dPAN: distributed package management network

Xu Zhang

To cite this version:

Xu Zhang. dPAN: distributed package management network. Distributed, Parallel, and Cluster Computing [cs.DC]. 2010. dumas-00530795

HAL Id: dumas-00530795

https://dumas.ccsd.cnrs.fr/dumas-00530795v1

Submitted on 29 Oct 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

dPAN: distributed package management network
Internship report

ZHANG Xu
xu.zhang@telecom-bretagne.fr
Guided by: Fabien Dagnat, Gwendal Simon

August 16, 2010

Abstract

In this paper, we propose a fully decentralized system for software package management and distribution. Comparing to existing repository-dependent approaches, our system addresses more attention on the massively growing User-Generated software content and the emerging Internet of Things. We first illustrate some statistical results and reveal the major properties of package distribution. Then we revisit the current structure of software packages and formally define our system model. After this, several candidate application-level protocols for efficient package retrieval and advertisement are investigated. At last, some open issues concerning the package certification and the heterogeneity of participants are highlighted for future research.

Keywords: package management system, user-generated content, overlay networks, peer-to-peer systems, Internet of things

Contents

1 Introduction **3**

2 Background **4**

3 Package statistics **6**
3.1 Package dependency graphs 7
3.2 Software package characteristics 8

4 Related works **10**
4.1 Overview 10
4.2 Limitations 11

5 A repository-less package management system **12**
5.1 Distributed system model 12
5.2 System functionalities 15

6 Application level protocols **16**
6.1 Publish/Subscribe-based metadata dissemination 17
6.1.1 General model of Publish/Subscribe systems 17
6.1.2 Subscription model 18
6.1.3 Distributed Notification Service 20
6.2 Gossip-based metadata dissemination 21
6.2.1 Construction of a content/interest-proximity based overlay network 22
6.2.2 Proximity function 24
6.2.3 Bloom filter 25
6.3 Content-based package searching 27
6.3.1 Freenet file-sharing network 27
6.3.2 Content Centric Network 29
6.3.3 Our proposal 31

7 Remaining challenges **32**
7.1 Device Heterogeneity 32
7.2 Certification Mechanisms 32

8 Conclusion **33**

List of Figures

1 Package dependency graph metamodel 5
2 Package size distribution 9
3 CDF of package popularity in Debian 9
4 Distribution of numbers of version of Debian packages 10
5 Package dependency graph metamodel 13
6 Generic model of Publish/Subscirbe system 18
7 Publish/Subscirbe routing strategies 21
8 Two layer structure for overlay topology construction 23
9 Example of Bloom filter 26
10 Request routing in Freenet 28
11 The CCN protocol stack 29
12 The CCN data and interest packets 30
13 The CCN node architecture 30

List of Tables

1 Statistics obtained from two FOSS repositories 8
2 The CYCLON protocol 24
3 The VICINITY protocol 24

1 Introduction

User-Generated Content refers intuitively to media content, and by extension to new services offered by some content aggregators to end-users over the Web. There is a less popular but very impacting class of content, which is also massively produced: *software*. The number of software that are daily created or updated is overwhelming: the community of *Free and Open Source Software (FOSS)* contains typically several millions of software producers (from amateurs to professionals). The increasing popularity of *application stores* (*e.g.* more than 225,000 applications in the Apple AppStore for a total of 5 billions downloads since its inception two years ago[1]) confirms several critical trends in the software industry:

- *crowdsourced software has become a key economical argument.* Apple typically takes advantage of the number of third-party applications that are available exclusively on its devices. The capacity to offer, in a very short time, the largest and most diverse amount of innovative software and services is a challenge. In this context, most actors of the communication industry, including phone manufacturers and network operators, propose incentives for developers (from monetary compensation to open access to data and API), which tend to reinforce the proliferation of new software.

- *pervasive environments need crowdsourced software.* The explosion of the number of devices, as well as commercial issues (especially the time-to-market), induce a gigantic demand for software development. Actually, this demand exceeds by far the capacity of classic software producers. For example, the strength and dynamism of the Linux community is a key factor explaining the rising popularity of Linux OS for small devices.

In comparison to classic content aggregation, the management of *user-generated software* appears however to be a challenging task. Indeed, modern software often consist of a huge number of small packages. These packages have inter-dependent relationships that may easily be broken during the deployment life-cycle. Thus finding an efficient and reliable way to maintain, distribute and install these software packages over billions of machines is definitely an issue. In current approaches, software distributors rely on centralized servers, the *repositories*, which are collecting all the packages that have been certified. We distinguish two major drawbacks in this architecture:

- *the certification of packages.* The software distributor plays the role of a certification authority. Users must submit their packages if they want them to be integrated into the repositories. The distributor verifies the integrity of the submitted packages and makes the valid ones available for other users to download. As addressed in [1], there exist various approaches and tools facilitating the management of large repositories of packages. However, the centralized nature requires expensive

[1] `http://www.appleinsider.com/articles/10/06/07/apple_says_app_store_has_made_developers_over_1_billion.html`

infrastructure and costly human management. The process of certificating third party packages is slow and complex. More and more developers complain about the increasing delay for software availability in the Apple AppStore[2]. Clearly, a centralized certification of packages does not scale.

- *the delivery of packages.* It has been emphasized by Microsoft researchers [2] that a set of repositories can not ensure a fast, planet-scale, delivery of packages. However, massive delivery of software patches is a key security requirement. If the number of devices grows as it is commonly admitted, the limits of a centralized repository-based architecture will soon be reached. Moreover, devices in pervasive environment are not necessarily always connected to the Internet. We need to also rely on intermediate devices and opportunistic ad-hoc communications if one wants to upgrade all devices, including the tiniest ones.

To fill the gaps between the current approach and the exigence imposed by the future software commercial ecosystem, this internship revisit, in a clean-slate approach, the package management system. As we will show later, the conception of a fully distributed (repository-less) system presupposes a dramatic modification on the common inter-dependent relationships between packages. Several distributed, autonomous algorithms should also be designed, and comprehensively tested over realistic platforms. The resulting system should guaranty scalable and reactive package upgrades. This challenge is considerable, but addressing it is inevitable in the perspective of the digital society.

The reminder of the report is organized as follow: in the next section we review the basis of classic package management systems; then we present some statistics on software packages in Section 3. Section 4 presents related works; Section 5 gives a new formalization and describes the main functionalities of our system; after this, several potential schemes for package searching and metadata dissemination are discussed in Section 6; the remaining issues on package certification and peer diversity are addressed in Section 7; at last Section 8 concludes the paper.

2 Background

Modern software distributions are generally composed by tens of thousands of packages, *e.g.* more than $25,000$ in *Debian* distribution[3]. Each package is a bundle containing some necessary files to compile or run a program. Packages embed metadata to provide information such as the package's name, version, a description of its functionalities as well as its dependency requirements. Due to the reusable nature of component based software, a package often depends on other packages to function correctly, it may also conflict with some others. On a device, the set of installed packages and their inter-dependencies is often represented by a graph called the *dependency graph.* This graph must remain consistent otherwise the system of the device is corrupted and (some of its functionalities) may become unusable. Therefore, while a package is installed, upgraded

[2] `http://www.paulgraham.com/apple.html`
[3] `http://www.debian.org/doc/debian-policy/index.html`

or removed, the dependency graph must be maintained consistent. Today, for most Debian Linux system administrators, using `apt-get` to install or update packages has become a daily operation. However, behind this single-line command, there is the whole process of resolving package dependencies, fetching packages from the Internet, installing them on the local system and configuring them to be fully functional. This complex task is handled, in most current operating systems, by a package management system in a fully transparent way.

Classical package management systems are usually composed by a (meta-)installer and a package distribution network that relies on central repositories/mirrors. The meta-installer discovers and downloads package/metadata from the repositories, locally resolve dependencies and determines whether the package is installable with respect to the environment it is deployed onto. The repositories/mirrors are managed by the software distributors who are also responsible for maintaining the global consistency of the distribution, *e.g.* no alternative versions of a package present in a single release; no uninstallable package in a snapshot *etc.* Figure 1 illustrates the generalized metamodel of a package repository.

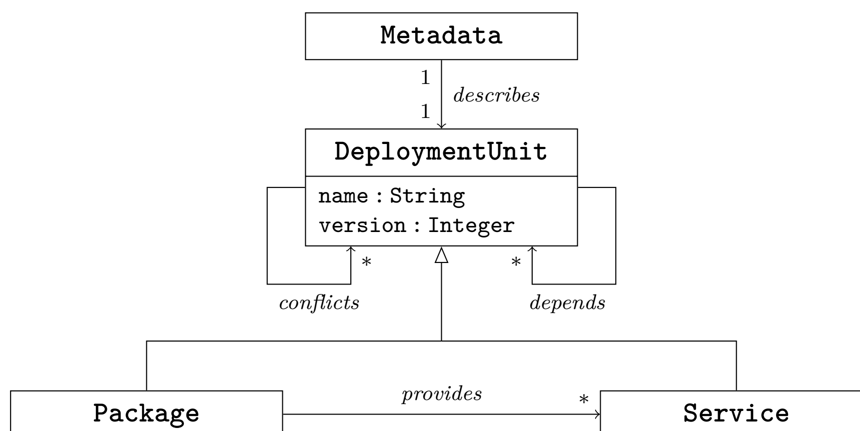


Figure 1: Package dependency graph metamodel

Here we generalize the concept of *packages* to *deployment units* that may either be *packages* or *services*. A *service* provides an abstraction of a functionality that may be provided by several packages. It coincides with *virtual package* of Debian distributions [1] and allows to express disjunctive requirements. For instance, when a package requires a service s, any package providing s may be used to fulfill this requirement.

Each deployment unit embeds metadata describing all its inter-unit relationships as well as other relevant information. To simplify, our presentation focuses on the essential relationships between packages: *depends* and *conflicts* [3]. If a package p *depends* on another package p' (or a service s), then p' (or any package providing s) must be present in the system in order to make p fully functional. When a package p *conflicts* with a package p', they cannot be both installed in a system.

