Chapter 1

Inter-package dependency networks in open-source software

Nathan LaBelle
Eugene Wallingford
Computer Science Department, University of Northern Iowa
labelle@uni.edu, wallingf@uni.edu

Internet-based repositories of open-source software provide a growing collection of software programs that interact with each other by code reuse. This software is developed by otherwise non-interacting, disjoint development groups with different goals, resources, and development methodologies. The software has a variety of functions, exists at assorted levels of maturity, and is written in a variety of programming languages. We have mined several large repositories and show that despite diversity in development groups and computer system architecture, resource coupling at the inter-package level creates small-world and scale-free networks with a giant component; which makes package networks similar to other natural and engineered systems. We document the effect that network structure can have on software robustness and security.

1.1 Introduction

Open-source software (OSS) applications are often distributed in the form of packages, which are bundles of related components necessary to compile or run an application. For many OSS packages, the source code is freely available and reuse of the code for derivative works is encouraged. Because of this, resource reuse is considered to be a natural pillar: a package is often dependent on resources in some other packages to function properly. These packages may be

third-party code libraries, bundles of resources such as images, or compiled utilities such as *grep and* sed. Package dependencies often span between project development teams, and since there is no central control over which resources from other packages are needed or in what ways, the software system self-organizes in to a collection of discrete, interconnected components.

The OSS system is interesting to complexity scientists for a number of reasons. Each of these reasons is both an assumption and an area for future research in the areas of both the dynamics of software engineering and the study of complex systems:

- Variable Goals – The system is developed by a diverse set of developers, each with a different set of goals and strategies. For example, some developers may spend a majority of effort on optimizing end-user usability, while other developers focus resources on algorithm efficiency. The goals of the development group are dependent on the nature of the project being created.

- Limited Information – No single developer has global system knowledge. We presume that each development entity can only process a limited amount of information about other packages in the system. How much information about other projects each developer in an OSS group can effectively utilize is an open question.

- Usability "In The Wild" – Recent estimates put the value of the Debian Linux Distribution at approximately $6.1 billion USD and claim that it contains over 105 million lines of code [1]. Popular OSS applications such as Apache, Mozilla, and the X-Window System are widely used.

- Adaptation – Because source code is available to users, they are encouraged to modify software to fit their uses. Under certain licensing schemes such as the widely-known GNU Public License, code reused in derivative works must be made public.

- Unsynchronized Development – Changes in OSS are not synchronized between groups. When a development group makes a change, it is typically logged with a version control system and "updates" such as bug fixes are ad-libbed into a plain-text changelog.

- Large System Size – At the time of writing, there are about $n = 22,000$ packages with more than $m = 84,000$ dependencies in the largest official Debian repository.

- Dynamic Structure – New software is constantly being created while older software is being modified. Additionally, a package may become "unmaintained" at any time, or may be admitted to a more mature repository as robustness and stability increase.

- Avalanches of Changes – Recent studies [5] have found that the amount of code added or removed fits a scale-free distribution. Depending on the level of coupling, modularity, and orthogonality, code avalanches can theoretically cascade through a connected system. There may also be long-term temporal correlations in software developer performance which yield chaotic behavior in development dynamics.

We have researched one aspect of complexity in OSS: the graph structure of package dependencies. The next sections describe basic statistical definitions for categorizing networks and previously studied networks of software components identified at in-program abstraction levels. §1.4 gives methods of data analysis and the data sets used. §1.5 describes our results, and the last section gives a discussion and areas for future research in the area of software networks.

1.2 Complex networks

A network is a usually unweighted and simple graph $G = (V, E)$ where V denotes a vertex set and E an edge set. Vertices represent discrete objects in a system, such as social actors, economic agents, computer programs, or biological producers and consumers. Edges represent interactions among these "interactons". If software functions are represented as vertices, edges can be assembled between them by defining some meaningful interaction between the functions, such as inheritance or procedure calls.

