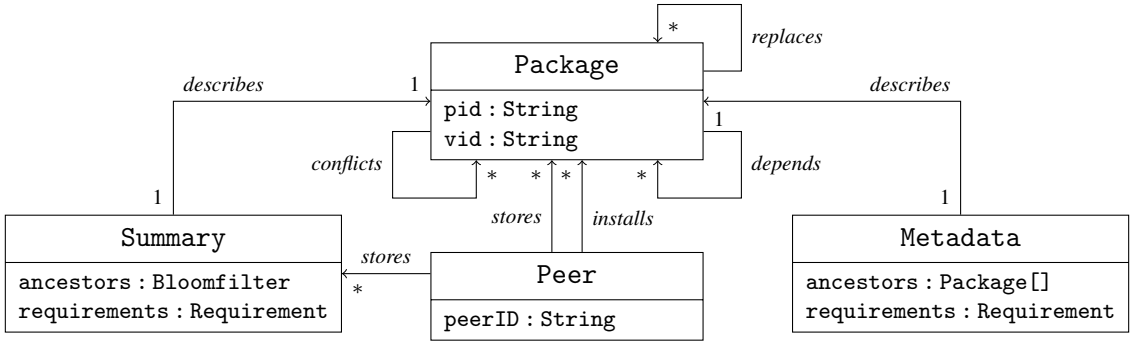


Figure 1: Conceptual model of a repository-less package management system

tions performed by a peer to locally install, upgrade or remove a package. The packages being deployed are old Debian packages collected from the Debian Snapshot Repository[9], with their metadata transformed into our specific format. Yet we can not evaluate our system on a real decentralized upgrading scenario as Debian packages do not have branches in their evolution history. In a next step, we plan to acquire source packages directly from upstream sources and combine our prototype with the Nix[10] package manager. Research on higher level functionalities is also going on. Some peer-to-peer algorithms will be implemented in our future works to enable the notification and delivery of packages.

## 6 Open issues

The design and implementation of a complete system requires investigation on a wide range of topics. Many open issues are left for future research.

**Inter-dependent devices** Pervasive software often require to be deployed onto more than one devices. For example, a digital home management software spread over a lot of captors, actuators and the various electronic devices of a house. In such a context, updating a package on one peer may force another peer to update some of its packages as the interface between packages must remain compatible. We already realized early work on the problem of coordinated dependency resolution for the deployment of distributed component-based applications in a LAN. But, the complete formalization of this approach and its adaptation to the Internet of Things remain open issues.

**Certification mechanism and failure recovery** Package management security is essential to the overall security of the computing system [1]. As any peer can release a new package at anytime, it is impossible to prevent the spread of malware. A distributed trust system is required for evaluating packages. Even though we are not able to propose a precise solution by now, we believe that a reputation evaluation approach based on social network could be used to select the most trusted peers [7]. The more a peer is trusted, the better it will be considered as a source for download. However, peer-to-peer trust and reputation systems have not been yet implemented at a large scale. The security requirements and the perspective to scale to the Internet of Things requires major progresses in the evaluation of these systems. Besides a certification mechanism, the system must also be able to prevent failure and to recover if any

[9] `http://snapshot.debian.org/`
[10] `http://nixos.org/nix/`

failure arise. The recent work of [2] on using a model-driven approach to simulate upgrades combined with roll-back mechanism concentrates on unintentional failure but could serve as a basis to explore a more defensive approach.

## References

[1] Justin Cappos, Justin Samuel, Scott Baker & John H. Hartman (2008): *Package Management Security*. Technical Report, Department of Computer Science, University of Arizona.

[2] Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio & Stefano Zacchiroli (2010): *Supporting Software Evolution in Component-Based FOSS Systems*. Science of Computer Programming .

[3] Eelco Dolstra (2005): *Efficient upgrading in a purely functional component deployment model*. In: Component-Based Software Engineering, pp. 219–234.

[4] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak & Xavier Leroy (2006): *Managing the Complexity of Large Free and Open Source Package-Based Software Distributions*. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Ingineering (ASE'06), IEEE Computer Society Press, pp. 199–208.

[5] Peyman Oreizy (1998): *Decentralized software evolution*. In: Proceedings of the International Conference on the Principles of Software Evolution (IWPSE 1).

[6] Tijs Van Der Storm (2005): *Continuous release and upgrade of component-based software*. In: Proceedings of the 12th International Workshop on Software Configuration Management (SCMC12), ACM Press, pp. 43–57.

[7] Li Xiong & Ling Liu (2004): *PeerTrust: Supporting Reputation-Based Trust for Peer-to-Peer Electronic Communities*. IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING 16, pp. 843–857.