Each deployment unit exists in several versions forming a total order. A requirement

may relate to any version of a package. It is specified by constraining the name of the package with a version constraint using any usual comparison operator ($<, \leq, =, \geq$ *and* $>$). Given all the stored metadata, the previously mentioned package dependency graph may be defined as a directed labeled graph $G = (V, E, L_v, L_e)$. Each vertex $v \in V$ denotes an individual version of a deployment unit and each edge $e \in E$ represents a dependency relation between two units. the vertex label $l_v \in L_v$ is the identifier of the unit and the edge label $l_e \in L_e$ denotes the relation type. The aforementioned concepts can be formalized as follow.

Definition 1. *A **deployment unit** is a pair $u = (n, v)$ where n is its name and v its version number.*

Definition 2. *A **repository** is a tuple $R = (U, D, C)$ where U is a set of deployment units, $D : U \to \mathcal{P}(U)$ is the dependency function associating to a unit the set of units it depends on ($\mathcal{P}(U)$ denotes a subset of U) and $C \subseteq U \times U$ is the conflict relation.*

Definition 3. *An **installation** of a repository $R = (U, D, C)$ is a subset I of U, giving the set of packages installed on a system. An installation is **healthy** when the following conditions hold:*

1. Abundance*: every package has what it needs. Formally, $\forall \pi \in I, D(\pi) \subset I$.*

2. Peace*: no two packages conflict. Formally, $(I \times I) \cap C = \emptyset$.*

Definition 4. *Three operations are defined on packages, installation, upgrade and removal:*

1. *A package π of a repository R is **installable** in an installation I, if and only if there exists a healthy installation I' of R such that $I \cup \{\pi\} \subset I'$.*

2. *A package π of an installation I can be **upgraded** if and only if there exists a package π' in R such that $\pi' > \pi$ and π' is installable on $I \setminus \{\pi\}$.*

3. *A package π of an installation I can be **removed** if and only if $I \setminus \{\pi\}$ is a healthy installation.*

With this formalized model, the problem of determining the installability of a given package can be transformed into a well known satisfiability (SAT) problem by simply interpreting a package π as a boolean variable with the intuitive meaning that the package π is installed in the chosen solution or not. The complexity of a SAT problem is NP-complete and there exists many SAT solvers that can resolve such a problem.

3 Package statistics

As introduced in the last section, software packages are simply bundles of files. It is intuitive that a package management system shares a lot of communality with most file-sharing and content delivery systems. However, package management also exhibits many

particularities due to the special characteristics of software packages. In this section, we aim at studying the common properties and global convergence of the software packages, and present some statistical results. Some of these results are extracted from existing research works, some others are obtained through our own effort. The main purpose the statistics are to illustrate:

1. how a real package dependency graph looks like (its size, connectivity, distribution of degree *etc.*)?
2. what are the characters of software packages and metadata(their size, popularity, frequency of update and similarity between different versions)?

3.1 Package dependency graphs

package dependency graphs exhibits several non-trivial properties such as scale-free degree distributions and the small-world structure. The degree of a vertex v, denoted k, is the number of vertices adjacent to v, or in the case of a digraph either the number of incoming edges or outgoing edges, denoted k_{in} and k_{out}. A package dependency graphs is scale-free means that, its distribution of edges roughly follows a power-law: $P(k) \propto k^{-a}$. That is, the probability of a vertex having k edges decays with respect to some constant $a \in \mathbb{R}^{+}$. The small-world effect states that $C_{random} \ll C_{sw}$ and $L_{random} \approx L_{sw}$, where C is the clustering coefficient of a graph, and L is the characteristic path length. The clustering coefficient is a measure of degree to which nodes in a graph tend to cluster together. The local clustering coefficient of a vertex in an undirected graph is given by $C_i = \frac{2|\{e_{jk}\}|}{k_i(k_i-1)} : v_j, v_k \in N_i, e_{jk} \in E$, where N_i is neighborhood of vertex i, $|\{e_{jk}\}|$ is the number of links between the vertices within the neighborhood N_i. The clustering coefficient for a graph is the average over all vertices, $C = \frac{1}{n}\sum_{v \in V} C_v$.

In [4], statistical results have been empirically obtained by mining two well-known FOSS repository: the Debian GNU/Linux software repository and the FreeBSD Ports Collection. The Debian network contains $n = 19,504$ packages and $m = 73,960$ edges, giving each package an average coupling to 3.79 packages. For the Debian network, $C = 0.52$ and $L = 3.34$. This puts the Debian network in the small-world range, since an equivalent random graph would have $C_{random} \approx 0.0019$ and $L_{random} \approx 7.41$. There are 1,945 components, but the largest component contains 88% of the vertices. The rest of the vertices are disjoint from each other, resulting in a large number of components with only 1 vertex. The diameter of the largest component is 31. The distribution of outgoing edges, which is a measure of dependency to other packages, follows a power-law with $K_{out} \approx 2.33$. The distribution of incoming edges, which measures how many packages are dependent on a package, follows a power-law with $K_{in} \approx 0.90$. While 10,142 packages are not referenced by any package at all, the most highly referenced packages are referenced thousands of times. 73% of packages depend on some other package to function correctly.

The BSD compile-time dependency network contains $n = 10,222$ packages and $m = 74,318$ edges, coupling each package to an average of $k = 7.27$ other packages. For

	Debian	*BSD*
n	$19,504$	$10,222$
m	$73,960$	$74,318$
$\|\Omega\|$	$17,351$	$7,441$
α_{in}	0.9	0.62
α_{out}	2.33	1.28
C	0.52	0.56
L	3.34	2.86

Table 1: Statistics obtained from two FOSS repositories

the BSD network, $C \approx 0.56$ and $L \approx 2.86$. An equivalent random graph would have $C_{random} \approx 0.007$ and $L_{random} \approx 7.11$. Hence, the BSD network is small-world. The degree distribution of the BSD network also resembles a power-law, with $K_{in} \approx 0.62$ and $K_{out} \approx 1.28$. For the run-time network, results were similar: the run-time network is both small-world and follows a power-law.

The measures described above are based on large repositories. In our research, we are also interested in the properties of dependency graphs installed (or stored) on individual peers. We have developed several simple Python scripts, which may be executed on the Linux Ubuntu OS. The scripts extract information from the current installation, and calculate the aforementioned measures. For example, on my personnel laptop, there are 1240 packages installed, which constitute a dependency graph containing $10,318$ edges, among which $7,431$ are *Depends* relations, $1,400$ are *Conflicts* relations , 433 are *Provides* relations and $1,054$ are *Replaces* relations. The characteristic path length of the installed package dependency graph is 2.36315884803, and the average clustering coefficient is 0.423166723898. Comparing to random graph, the installed dependency graph exhibits the "small world" property. There are 8 connected components in total and the biggest one contains 1232 packages, which means that the other 7 components are single packages. Actually, we are planning to launch these scripts on more computers so that we can compares the installations of different peers. This work is still on going.

3.2 Software package characteristics

In [5], a trace analysis has been taken from a large repository hosting packages and updates for approximately ten different Linux distributions and consisted of roughly 2.2 million files. The distribution of package size is shown in Figure 2. As we can see, 81 percent of these files are smaller than 256 KB and 91 percent of them are smaller than 1 MB. Thus software packages are generally very small in size. As we will discuss later in the related works, this property makes existing P2P file sharing protocols such as *BitTorrent* suboptimal.

The total number of software packages is enormous. For instance,the Debian GNU/Linux distribution comes with more than $25,000$ packages, occupying approximately 119,000 MB in size. However, users' interests on packages are very sparse. The Debian distribu-

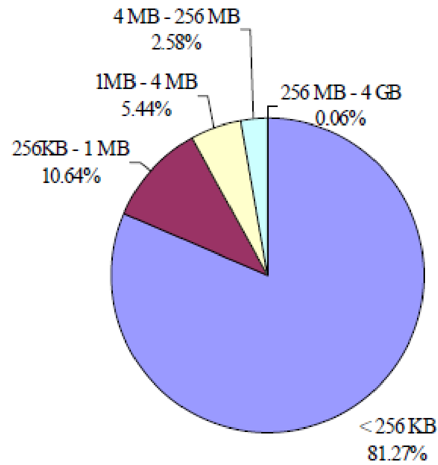


Figure 2: Package size distribution

tion has tracked the popularity of its packages. Statistics show that even though some packages are installed by everyone, 80% of the packages are installed by less than 1% of users [5].

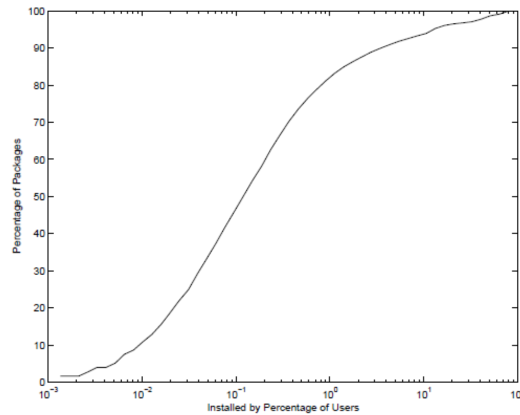


Figure 3: CDF of package popularity in Debian

The packages are constantly be updated. Though only a very small portion of it at a time is updated, these updates occur on a daily basis. In the Debian archive, every single day, approximately 1.5% of the 119,000 MBytes archive is updated with new versions [5]. During the internship, we have also collected a large quantity of package metadata from the Debian snapshot web site [4]. The metadata consist of 114308 different versions of 29266 packages, each package has in average 4 different versions. The distribution of

[4] `http://snapshot.debian.org/`

numbers of versions for each package is given below. More than 90% packages have less than 10 versions. It needs to be mentioned that the metadata we've collected is not continuous in time, which means that not all the historical versions have been collected.