The degree of a vertex v, denoted v_k, is the number of vertices adjacent to v, or in the case of a directed graph either the number of incoming edges or outgoing edges, denoted k_{in} and k_{out}, respectively. The distribution of edges in real-networks roughly follows a power law: $P(k) \sim k^{-\alpha}$. That is, the probability of a vertex having k edges decays with respect to some constant $\alpha \in R^+$ which is typically between 2 and 3. This is significant because it shows deviation from randomly constructed graphs, first studied by Erdös and Rényi and proven to take on a Poisson distribution as $n \to \infty$, where $n = |V|$ [12]. The power-law distribution of edges implies that many vertices will not be highly connected and will have one or two edges, while some vertices will be hubs of the network and contain thousands of edges.

Real-world networks have two specific length-scaling features not found together in random networks: (1) a low characteristic path length and (2) a high degree of clustering [12]. Together, these properties are known as the "small world" (SW) effect, popularly known as "Six Degrees of Separation". Let L be the mean shortest path length between nodes and C be the average clustering. The clustering coefficient C is the propensity for local cliques to form, and is much higher in SW networks than in randomly constructed networks [12]. If a vertex v is adjacent to vertices u and w, then u and w are more likely to be adjacent to each other than in a random network. The clustering coefficient measures this probability, which is the likelihood of the neighbors of a vertex to also be neighbors. A network is small world if $L_r \approx L$ and $C_r \ll C$ where C_r

and L_r denote the clustering and mean geodesic path length respectively, for an equivalently sized randomly constructed network. The closed-form of both C_r and L_r given a network size n and a number of edges m is widely known [12].

A connected component Ω is a subgraph of G such that for all pairs $(i, j) \in V$, there exists a path from i to j. The size of the connected component is the number of vertices such that there exists an edge traversal path between each vertex in the set. For real-world networks, the size of this set is approximately 90% or more.

1.3 Software networks

Previous research in networks of software has focused on software at low levels of abstraction relative to the current research, and has concentrated on dependencies of components inside a single program. In this paradigm, networks may be constructed either from run-time analysis of objects stored on the programs heap or compile-time analysis of source code by using syntax analysis. Clark and Green [2] found Zipf distributions, a ranking distribution similar to the power-law and also found in word frequencies in natural language [14], in the structure of CDR and CAR lists in large Lisp programs during run-time.

In program source code the small world effect and power-law edge distribution in networks of objects (procedures) where edges represent meaningful interconnection between entities, such as inheritance [10, 8]. In the case of procedural languages, procedures are represented as vertices and edges between vertices symbolize function calls. Run-time networks have similar statistical features as source-code networks [8]. Networks where the vertices represent source code files on a disk and edges represent a dependency between files rather than objects or procedures have been studied with similar results [6] . The small world property and a power-law distribution of edges has also been found in documentation systems such as JavaDoc [13]. From a complexity science perspective, each of these results is significant because it shows tendency towards similar behavior at varying abstraction "scales".

1.4 Methodology

We mined official software repositories from two UNIX-like operating systems: Debian and FreeBSD (BSD). The BSD repository contains two subsets: compile-time dependencies and run-time dependencies. These were mined from the PORTS system [4]. The Debian Linux Distribution (DLD) contains several sub-distributions of binary (run-time) packages. These include 11 different architectures ranging from lightweight processors such as ARM to desktop systems as i386 and Mac68k, as well as high-performance architectures such as SPARC and Silicon Graphics and supercomputing architectures such as Hitachi. Each architecture also normally has three sub-repositories which dictate code maturity: Unstable, Testing, and Stable. Code maturity is determined by the number

of bugs in a specific software package. Admission to the three groups is increasingly rigorous. A new version of a program may reside in Unstable for some time before being moved to Testing, and finally to Stable. This categorization is determined by the repository managers. However, the large system size and inability to catch all bugs before admission limits the effects the repository managers can have on the network. In this research, all packages, including conflicting packages (which should not be concurrently installed on a system, as they may contrast with each other) are included.