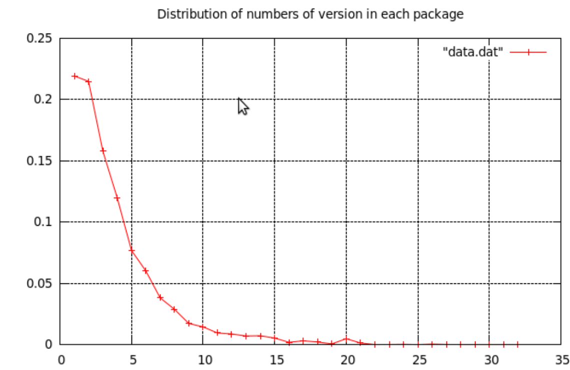


Figure 4: Distribution of numbers of version of Debian packages

Beside the number of versions per package, we are also interested in analyzing the differences between packages having same name but different versions. Here we define the difference between two package as the number of different dependency specifications in the metadata divided by the total number of dependency specifications in the metadata of the two packages. We have obtained the following distribution, that nearly 70% of the version pairs have only less than 15% different *Depends* relations, and more than 95% of the versions pairs have less than 10% different *Conflicts* and *Provides* relations. Thus we can conclude that packages with same name but different versions are in general very similar to each other in terms of metadata specification.

4 Related works

To the best of our knowledge, no previous studies have discussed repository-less package management systems. The aforementioned problems of current systems have however been highlighted in a few works, which have promoted peer-to-peer assisted systems, where every peer is both client (a device periodically maintaining the consistency of its operating system) and server (a repository able to supply clients with their downloaded software packages). We give a short description of these studies in the following, then we emphasize their limitations.

4.1 Overview

In [2], a peer-to-peer network is used for the delivery of Microsoft Windows software updates. Authors show in particular that, if peers stay in the system to serve as many bytes as they have received, then the benefit of the P2P system increases dramatically and the load at the server becomes almost negligible (nearly 10% of the load at a centralized

server). but they also detail some challenging problems, such as guaranteeing secure and timely package delivery, protecting user privacy, *etc.* However, this work is considered as a pioneer in this area, and it illustrates accurately the scalability requirement with interesting numbers from their real-world experience. Since this seminal publication, only two other approaches have been described.

The first proposal [6] is a straightforward implementation of the well-known *Bit-torrent* protocol for massive peer-to-peer package distribution. The idea behind using popular peer-to-peer file-sharing systems is promising: when multiple users are requiring the same package simultaneously, they upload data to each other, therefore reducing the workload on the central server. In other words, this proposal does address neither the problem of package certification (only certified packages have a Bit-torrent tracker), nor the problem of accessing the package (users know the Bit-torrent tracker of the required package). Authors focus on delivering massively a package, which may be useful when a patch should be fast diffused at large-scale.

The apt-p2p [7] project addresses the problem of accessing a package. In this approach, all software packages are distributed over participants, which are organized into a Distributed Hash Table (DHT) storing a list of *(key, string)* pairs. The keys are assigned using the cryptographic hashes of the packages and the string stores the location of all the peers possessing the package. When a user wants to download a certain package, it firstly searches for its hash in an indexing file, which is maintained in the server of the software provider. Then, the application uses the DHT to discover from the user's request some other users having the target package. After the package is retrieved, the demanding peer location is also added to the DHT, as it is now a source for others to download from. Note that the Bit-torrent package diffusion can easily be combined with this DHT approach. Actually, the former focuses on the delivery although the latter manages the discovery of the set of peers having the package. Note also that several works have been based on such a combination of existing peer-to-peer systems [5, 8].

4.2 Limitations

We identified at least three problems in the Bit-torrent approach. Firstly, the statistics in last section reveal that most of the packages of a software release are very small in size. Recall that the size of a unit piece in Bit-torrent is 512 KBytes. Hence, most packages can not fulfill a single piece. The downloading becomes inefficient as a large percentage of bandwidth is wasted on overhead. Secondly, the total set of packages is a very large repository, including many different versions and architectures. This increases the diversity of peer requests, and reduces the opportunities for sharing. Finally, the archive is too frequently updated. For Bit-torrent, package updates require the creation of new torrents and reduces the common interests on every single package.

The apt-p2p project exhibits also some drawbacks, which makes it quite useless. As a structured peer-to-peer system, it is heavy and costly to maintain in a dynamic environment, especially in pervasive environments where small laptops and smart-phones are expected to move, connect and disconnect frequently. Another drawback is that packages are randomly distributed over peers in the DHT (it is actually the principle

behind the DHT). However, packages are linked into a structured dependency graph, and communities of developers and users sharing similar packages should enable a more efficient data allocation and organization.

Finally, we would like to emphasize that these solutions address only a part of the problems related with packages management. Authors do not deal with the inefficiencies of the centralized certification process in the context of User-Generated Software and pervasive environment. In our opinion, and as we will see now, the problem should be treated entirely because all functions (discovering owners of packages, delivering the packages and certifying packages) are inter-dependent.

5 A repository-less package management system

In all the previous works, even though the workload of package delivery has been distributed across the network of peers, the repository still exists as a centralized administer. Our claim is that the centralized approach should be revised not only for the package delivery but also for the package certification. Actually, the amount of user generated software and the amount of devices to upgrade make the management of repositories too costly. We need an autonomous system, which handle the administration of machines in an autonomous and fully distributed way. We envision a repository-less package management system.

5.1 Distributed system model

The elimination of the central repository will lead to intrinsic changes in the system model. In a repository, packages being released are identified by a unique pair consisting of a package name and a version number. The version numbers are usually encoded into a sequence of integers by the repository manager so that there exists a total order on versions. However, in a repository-less system, there is no global administrator for revision control. Imagine that two developers modify the same version of a package concurrently, without knowing the existence of the others. If the package version is still identified by integers, then both developers may attribute the same version number to their new release. In this case, the concept of version as number does not make sense any more. As we still need to resolve version-specific package dependencies, another mechanism is needed to denote the heritage relations between packages. One possible solution is that every package keeps track of its own history in its metadata. The version comparison between two packages is realized by looking for one package in the history of the other. Due to the lack of a repository, the system model of section 2 must be modified and follow the new metamodel for repository-less package deployment depicted in Figure 5.

Note that the metamodel in Figure 1 is for a universal repository, while Figure 5 models a distributed system consisting of *peers*. Each peer assume part of the central repository functionality by storing a certain number of packages. Only a subset of these stored packages are actually installed. The notion of *version* is replaced by the ances-

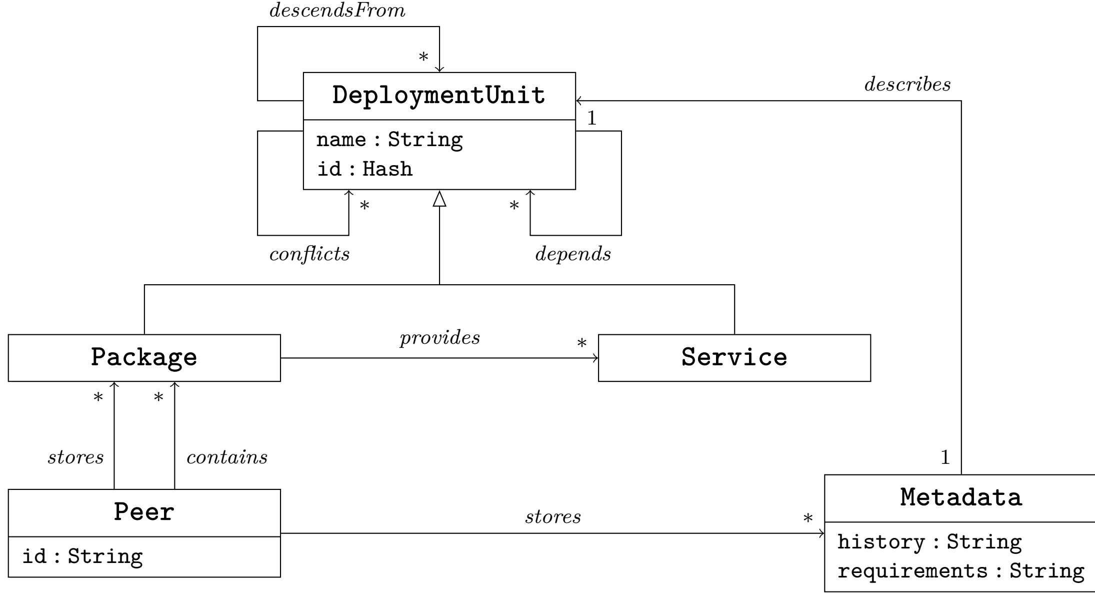


Figure 5: Package dependency graph metamodel

tor relationship between deployment units. A deployment unit must still be uniquely identified by an *id* (for instance, a sufficiently long hash of the data contained in the unit could be used). The attribute *name* is still needed to identify families of deployment units. However, the name of a unit may differ from the name of its ancestors as a developer can create a new component by merging two or more existing components of different families. In this case, we impose that the name of the new package to be the concatenation of its parents name, with a special character inserted between the two parts.

In the metadata, a new feature is added to record the entire revision *history* of a unit. The association *descends* is used to denote the direct heritage relation between two units. Due to these changes, the formal definition of the system model must also be adjusted. The *depends* and *conflicts* functions can no longer be defined using the repository. But only can we define them from the metadata requirements that are available to the peer. A new formalization is given below.

Definition 5. *A* ***deployment unit*** *is a pair $u = (n, id)$ where n is the unit name and id is the unique identifier of the unit.*

Definition 6. *The* ***metadata*** *m of a deployment unit u is a pair (R, h) where R is the set of requirements of u and h is its revision history*[5]*.*

- *A revision history $h = (U_1, U_2..., U_n)$ is a sequence of elements U_i that are sets of*

[5]We write $m.u$ and $u.m$ as there is a one to one association between m and u, we further write $u.R$ and $u.h$ to express the requirements and the revision history of u.

deployment units. In h, the units contained in U_i are the direct ancestors of the units contained in U_{i+1}.