1.5 Results

For the Debian unstable (i386) 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 $C_r \approx .0019$ and $L_r \approx 7.41$. 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 $\alpha_{\text{out}} \approx 2.33$. The distribution of incoming edges, which measures how many packages are dependent on a package, follows a power-law with $\alpha_{\text{in}} \approx 0.90$. 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 $\bar{k} = 7.27$ other packages. For the BSD network, $C \approx 0.56$ and $L \approx 2.86$. An equivalent random graph would have $C_r \approx 0.007$ and $L_r \approx 7.11$. Hence, the BSD network is small world. The degree distribution of the BSD network also resembles a power-law, with $\alpha_{in} \approx 0.62$ and $\alpha_{out} \approx 1.28$. For the run-time network, results were similar: the run-time network is both small world and follows a power-law.

Table 1.1 gives values for various architectures and maturity levels in the DLD. From the small world values of C and L, we can see that the network is more highly clustered than the equivalent random networks, whose values are given as C_r and L_r. We can also see that a large connected component emerges in each of the package networks. In each network, there is a tendency towards all-or-none behavior in joining the component: those packages that are dependent on other packages, or dependent upon other packages, all join the same component. This gives rise to a number of packages that are not connected at all to the large component. Thus, there is only one component in each graph larger than 1 package. From Table 1.1 we can see the similarity in package statistics despite maturity level and architecture. The small size of the AMD64 repository is due to its relatively recent introduction. It is an open question how the rate and quality of software ported to this architecture are influenced by user adoption, demand, and difficulty.

Table 1.1: Statistics for the DLD Package Networks

	n	m	$\|\Omega\|$	C	C_r	L	L_r
i386 Unstable	22264	84437	87.9%	0.52	0.0019	3.341	7.41
i386 Testing	21310	75699	85.7%	0.49	0.0016	3.441	7.50
i386 Stable	19677	74642	88.8%	0.533	0.0019	3.352	–
AMD64 Unstable	11061	15980	63.8%	0.50	0.0013	5.516	25.3
Alpha Unstable	21537	83722	87.5%	0.53	0.0018	3.411	7.34
Alpha Testing	20814	78198	86.5%	0.49	0.002	3.543	7.51
Alpha Stable	18687	70696	88.1%	0.53	0.002	3.355	7.39
HP-PA Testing	20466	77918	86.1%	0.54	0.0018	3.402	7.42
HP-PA Stable	18670	70599	88.1%	0.53	0.002	3.357	7.39

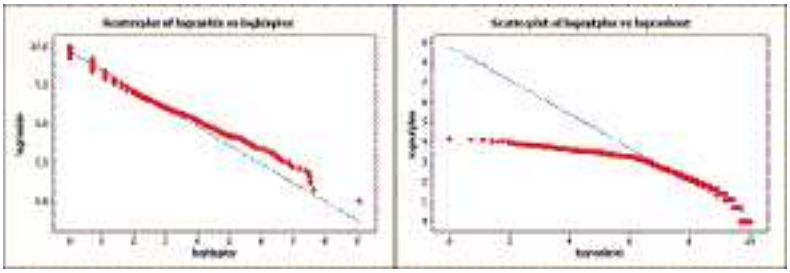

Figure 1.1: Zipf distribution of edges k_{in} and k_{out} (respectively) on a $\log - \log$ scale.

Ranking the node edge values in the DLD network and plotting *rank* versus *value* on a $\log - \log$ scale allows us to visualize the Zipf distribution of edges in the network. These fits are shown in Figure 1.1. The finite effects of the out-degree are shown. The out-degree is limited in software packages. No package depends on thousands of packages for use, implying that (1) no package that makes use of most other packages for functionality exists, (2) that there is no demand for an all-inclusive package, or even (3) no package is large enough to require resources from every other package – it is possible to provide a plethora of functionality by combining packages, rather than by reinventing the wheel.

In the Debian network, the 20 most highly depended-upon packages are libc6 (7861), xlibs (2236), libgcc1 (1760), zlib1g (1701), libx11-6 (1446), perl (1356), libxext6 (1110), debconf (1013), libice6 (922), libsm6 (919), libglib2.0-0 (859), libpng12-0 (622), libncurses5 (616), libgtk2.0-0 (615), libpango1.0-0 (610), libatk1.0-0 (602), libglib1.2 (545), libxml2 (538), libart-2.0-2 (524), and libgtk1.2 (474). The number in parentheses represents the number of incoming edges. The list is composed mainly of libraries that provide some functionality to programs such as XML parsing or that provide some reusable components such as graphical interface widgets. Because the most highly-connected package (libc6) is

required for execution of C and C++ programs, we can infer that these are the most widely used programming languages.