- *For any two deployment units u_1, u_2, $u_1 < u_2$, if and only if $u_1 \in u_2.h$.*
- *A requirement is a predicate $r : U \times \{depends, conflicts\} \times \Sigma \to \{true, false\}$ where U is a set of deployment units and $\Sigma = \{name\ is\ n, = u, > u, < u, \leq u, \geq u\}$ is the set of version constraint specifications.*

  1. $r(U, depends, name\ is\ n) = \exists u \in U, u.n = n$
  2. $r(U, conflicts, name\ is\ n) = \forall u \in U, u.n \neq n$
  3. $r(U, depends, = u) = u \in U$
  4. $r(U, conflicts, = u) = u \notin U$
  5. $r(U, depends, < u) = \{u' \in U \mid u' < u\} \neq \emptyset$[6]
  6. $r(U, conflicts, < u) = \{u' \in U \mid u' < u\} = \emptyset$

Definition 7. *A peer is a tuple $P = (U_s, U_i, M)$ where U_s is the set of deployment units stored on the peer and U_i is the set of units installed on the peer. U_i is also called the* installation *of P, we have $U_i \subseteq U_s$. M is the set of metadata stored on the peer ($\forall u \in U_s, u.m \in M$).*

In the new model, a peer plays the role as both a server and a client. The global repository (server) now is replaced by many distributed U_s and U_i is equivalent to the concept of *Installation* (client) in the model of Section 2. there is no requirement on the storage part U_s, allowing a peer to cache any package encountered on the package management network. However, the installation part U_i must still be healthy. In M, we may also store metadata of packages that are not in U_s.

Definition 8. *A set of deployment units U is* ***healthy*** *if and only if the requirements of all its elements are satisfied within U: $\bigwedge_{u \in U} u.R$. A peer is* ***healthy*** *if and only if its* installation *U_i is a healthy unit set.*

The three operations defined in Section 2 can now be defined the same way substituting the global repository by the peer.

Definition 9. *Assuming a peer $P = (U_s, U_i, M)$, three operations are defined on packages, installation, upgrade and removal:*

1. *A package π is* ***installable*** *in P if and only if there exists a* healthy installation *U_i' such that $U_i \cup \{u\} \subset U_i' \subset U_s$.*
2. *A package π of P can be* ***upgraded*** *if and only if there exists a package π' in U_s such that $\pi' > \pi$ and π' is installable on $U_i \setminus \{\pi\}$.*
3. *A package π of P can be* ***removed*** *if and only if $U_i \setminus \{\pi\}$ is a healthy installation.*

[6] The cases for $> u'$, $\geq u'$, $\leq u'$ are omitted for simplicity as they are similar.

5.2 System functionalities

Current package managers such as the *Advanced Packaging Tool* (APT) and the *RedHat Package Manager* (RPM) are designed only to provide facilities for software consumers. The operations of producing, certificating and publishing a package are handled on the repository side by the software distributor. In our system, the role of repository is distributed across the network. Peers are both software consumers and software producers. Thus, the functionalities offered by our system must be twofold:

- On the software consumer side:
  - *Install* or *update* one specific deployment unit: the user queries the deployment unit using its name optionally with version constraint, and fetches all the necessary metadata from the network, after resolving the dependencies, it downloads all the missing deployment units, configures and installs them on its local. If no version constraint is specified, the peer will default to choose the latest version according to its knowledge. If a deployment unit u with the required name is already install on the peer, then it is an *update* operation, the version constraint will be implicitly set to $u' > u$. The problem here is: how can a peer know what is the latest version of a deployment unit without a global knowledge of all the available units in the network? As we will discuss later, a metadata dissemination mechanism is required to guarantee that peers are timely informed with the latest features.
  - Upgrade the whole system: the peer checks its installation, updates all the installed deployment units that have a later version than the currently installed ones. In another word, this is done by executing the *update* operation for all the installed deployment unit. Note that some updates may be in conflict with some others, and multiple solutions may be possible while updating the whole system. Thus the *update* operations of individual deployment units must be done collaboratively in order to maintain the consistency of the installation. Some predefined strategy is needed while deciding among conflicting updates. For example, core security patches may be given higher priority and deployment units may be ranked according to the frequency of utilization.
  - Synchronize with a specified peer: the user selected a specific peer to follow. And synchronized periodically its installation with that peer. This functionality is not supported by current major package managers. However, in a fully decentralized network, it is useful for configuring newly joined peers. It provides also an autonomous approach for maintaining heterogeneous devices in the internet of things.
- On the software publisher side:
  - Certificating a new deployment unit: test the safety and functionality of a new deployment unit before it is made available in the network. This is crucial as software packages are more security-sensitive than other forms of

user-generated content. However, in this paper, we currently consider as out of scope all the processes happening before a deployment unit is published, and focus only on the searching and notification issues. But it is important to mention that the certification of user generated software must be tackled by our system. A new certification mechanism must be proposed in our future work. In section 7, we will give a short discussion on software certification.

– Notifying other peers with newly available units: after a deployment unit is issued, the producer must disseminate the metadata of the unit across the network so that other peers can discover and download it.

6 Application level protocols

As previously mentioned, our system must tackle the problems of efficient content searching (pull) and metadata dissemination (push). In this section, we study candidate approaches and propose a content/interest-proximity based overlay structure dedicated to our application.

First, a peer has to constantly and opportunistically notify other peers about its new deployment units. The metadata of the new units should be advertised to users in an efficient and timely way. Note that the total quantity of metadata is enormous, flooding all the metadata on the entire network is clearly not scalable. In practice, only peers that are potentially interested in the new deployment unit should be notified. This requirement corresponds exactly to the abstraction offered by the publish/subscirbe primitive. However in most publish/subscribe systems, users must explicitly express their interest. An application specific subscription language is required. To our knowledge, no previous works have addressed the classification of software packages in fine granularity, thus designing an expressive and scalable subscription language for package management is a real challenge. Besides, in the Internet of Things, the administration of machines is often autonomous, making it complicated to configure the subscriptions for different devices. A more detailed description of publish/subscribe systems will be given later in this section. Another candidate approach is P2P *gossip-based* data dissemination: every peer is connected to a set of other peers, called its *neighbors*, and periodically exchanges with them information. A peer can decide to change its neighbors at any time, so the system is flexible and adaptive to network changes. Gossip-based systems have been proved to be efficient for distributed information sharing applications [9]. In this approach, we can not deterministically choose the destinations of the notifications. However, we can guarantee that peers with related interests on the deployment unit will be notified with higher probability. This may be achieved by grouping peers into a clustered semantic overlay according to their content proximity, as peers having similar installations are more likely to share common interest in selecting new packages. While issuing a new deployment unit, the owner of the unit first notifies peers that are semantically close to him, the receivers then forward the message to other peers.

Second, a peer that has been able to fetch the metadata of a new deployment unit should discover by itself all the missing deployment units before actually installing it. In

this section, we also look toward a fully decentralized approach for package searching. We inspire from the recently proposed *information centric network* (*e.g.* [10]) and the well known Freenet [11] file sharing system. The idea is that the requester does not reference any specific physical peer hosting the requested deployment unit, but network elements collaborate in order to determine the closest peers that actually store this deployment unit. In such networks, every peer is typically invited to store the deployment units that have been recently requested (from what this peer has observed). Therefore, the more requested is a deployment unit, the easier it is to find.

6.1 Publish/Subscribe-based metadata dissemination

The general service offered by a publish/subscribe system is the delivery of information from publishers to interested subscribers in a loosely-coupled fashion. In such a system, publishers produce information (or *notifications*), which is consumed by subscribers. The most important property of pub/sub paradigm is that: subscribers are not directly targeted from publishers, but rather they are indirectly addressed according to the content of notifications. That is, a subscriber expresses its interest by issuing subscriptions for specific notifications, independently from the publishers that produce them, and then it is asynchronously notified for all notifications that match its subscriptions. This propagation mechanism is realized by introducing a logical intermediary between publishers and subscribers, the *Notification Service*. Both publishers and subscribers communicate only with the Notification Service, which (*i*) stores all the subscriptions, (*ii*) receives all the notifications from publishers, (*iii*) dispatches all the notifications to the correct subscribers. The result is that publishers and subscribers exchange information without directly knowing each other. This anonymity is one of the main features of the pub/sub paradigm and simply stems from the level of indirection provided by the Notification Service [12].

6.1.1 General model of Publish/Subscribe systems

In [12], a generic model of publish/subscribe systems is given as follow: the system is represented by a triple (Π, Δ, Σ) of sets of processes. Sets in the triple are defined depending on the role of the processes in the system: $\Pi = p_1, ..., p_n$ is a set of n processes, called the publishers, which are producers of information. $\Sigma = s_1, ..., s_m$ is a set of m processes called the subscribers, which are consumers of information. Σ and Π may have a non-zero intersection, that is a same process may act both as a publisher and as a subscriber. $\Delta = B_1, ..., B_o$ is a set of o processes, called brokers.

publishers and subscribers are decoupled: a process in Π can not communicate directly with a process in Σ and vice versa. They can only communicate with any other process in Δ. If the number of brokers $|\Delta| = 1$, then we have a centralized implementation for the Notification Service. Otherwise, if $|\Delta| > 1$, the Notification Service is implemented as a network of distributed brokers.

As shown in Figure 6, publisher submits a piece of information e to other processes by executing the *publish(e)* operation on the Notification Service. The Notification Service

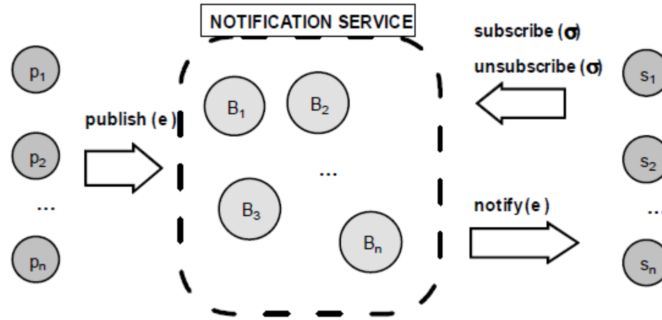


Figure 6: Generic model of Publish/Subscirbe system

dispatches a piece of information e submitted by other processes to a subscriber by executing the *notify(e)* on it. A subscription σ is respectively installed and removed on the Notification Service by subscriber processes by executing the *subscribe(σ)* and *unsubscribe(σ)* operations. A notification e is often represented by a set of attribute-value pairs. Each attribute has a name, a simple character string, and a type. The type is generally one of the common primitive data types defined in programming languages or query languages (*e.g.* integer, real, string, *etc.*). A subscription is a pair $\sigma = (f, s)$, where $s \in \Sigma$ is the subscriber which is interested in notifications declared through the filter f. We say a notification e matches a subscription σ if it satisfies its filter $\sigma.f$. The task of verifying whenever a notification e matches a filter f is called matching ($e \sqsubset f$).