1.6 Discussion & Future Research

The package network is particularly interesting because it is diverse, distributed, relevant, and large. We have shown that despite differences in computer architecture, development groups, development methodologies, programming language, program goals, resources, and levels of maturity, the OSS package network has an organization similar to other scale-free and small world networks. Since software interconnections are scale-free inside programs, with highly referenced objects or procedures being extremely important for optimization, we hypothesize that at the level of dependencies between programs, the scale-free characteristics may be a signature of reuse at lower abstraction levels. Of interest to complexity scientists is the hypothesis that at every scale of computer program usage, from inner-program data structures, objects and procedures, to inter-program reuse, the small world and scale-free observations are supported. Of interest to software engineers is the fact that despite architecture, resources, design methodologies, code maturity, requirements: the organization of software at the repository level is similar.

The effects on software engineering are widespread. In 2004, a number of security vulnerabilities were identified in libPNG, a library used for the manipulation of Portable Network Graphics image files [9]. Because libPNG is referenced by some 622 packages, each of these dependent packages may have unforeseen vulnerabilities, depending on how they use libPNG. This is one example of the network's effect on software. The scale-free nature of the frequency of dependencies implies that many packages will be relatively unconnected, while some packages will be orders-of-magnitude more connected.

Future research in package networks can take several directions. First, the dataset can be changed to include more computer software from other hardware and software combinations, such as MacOS or Windows DLL relationships or different repositories such as SourceForge. Secondly, new methods for extracting additional information from the network should be developed. This information is especially valuable if it can alleviate problems in the difficult task of software development. Also, there is no model of network formation that takes software dynamics (reuse, refactoring, addition of new packages) in to account. The impact of the network structure on software dynamics should be investigated.

Bibliography

[1] Amor, Juan Jose, Robles, Gregorio, and Gonzalez-Barahona, Jesus. "Measuring Woody: The size of Debian 3.0." Arxiv: cs.SE/0506067.

[2] Clark, Douglas, and Green, Cordell, "An empirical study of list structure in Lisp", *Communications of the ACM* **20** (1977), 78–87.

[3] The Debian Organization. Debian Packages, http://packages.debian.org.

[4] FreeBSD. "The FreeBSD Ports Collection", http://www.freebsd.org/ports.

[5] Gorshenev, A. A., and Pismak, Y., "Punctuated equilibrium in software evolution" em Physical Review E **70** (2004), 067103.

[6] de Moura, Alessandro, Lai, Ying-Cheng, and Motter, Adilson, "Signatures of small-world and scale-free properties in large computer programs", *Physical Review E* **68** (2003), 017102.

[7] Myers, Chris, "Software systems as complex networks: structure, function, and evolvability of software collaboration graphs", *Physical Review E* **68** (2003), 046116.

[8] Potanin, Alex, Noble, James, Frean, Marcus, and Biddle, Robert, "Scale-free geometry in OO programs", *Communications of the ACM* **48** (2005), 99–103.

[9] U.S. Computer Emergency Readiness Team. "Technical Cyber Security Alert TA04-217A". Accessed October 25, 2004 from http://www.us-cert.gov/cas/techalerts/TA04-217A.html (2004).

[10] Valverde, Sergei, Cancho, Ramon Ferrer, and Solé, Ricard (2002), "Scale-free networks from optimal design" *Europhysics Letters* **60** (2002), 512–517.

[11] Valverde, Sergei, and Solé, Ricard, "Hierarchical small-worlds in software architecture", *SFI/03-07-044*, The Santa Fe Institute, (2004).

[12] Watts, Duncan, and Strogatz, Steven, "Collective dynamics of small world networks", *Nature* **393** (1998), 440–442.

[13] Wheeldon, Richard, and Counsell, Steve, "Power law distributions in class relationships", *Third IEEE International Workshop on Source Code Analysis and Manipulation*, IEEE (2003), 45–54.

[14] Zipf, George, *The Psycho-Biology of Language: An Introduction to Dynamic Philology*, MIT Press, 1965.