6.1.2 Subscription model

There are basically two types of subscription models that are of our interest: topic-based model and content-based model. The two models mainly differ in the expressiveness of the subscription language and in the complicity of implementation. The more expressive and flexible a Notification Service is, the more difficult it is to implement and to scale up.

Topic-based model: in the topic-based model, notifications are grouped in topics (or subjects) which represent many-to-many distinct (and static) logical channels. Participants can publish events and subscribe to individual topics, which are identified by *keywords*. The notion of topic is similar to that of *groups* in group communication. Subscribing to a topic T is equivalent to becoming a member of a group T, and publishing an event on topic T corresponds to broadcasting that event among the members of T [13]. The main drawback of the topic-based model is the limited expressiveness it offers to subscribers, as it statically divides the interest space into discrete groups represented by string keys. Many improvements of the topic-based publish/subscribe model have been proposed. Most of them use hierarchical addressing to classify topics,

which allows programmers to organize topics according to containment relationships. A subscription made to some node in the hierarchy implicitly involves subscriptions to all the subtopics of that node. However, for our application, software packages can hardly be classified into hierarchical structures due to their massiveness and variability. Thus, the topic-based subscription model is not powerful enough to express users' interest on particular packages.

Content-based model: in the content-based model, subscribers express their interest by specifying constraints on the notifications they want to receive. The filter in a subscription is represented by a conjunction of constraints over different attributes of the notification. Most subscription syntax support equality and comparison operators as well as regular expressions. Generally constraints can be joined inside filters through AND/OR expressions. Two examples are shown below:

`PackageName = 'python' and Version` $>$ `'1.5'`

`ServiceName = 'packageManager' and Version` $\geq$ `'2.0'` [7].

In content-based publish/subscribe, notifications are not classified into clusters according to any external criteria, (*i.e.*, topic name), but rather are distinguished according to the content of the notifications themselves. The set of destinations for each notification cannot be determined a priori but has to be computed at publication time. Thus, the higher expressiveness of content-based subscription is obtained at the price of more overhead in calculating the set of interested subscribers for each notification. Besides, in content-based publish/subscribe systems, it is not always trivial for users to exactly express their interest. Because the metadata of software packages is opaque, the subscription syntax is often very different from nature languages. Thus, specifying subscription queries that return only relevant items is challenging. In [14], the process of managing complex subscriptions is automated according to the *user attention* on data. The authors record users' Web browsing histories, extract the most common keywords from all the visited pages, and recommend subscriptions queries out of them. Then users' feedbacks are collected to help determine wether a subscription should be kept or removed. In our package management system, we may also imagine a similar approach, that is, let the package manager to automatically recommend subscriptions according to the peer's current installation. We simply assume that a peer is interested in the update information of all the deployment units that are already installed on it. The main challenge in designing such a system is to ensure the scalability, as the total numbers of subscriptions and notifications are huge, and the calculation for filter matching is based on a per-notification basis, the routing of notifications to the corresponding brokers and subscribers consumes lot of network resource. Due to these reasons, a distributed Notification Service architecture becomes a nature choice.

[7]In our new system model, the version should no more in the form of a number but a unique string identifier

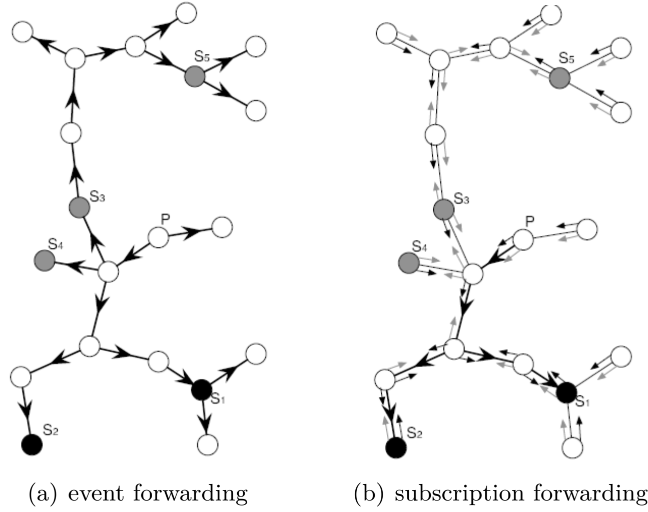
6.1.3 Distributed Notification Service

A distributed Notification Services often consists of a network of interconnected brokers. Peers can play the role as: publishers, subscribers or brokers. Each broker is attached to a number of clients (publisher/subscribers) and is responsible for storing a number of subscriptions. If a subscription σ is stored at a broker B, then B is referred to as the target broker of σ. When an event e is published, it should be routed to the target brokers where the matching subscriptions are stored. In most implementations, the brokers are organized into an acyclic topology to aid the routing process. The most common routing strategies are subscription forwarding based and event forwarding based (or some sort of hybrid combination of the two) [15].

The idea of event forwarding is just to flood each event to all the brokers. The information about a client's subscriptions, on the other hand, is never communicated but is stored local to the broker the client is attached to. This information, stored in the broker's subscription table, is checked whenever an event is received, to determine whether any of the attached clients should receive a copy of the event.

Clearly, event forwarding is not scalable as all events are sent to all brokers, regardless of the presence of matching subscriptions. Subscription forwarding improves the efficiency of event routing by spreading knowledge about subscriptions across the network of brokers. Specifically, when a client issues a subscription for a given event pattern, not only is the pattern inserted into the subscription table of the broker the client is attached to, but the subscription message is also forwarded to all the neighboring brokers. This process effectively builds up a routing table at each broker. A received event is only forwarded to those brokers who have registered matching subscription patterns. However, the forwarding and replication of subscriptions also introduce important overhead, especially when the number of subscription is big and when the subscriptions are constantly updating. Actually, the relation between the two routing strategies is a tradeoff between event flooding and subscription flooding. In figure 7, a event is published by broker P, the black circles S_1, S_2 are two target brokers, the greys arrows indicate the flow of subscriptions and the black arrows indicate the flow of events.

In recent years, many works such as [16] and [15] have proposed efficient content-based routing protocols which significantly reduce the redundancy in diffusing subscriptions and events. The general idea is to model the subscription patterns as n-dimensional spaces referred to as *zone of interest*. In the routing table of each broker, each entry denotes the zone of interest that can be reached through one outgoing link. The diffusion of subscriptions is alleviated by considering the covering relation between zones of interest. If a new subscription does not change the zone of interest of a broker, then it wont be broadcasted to other brokers. Another idea is to construct a self-organized overlay topology which provides facilities for the routing process. For example, brokers having similar subscriptions may be directly connected together in the overlay to form clusters, so that during the propagation, an event is expected to follow a single path toward the cluster rather than being diffused in different directions towards dispersed target brokers. In this way, the number of pure forwarding brokers (brokers that receives the event but do not have any matching subscription) may be reduced.



(a) event forwarding (b) subscription forwarding

Figure 7: Publish/Subscirbe routing strategies

6.2 Gossip-based metadata dissemination

Gossiping in distributed systems refers to the repeated probabilistic exchange of information between peers. In such a system, each peer possesses a partial view of the participants in the network, referred as its neighbors. In principle, gossiping is the endless process of a peer randomly choosing a member from its neighbors and subsequently exchange information with it. The effect of gossiping is that information can spread within a group just as it would in real life. It is strongly similar to epidemics, by which a disease is spread by infecting members of a group, which in turn can infect others [9].

The general framework of gossip-based protocol is shown as follow. Two separate threads are involved: an active thread which initiate the communication, and a passive thread listening to incoming requests.

```
1 Q = selectPeer();
2 bufferSend = selectDataToSend();
3 sendTo(bufferSend, Q);
4 bufferReceive = receiveFrom(Q);
5 cache = selectDataToKeep(bufferReceive);
6 processData(cache);
```

Algorithm 1: Active thread (peer P)

Consider two peers, P and Q, both equipped with a cache, consisting of references to other peers in the system. A cache is also capable of storing application data. Peer P first selects peer Q as target peer to communicate. It then decides on what data that it will send to Q, and stores this in a send buffer. The information is then sent to Q, which, in turn, selects information to return to P. After the exchange has completed,

```
1 (bufferReceive, P) = receiveFromAny();
2 bufferSend = selectDataToSend();
3 sendTo(bufferSend, P);
4 cache = selectDataToKeep(bufferReceive);
5 processData(cache);
```

Algorithm 2: Passive thread (peer Q)

both decide separately which exchanged information will actually be stored in the cache. In case the cache is limited in size, this may imply that cache entries will need to be replaced. The implementations of the functions `selectPeer(), selectDataToSend() and selectDataToKeep()` are application specific. In our case, we aim to use gossip for package metadata dissemination. Thus the data need to send is just information (metadata) of newly issued deployment units together with the sender's id and a timestamp. A peer can simply keep all the metadata it receives, as long as there is enough space in its cache, but replications of same data on the same peer should be avoid. When the cache memory is full, we may choose to discard the oldest or least recent used (LRU) metadata. The most crucial function here is the `selectPeer()` function, which should probabilistically ensure that most peers with related interests can receive the data timely. The solution that we propose is to build a semantic overlay network which cluster peers with similar interest together. This is based on the assumption that a peer issuing a deployment unit is in most cases itself interested in that deployment unit. Such an approach is very popular in current P2P systems, both for data searching and dissemination.

6.2.1 Construction of a content/interest-proximity based overlay network

An overlay network is a virtual network consisting of logically connected nodes built on top of the existing physical network. It provides facilities to implement network services that are not available in the physical network, such as membership management, message routing and data location. Recent research on content delivery has focused on constructing semantic overlays to facilitate content searching and data dissemination. In a semantic overlay network, peers are connected to a set of semantically related neighbors. While pulling or pushing content, a peer first sent the interest or the data to its neighbors.

In [17], the construction of a content-proximity based semantic overlay has been thoroughly addressed. The author outlined an epidemic-style algorithm which converges the overlay topology to obtain highly desired properties. In their system model, each peer maintains a dynamic list of semantic neighbors, called its semantic view, of fixed small size l. Similar peers are clustered together using a proximity function $d(p_1, p_2)$ which measures the distance between the content of two peers $p1, p2$. The algorithm aims at filling a peer's semantic view with its l semantically closest peers out of the whole network, that is, to pick peers $Q_1, Q_2, ..., Q_l$ for peer P's semantic view, such that

the sum $\sum_{i=1}^{l} d(P, Q_i)$ is minimized. The construction of such semantic views consists of two sides.

First, the algorithm explores the transitivity of the semantic proximity function $d(p_1, p_2)$). That is, if P and Q are semantically similar to each other, and so are Q and R, then some similarity between P and R is likely to hold. Thus exploiting the transitivity in d should quickly lead to highly clustered overlay topology.

Second, it is important that all nodes in the network are examined. If only the transitivity is followed while selecting neighbors for a peer, then we eventually will be trapped only in a single semantic cluster. Similar to the special long links in small-world networks. Thus it is necessary to establish links to other semantically-related clusters. The resulting overlay topology must exhibit some sort of randomness.

In their design, a gossip based two-layered protocol is proposed. The lower layer, referred as CYCLON [18], is responsible for maintaining a connected overlay and for periodically feeding the top-layer protocol with nodes uniform randomly selected from the network. While the top-layer protocol, called VICINITY, is responsible for selecting peers that are semantically as close as possible. Figure 8 illustrates the two layer structure of their protocol.

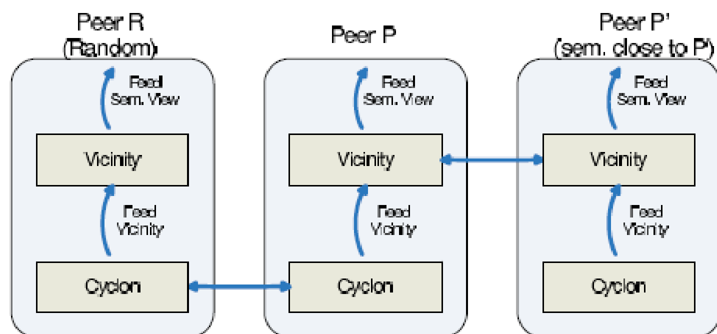


Figure 8: Two layer structure for overlay topology construction

Both CYCLON and VICINITY follows the general gossiping protocol skeleton described in algorithm 1 and 2, the item gossiped by a peer P contain the following three fields:

1. P's identifier that allows any other peer to contact P
2. A timestamp recording the creation time of the message
3. A descriptor of P that summarizes P's interest or content

In the cache of each peers, l_v items are stored as a partial view for VICINITY and l_c items are stored as a partial view for CYCLON. The three core functions `selectPeer()`, `selectDataToSend() and selectDataToKeep()` are defined as follow.

function(CYCLON)	Description
selectPeer()	Select peer from the item with the oldest timestamp
selectDataToSend()	Randomly select g_c items from the CYCLON view
selectDataToKeep()	Keep all g_c received items, replacing (if needed)the g_c ones selected to send. In case of multiple items from the same node, keep the one with the most recent timestamp.

Table 2: The CYCLON protocol

function(CYCLON)	Description
selectPeer()	Select peer from the item with the oldest timestamp
selectDataToSend()	Select the g_v items of nodes semantically closest to the selected peer from the VICINITY view and the CYCLON view
selectDataToKeep()	Keep the c_v items of nodes that are semantically closest, out of items in its current view, items received, and items in the local CYCLON view. In case of multiple items from the same node, keep the one with the most recent timestamp.

Table 3: The VICINITY protocol

Such a two layer framework offers exactly the service required by our application. The only remaining task here is to define a proximity function that gives reasonable measures of the interest similarity between different peers.

6.2.2 Proximity function

The proximity function $d(p_1, p_2)$ provides a metric that quantifies the distance between two peers in terms of content or interest. p_1, p_2 are the descriptors of the two peers being measured. Concerning the package management application, we assume that peers that have installed similar packages are like-minded, and tend to be sharing lots of common interests in choosing new packages. Thus the measure of interest proximity can be reflected by the measure of content proximity. The peer descriptors may be chosen as the package dependency graphs or simply the lists of installed (stored) packages. If the package dependency graph is used, then the problem is translated to defining the distance metrics between two labeled directed graphs, which is a widely studied subject. There are many solutions based on exploiting the maximum common subgraph or subgraph isomorphism [19]. The complexity of resolving such problems is NP-complete. Some other methods based on graph kernels [20] are more efficient to compute, however, still require implementations of sophisticate algorithms. In addition, the entire dependency graph consists of an important quantity of data that is to costly to be exchanged frequently among peers. A more compressed data structure is desired for summarizing a

peer's content. At last, the inter-package relationships (edges) are pre-determined in the package metadata, hence the topology of the dependency graph is basically determined by the set of nodes. Due these reasons, it is a good choice to use the list of all the packages on a peer instead of the package dependency graph as the peer's descriptor. The statistics in section 3 have revealed that multiple versions may exist for one package, which reduce the possibility of two peers sharing the exact same package (same name and same version). We have also observed that generally, different versions of a same package have very similar dependencies. Thus while comparing the content of two peers, we consider packages with same name but different versions as totally equivalent. This improves the rate of coincidence and simplifies the comparison process. We formally define, in our application, the distance between to peers as $d(p_1, p_2) = 1 - \frac{|p_1 \cap p_2|}{max(|p_1|,|p_2|)}$, where p_1,p_2 are the two lists of names of the installed (stored) deployment units.

6.2.3 Bloom filter

In order to achieve efficient package delivery, we look forward to organize peers into an overlay network based on their content similarity. This approach requires a compressed data structure that can summarize peer's content, *i.e.* all the packages stored on the peer. Ideally, the entire package dependency graph can be used, however it costs too much bandwidth to transfer this data and the comparison between two directed labeled graphs needs complicated algorithms. Concerning our case, we assume that a package dependency graph is mostly determined by its collection of packages (vertices), because intuitively, similar packages tend to have similar dependencies (edges). The on going statistic on Debian repositories is just aiming to examine this trend. Thus we propose to use the list of package names as the descriptor of a peer's content. Bloom filter is a widely used technique for representing large sets of contents and supports membership queries.

A Bloom filter for representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements is described by a vectors 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) to a integer over the range $\{1, \ldots, m\}$. And the bits $h_i(x)$ is set to 1, $1 \leq i \leq k$. A location can be set to 1 multiple times, but only the first change has an effect. To check whether an element y is in S, we need to check wether all the bits $h_i(y)$ is set to 1, $1 \leq i \leq k$. If not, then clearly y is not a member of S. If all $h_i(y)$ are set to 1, we assume that y is in S, although we are wrong with some probability. Hence, a Bloom filter may yield a false positive, where it suggests that an element x is in S even though it is not. Figure 1 provides an example. The initial Bloom filter is a vector consisting of only 0. Two elements x_1, x_2 are encoded by three hash functions. The resulting bit locations are set to 1. And two elements y_1, y_2 are checked against the filter. y_1 is obviously not contained in the set, but y_2 appears to be. However, this may be wrong with a small probability of false positive, which is given by: $f = (1 - e^{-kn/m})^k$. From the equation, we can deduce that for a given number of hash function k, the longer the filter length m, the lower the false positive rate f is. However for a given m, the variation of k has a tradeoff effect. A bigger k will change more bits to 1, however will also increase the probability of 0 bit during membership

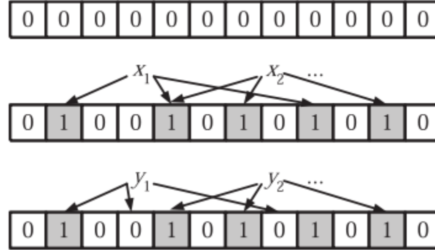


Figure 9: Example of Bloom filter

checking. It has been proved [21] that when $k = m \ln 2/n$, the false positive rate reaches its minimum at $P_{min} = (\frac{1}{2})^k = 0.6185^{\frac{m}{n}}$. In real implementations, the values k and m must be well chosen to obtain an acceptable false positive rate without adding to much overhead and side effects.

In our application, the content similarity between two peers can be measured using the Hamming distance (number of different bits) between the two Bloom filters representing each of them. Only package names are hashed and are inserted into the Bloom filter of the current peer, the version constraints are not taken into account. It means that we consider packages with identical name but different versions as equivalent in terms of content similarity. The notion of *name* here requires us to revisit our system model defined in the previous report. In the former definition. A package's name can differ from its ancestors, because one may create a new package by merging two existing ones. Thus the new package is a common child of two different families and can have a totally different name. However in this case, after being hashed, we can no longer find any equivalence between the new package and its ancestors. One solution to solve this problem is to name the new package using the concatenation of its parents' names. While performing the hash function, the two parts of its name (extended from different parents) are hashed separately, and are inserted into the filter as two individual packages. While checking the exsitence of such a package, both parts of its name should be queried against the filter.

As machines on the Internet are constantly updating their system, their contents keep changing over time. Their filters must also be updated frequently. For a standard Bloom filter, it is simple to add new element; hash the new element and change the bits to 1. However, the deletion of element is not supported; we can not change all the corresponding 1 bits to 0 because these bits may also be set by some other elements. To tackle this problem, we can use an advanced version of Bloom filter, the *counting Bloom filter*, which uses a vector of counters instead of a vector of bits. When a new item is inserted, the corresponding counters are incremented by 1; when an item is deleted, the corresponding counters are decremented by 1. Existing analysis [21] has shown that normally 4 bits per counter should suffice for most applications.

To avoid complicated hash function design issues, we use constant number of hash functions k for all peers. One of the design goal here is to ensure an upper bound

of the false positive rate f, for instance, $f \leq 0.01$. Assuming the values of k and m/n are optimally chosen to reach the best tradeoff, we should have $0.6185^{m/n} \leq 0.01$, thus $m/n \geq 9.58$, and $k = m \ln 2/n \approx 6.64$. In practice, the value of k must be an integer, thus we choose a suboptimal value $k = 6$ for the purpose of reducing the cost of hash computation. And for simplicity, we enlarge the number of bits per element to $m/n = 10$. Using the equation $f = (1 - e^{-kn/m})^k$, we have $f \approx 0.008 \leq 0.01$. As most implementations of Bloom filter, the MD5 encryption algorithm can be easily adapted to generate our 6 independent hash functions. One possible solution is to remove the last 8 bits of the standard 128 bits output of MD5 algorithm, and divide the remaining 120 bits into 6 sections. Each section is used as an independent hash function that compresses the element into a 20 bits representation. The maximum filter length can be $m = 2^{20} = 1,048,576 \approx 1,000,000$ bits, thus the maximum number of elements in a set can reach $n \approx 100,000$. In practice, the number of packages stored on a machine can hardly be this huge. Generally, a personal computer running Debian/Linux system may consist of only several thousands of packages. Besides, the storage capacity of different machines may vary a lot depending on their hardware types. It is unnecessary and inefficient to represent all peers by a 1 Mbits filter. However, the filter length m should be a constant, or at least belongs to a finite set of discrete possible values, so that we can compare the similarity of different filters. The design principle of Bloom filter in P2P applications has been thoroughly addressed in [22].

6.3 Content-based package searching

So far, we have discussed Publish/Subscribe systems and Gossiping-based systems, both are good candidates for disseminating notifications of newly created packages (push). However, like all the current package managers, our system should also address the content searching problem (pull). In the related works, we have revealed the main flaws of Bittorrent and DHT based approaches. Here, we are aiming to propose a fully decentralized system that operates as a network of identical peers. The storage and computation workload should be fairly distributed across the network edges, and peers should cooperate to efficiently route requests to the source location of data. Content retrieving is one of most studied application domain of P2P systems during the last years. In this internship, we can hardly attempt to invent any clean-slate solution, but rather revisit existing works that may be eventually adapted to achieve our goal. In this subsection, we mainly focus on the Freenet protocol and the Content Centric Network (CCN).

6.3.1 Freenet file-sharing network

Freenet is a peer-to-peer network implemented for storing and retrieving data files. Each node maintains a local data store and a dynamic routing table. Files in the data store are named using location-independent keys and are shared to all the participants in the network. In the routing table, each entry contains an addresses of another node and the keys of files it is holding.

While searching for a file, queries are passed along from node to node in a chain of proxy requests with each node making a local routing decision in the style of IP routing about where to send the query next. Nodes know only their immediate upstream and downstream neighbors in the chain. Each request is given a *Hops-To-Live* (HTL) limit, similar to the IP *Time-To-Live* (TTL) which is decremented at each peer to prevent infinite chains. Each request is also assigned a pseudo-unique random identifier, so that peers can avoid loops by rejecting requests they have seen before. If this happens, the preceding peer chooses a different peer to forward to. This process continues until the request either is satisfied or has exceeded its HTL limit. The success or failure signal (message) is returned back up the chain to the sending peer.

In addition, Freenet uses its data store to increase system performance. When an object is returned (forwarded) after a successful retrieval (insertion), the peer caches the object in its data store, and passes the object to the upstream (downstream) requester which then creates a new entry in its routing table associating the object source with the requested key. So, when a new object arrives from either a new insert or a successful request, this would cause the data store to exceed the designated size and *Least Recently Used* (LRU) objects are ejected in order until there is space. LRU policy is also applied to the routing table entries when the table is full [23].

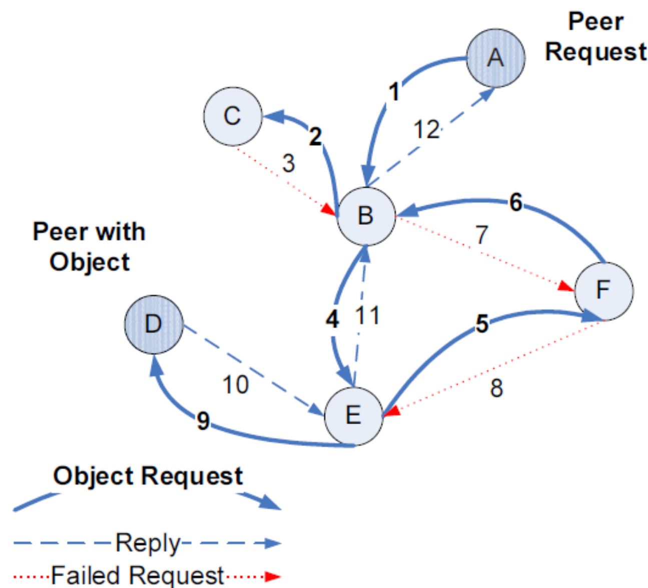


Figure 10: Request routing in Freenet

The figure above depicts a typical sequence of request messages. The user initiates a data request at peer A, which forwards the request to peer B, and then forwards it to peer C. Peer C is unable to contact any other peer and returns a backtracking failed request message to peer B. Peer B tries its second choice, peer E, which forwards the request to peer F, which then delivers it to peer B. Peer B detects the loop and

returns a backtracking failure message. Peer F is unable to contact any other peer and backtracks one step further back to peer E. Peer E forwards the request to its second choice, peer D, which has the data. The data is returned from peer D, via peers E, B and A. The data is cached in peers E, B and A, therefore, it creates a routing short-cut for the next similar queries.

The good property of Freenet is that no flooding search or centralized location index is employed. The network is resilient to peer failures, because of its lack of centralized structure. Files are dynamically replicated in locations near requestors and deleted from locations where there is no interest. The main limit of Freenet is that the storage of the routing table requires important disk space on each peer. The tiny devices in the Internet of things may not always have such storage capacity.

6.3.2 Content Centric Network

Content-Centric Networking (CCN) is a communications architecture built on *named data*. CCN has no notion of host at its lowest level: a packet "address" names content, not location.

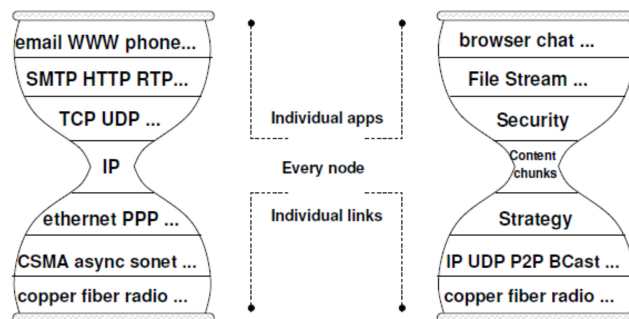


Figure 11: The CCN protocol stack

The CCN protocol stack is shown in Figure 11, CCN's network layer is similar to IP's and makes fewer demands on layer 2, giving it many of the same attractive properties. Additionally, CCN can be layered over anything, including IP itself. There are two types of CCN packets, *Interest* and *Data* (see Figure 12). A consumer asks for content by broadcasting its interest over all available connectivity. Any node hearing the interest and having data that satisfies it can respond with a Data packet. Data is transmitted only in response to an Interest and consumes that Interest. Data 'satisfies' an Interest if the Content Name in the *Interest* packet is a prefix of the Content Name in the *Data packet*.

The basic operation of a CCN node is very similar to an IP node: A packet arrives on a face, a longest-match look-up is done on its name, and then an action is performed based on the result of that lookup. Figure 13 illustrates the main data structures in a CCN router. There are three main components: the FIB (Forwarding Information

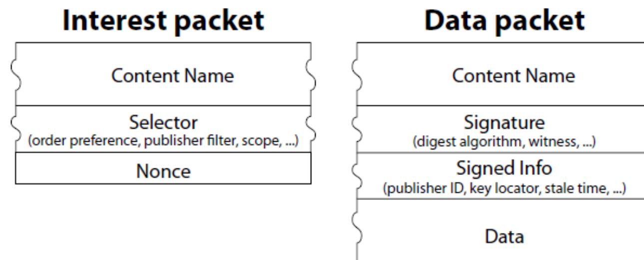


Figure 12: The CCN data and interest packets

Base), Content Store (buffer memory) and PIT (Pending Interest Table). The FIB is used to forward Interest packets toward potential source(s) of matching Data. It is almost identical to an IP FIB except it allows for a list of outgoing faces rather than a single one. The Content Store is the same as the buffer memory of an IP router but has a different replacement policy. To maximize the probability of sharing, which minimizes upstream bandwidth demand and downstream latency, CCN remembers arriving Data packets as long as possible (LRU or LFU replacement). The PIT keeps track of Interests forwarded upstream toward content source so that returned Data can be sent downstream to its requester.

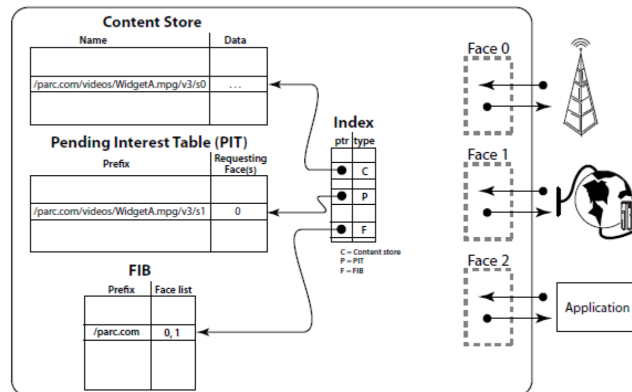


Figure 13: The CCN node architecture

When an Interest packet arrives on some face, a longest-match lookup is done on its Content Name. The index structure used for lookup is ordered so that a Content Store match will be preferred over a PIT match which will be preferred over a FIB match. Thus if there is already a Data packet in the Content Store that matches the Interest, it will be sent out the face the Interest arrived on and the Interest will be discarded (since it was satisfied). Otherwise, if there is an exact-match PIT entry the Interest's arrival face will be added to the PIT entry's Requesting Faces list and the Interest will be discarded. (An Interest in this data has already been sent upstream so all that needs

to be done is to make sure that when the Data packet it solicits arrives, a copy of that packet will be sent out the face that the new Interest arrived on.) Otherwise, if there is a matching FIB entry then the Interest needs to be sent upstream towards the data. The arrival face is removed from the face list of the FIB entry then, if the resulting list is not empty, the Interest is sent out all the faces that remain and a new PIT entry is created from the Interest and its arrival face. If there is no match for the Interest it is discarded (this node does not have any matching data and does not know how to find any).

The concept of CCN has just been carried out, so far, there is still no widely adopted real implementation. But similar to the Freenet protocol, the routing of request (Interest) is based on the content and it offers a caching mechanism which stores popular content closer to the consumers. However, the CCN routing mechanism requires an hierarchical naming scheme of content, as well as significant disk space to maintain the routing table on each node.

6.3.3 Our proposal

In our package management network, participants may not have sufficient capacity to maintain the routing table and to precisely locate all the packages. But still we can adopt the caching strategy proposed in CCN and in the Freenet protocol. To avoid flooding-based searching, we expect to use gossiping based approach to send out requests on top of an semantic overlay network. The construction and management of the overlay topology is similar to that described in section 6.2.1. The only difference is that now, each peer possesses two neighbor lists (partial view) of limited size l_1, l_2. The first list is called the *semantic neighbor list*, which contains the l_2 peers having the most similar content (just as the same in VINICITY). The second list is called the *shortcut list*, which stores the l_2 peers that have recently been able to reply with requested data. The idea is that, if a peer has a piece of content that one is interested in, then it is likely that it will have other pieces of content that one is also interested in [24]. This is especially true for package retrieving as a peer having the metadata of a deployment unit is very possible to have the deployment unit itself, and a peer having one deployment unit u is very possible to also have other packages u depends on. While issuing a request, the peer first queries its neighbors in the shortcut list, if no successful reply, it will then gossip the request to some of its semantic neighbors. The request will propagate in the network, until the data is retrieved or the HTL is reached. Like CCN and Freenet, each peer also as a pending table that stores the received requests and the requesters. When the data is return, it is sent back according to the pending table. Along the path of the transmission, all the peers that relay the data will cache it in its local content store. Actually, our proposal here is a tradeoff between the Freenet and the broadcast searching based Gnutelle protocol.

7 Remaining challenges

To achieve our ambitious goal, an advanced peer-to-peer solution is required to tackle all the limitations that have been presented in the first part of this paper. The aforementioned gossip-based algorithms and techniques from information-centric network give a strong background for the conception of such a peer-to-peer system. However, we introduce now a selection of challenges, which have the potential to generate in the next decade an important research activity from both academic and industrial actors.

7.1 Device Heterogeneity

The administration of the zillions of heterogeneous devices in the *Internet of Things* is known to be a major challenge [25]. Both hardware (storage, computing, transmission) and software (operating system, software, middleware) can be radically different. In the solution sketched in this paper, every device can act as a repository, not only for machines with approximately the same characteristics, but also for all kinds of devices. Actually, any package can be stored, therefore a full-featured familial desktop can potentially become the main repository for many connected objects in a home. Further, such a direct collaboration between heterogeneous devices tackle the issues related with the administration of tiny devices: no direct connection to the Internet, almost no storage capacity, need of a dedicated repository maintained by the object seller, *etc.*

As attractive can be our decentralized solution, several challenges are still on the road:

- current peer-to-peer systems do not necessarily handle very well a so large heterogeneity. In particular gossip-based algorithms require every peer to have a minimum activity, which can be too costly for the least capable devices. This calls for the study of new algorithms for loosely hierarchical peer-to-peer systems (*e.g.* with some self-proclaimed super-peers). Furthermore, the techniques from information-centric networks are designed for equipments having large capacity. In a more constrained environment, new algorithms should be conceived.
- the proposed system should be platform-independent and operable on generic physical networks with a large variability of connectivity. Especially, the case of hybrid networks with both ad-hoc and Internet communication should be explored.
- new experimental platforms should be conceived. Based on massive virtualization techniques, these new experimental platforms should ideally be able to emulate a large-scale network where the operating system of elements (devices) are highly configurable. Such a platform is a first but necessary step toward the design of a package management system.

7.2 Certification Mechanisms

Package management systems provide privileged, central mechanisms for the management of software on computing systems, many packages being installed in the root

context of the operating system. Thus package management security is essential to the overall security of the computing system [26]. All the user-generated packages and meta-data must be tested and qualified before they are released in order to avoid the diffusion of malware. In previous works, the certification is done by the software distribution provider. In our proposal, as there is no central administration to rely on, we must create other certification mechanisms.

In order to provide peers with the capability of comparing alternative packages and choosing the best one to install, a reputation evaluation approach based on social network can be used to select the most powerful and trusted peers [27]. Peers express their opinions on other peers by assigning positive and negative scores, and gain or loss *reputation* by aggregating feedbacks from others. The higher score a peer has, the more it is trusted, and therefore may be considered as a good source for uploading. Trusted peers may be attributed a higher weight to their opinion while evaluating others. Peers may also be grouped into clusters according to their similarity. A quantified measure of the *distance* between two package dependency graphs should be defined, *e.g.*, the number of *install* and *uninstall* operations to be performed to transform one dependency graph to another.

However, some challenges have also to be tackled.

- pure peer-to-peer trust and reputation systems have not been yet implemented at a very large scale. The security requirements and the perspective to scale to the Internet of Things make that huge progresses should be done in the evaluation of these systems. This point is critical as the security concern is obvious in such context.
- these reputation systems based on social networks fits with the characteristics of gossip-based systems, but they do not necessarily match the behavior of information-centric networks. Moreover, trust building may require to handle a lot of data, which can be impossible for devices having small storage capacity. Hence, new algorithms for heterogeneous information-centric networks have to be designed.

8 Conclusion

In this paper, concepts and principles related with package management system have been introduced. Serval related works based on software repository and P2P overlay have been examined. They all exhibit certain limitations with respect to the growing of user-generated content and the evolution of the Internet of Things. Among all the challenges, we raised our envision towards a fully-decentralized packages management system for user-generated software and heterogeneous devices. Till now, we have defined a novel system model without the notion of central repository. We have also conceived several preliminary ideas for implementing the application level protocols. Some statistical analysis on Debian packages is still on going. In the next step, we are planing to establish a detailed implementation plan of our system and build simulations to test our proposal.

References

[1] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 199–208, 2006.

[2] Christos Gkantsidis, Thomas Karagiannis, and Milan Vojnovic. Planet scale software updates. In *Proceedings of the ACM SIGCOMM Conference*, pages 423–434, 2006.

[3] Ralf Treinen and Stefano Zacchiroli. Solving package dependencies: from edos to mancoosi. *CoRR*, abs/0811.3620, 2008.

[4] Nathan LaBelle and Eugene Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[5] Purvi Shah, Jehan-François Pâris, Jeff Morgan, John Schettino, and Chandrasekar Venkatraman. A p2p-based architecture for secure software delivery using volunteer assistance. In *Proceedings of the IEEE Eighth International Conference on Peer-to-Peer Computing (P2P)*, pages 131–139, 2008.

[6] Ubuntu blueprint for using torrents to download packages. `https://blueprints.launchpad.net/ubuntu/+spec/apt-torrent`.

[7] Cameron Dale and Jiangchuan Liu. apt-p2p: A peer-to-peer distribution system for software package releases and updates. In *Proceedings of the 28th IEEE International Conference on Computer Communications (INFOCOM)*, pages 864–872, 2009.

[8] Serge Abiteboul, Itay Dar, Radu Pop, Gabriel Vasile, Dan Vodislav, and Nicoleta Preda. Large scale p2p distribution of open-source software. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 1390–1393, 2007.

[9] Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *Operating Systems Review*, 41(5):2–7, 2007.

[10] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In *Proceedings of the 5th ACM international conference on Emerging networking experiments and technologies (CoNEXT)*, pages 1–12, 2009.

[11] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system, 2000.

[12] Shehnaaz Yusuf. Survey of publish subscribe communication system. `http://www.medianet.kent.edu/surveys/IAD04F-pubsubnet-shennaaz/Survey2.html`.

[13] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe, 2003.

[14] Lars Brenna, Cathal Gurrin, Dag Johansen, and Dmitrii Zagorodnov. Automatic subscriptions in publish-subscribe systems. In *ICDCS Workshops*, page 23, 2006.

[15] Gianpaolo Cugola, Davide Frey, Amy L. Murphy, and Gian Pietro Picco. Content-based routing for publish-subscribe on a dynamic topology: Concepts, protocols, and evaluation.

[16] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19:332–383.

[17] Spyros Voulgaris and Maarten van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Euro-Par*, pages 1143–1152, 2005.

[18] François Bonnet, Frederic Tronel, and Spyros Voulgaris. Brief announcement: Performance analysis of cyclon, an inexpensive membership management for unstructured p2p overlays. In *DISC*, pages 560–562, 2006.

[19] Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph, 1998.

[20] S. V. N. Vishwanathan, Karsten M. Borgwardt, and Nicol N. Schraudolph. Fast computation of graph kernels. In *In Advances in Neural Information Processing Systems 19 (NIPS 2006*, pages 1–2, 2006.

[21] Andrei Z. Broder and Michael Mitzenmacher. Survey: Network applications of Bloom filters: A survey. *Internet Mathematics*, 2004.

[22] Hailong Cai, Ping Ge, and Jun Wang. Applications of bloom filters in peer-to-peer systems: Issues and questions. In *NAS*, pages 97–103, 2008.

[23] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, pages 72–93, 2005.

[24] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. Efficient content location using interest-based locality in peer-to-peer systems, 2003.

[25] John J. Barton. Software upgrade in ubiquitous computing. Technical report, HP Labs, 2008.

[26] J. Cappos, J. Samuel, S. Baker, and J.H. Hartman. Package management security. Technical report, Department of Computer Science, University of Arizona, July 2008.

[27] Li Xiong and Ling Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Trans. Knowl. Data Eng.*, 16(7):843–857, 2004.