From Bazaar to Kibbutz: How Freedom Deals with Coherence in the Debian Project

Mattia Monga
Università degli Studi di Milano
Dip. di Informatica e Comunicazione
Via Comelico 39 – 20135 Milan, Italy
mattia.monga@unimi.it

Abstract

The goal of obtaining a coherent distribution of software packages where all programs interact smoothly increases its complexity with the number of applications, the number of architectures involved, and the number of system configurations supported. The Debian project aims at producing a software system with thousands of components running on eleven different hardware architectures, with three different operating system kernels. This paper describes the project and how the work of hundreds of people that never meet one with another can be coordinated to produce reasonably robust and integrated systems.

1. Introduction

Applications do not exist in the desert. In fact, they run in very complex environments where an operating system kernel, some device drivers, system and graphic libraries, common services, etc. coexist in order to provide the software platform on which users can enjoy their applications.

For this reason, from the beginning (see for example the GNU Manifesto [10]) proponents of *free software* aimed at producing complete systems, because no real freedom is possible if developers have to rely on non free components. Historically, a major hurdle on this goal was the unavailability of an operating system kernel. However, in the last fifteen years several kernels (e.g., Linux, FreeBSD, GNU/Hurd, etc.) were made available to the *open source* community, and it was feasible to build entirely open source computing platforms which integrate basic utilities with sophisticated application software.

Discussion on open source software development often focuses on the techniques used to organize an open source project aimed at producing a well defined application. A famous paper by E. Raymond [9] describes a style of development metaphorically called *bazaar*. In a software bazaar anyone could contribute code to the original promoters of the project, who take care of integration in the mainstream code. This approach is contrasted by Raymond to the traditional software engineering process, that in another famous writing [2] F. Brook compared to the approach people used to build *cathedrals*, where an architect leads a small group of skilled and specialized workers, with precise schedules and responsibilities. In fact, as recent studies have shown [6], some of the most popular open source projects (e.g., the Apache web server, the Linux Kernel, the Mozilla browser) are in between the two extremes: the project is carried on and scheduled by a core group of developers, strongly committed to the product, and the openness of the source code enables contributions from individuals who correct some bugs or add some features. These contributions are often scarce and it can be safely assumed that in general open source projects have a core of developers, no larger than 10 to 15 people, who control the code base, and it is responsible for 80% of written code.

In this paper I want to discuss how an open source community can produce an integrated system, composed by aggregating several software packages that typically derive from independent sources. These systems are commonly called *distributions*. To date (February 2004), the Linux Weekly News list of Linux distributions contains 374 items [1]. Indeed, one of the business models proposed in order to make profit from open source software is selling the added value of an assembled distribution [5], and consequently some of the most successful distributors (Red Hat, SUSE, Mandrake, etc.) are commercial firms, driven by tight coupled groups of developers. Instead, in the following sections I will focus on the Debian project, aimed at producing a coherent distribution of free software leveraging only on the work of independent volunteers.

As previously said, the limiting case of open source software development process was metaphorically compared to bazaars, in which contributors put their work in the "magic cauldron" of the community. However, building a coherent distribution requires a great effort of coordination and cooperative work, thus the bazaar metaphor seems completely inappropriate. Yet, in the case of Debian, the cathedral metaphor is also inadequate, since no main architects are present and the work is carried entirely on a voluntary basis. Therefore, I suggest the new metaphor of the *kibbutz*[1], for a cooperative

1 Kibbutzim are Israeli communal form of agricultural settlement. Originally it was predominantly agricultural and practiced a very high level of sharing,

community of volunteers sharing a common goal. The properties that characterize such a community are:

- people join the community on a voluntary basis, and they do not expect to be paid for their work;
- members agree on an ambitious final goal ("making the desert where users live blooming of good and free software");
- members share a civil consciousness and they accept that their work is regulated by explicit rules established by direct democracy.

Though voluntary work is an essential part of Debian, given the great number (thousands) of people involved in the project, I believe that a study of their coordination effort can be valuable in a greater context, since most of the ideas have to do to management of complexity and heterogeneity and I suggest they could be applied also to commercial organizations.

The paper is organized as follows: Section 2 presents the Debian project and its organization, Section 3 describes the development process it adopts and how coordination is achieved, Section 4 explains how Debian systems can be customized and finally Section 5 draws some conclusions.

2. The Structure and Goals of Debian

2.1 The Debian Motivation

The Debian[2] project was started by Ian Murdock on August 16th, 1993 in order to "carefully and conscientiously put together and maintain and support with similar care" a distribution of Linux software by working "openly in the spirit of Linux and GNU" [7]. From the beginning all Debian members were volunteers and they are still not paid by Debian to do their work in the project. However, from November 1994 to November 1995 Debian was sponsored by the Free Software Foundation and Debian motivated the creation of "Software in the Public Interest", a non-profit organization that provides a mechanism by which The Debian Project may accept donations. The money collected pays hardware and actual duty expenses of Debian representatives.

A Linux distribution puts together pieces of software that are in general built by people unrelated with the distributors themselves. The Debian project requires that software included in a Debian system is compliant to the "Debian Free Software Guidelines": basically, software has to be licensed with an open source license [5] that allowed freedom of use, distribution and modification without discriminations and restrictions that can affect unrelated code.

The first release to a greater public of a Debian GNU/Linux system was issued on January 1994 (ver. 0.91). It contained a few hundreds of programs and was put together by a dozen of developers. Today (January 2004) the project count 1268 members distributed worldwide and it manages more than 13,000 binary packages (corresponding to more than 8,000 of source packages) ported to 11 different architectures (i.e., Alpha, arm, hppa, i386, ia64, m68k, mips, mipsel, powerpc, s390, sparc) [4]. At least three complete Debian systems exist: beside the main Linux based one, there is one based on the BSD kernel and another based on the GNU Hurd kernel. The original Debian founder, Ian Murdock, does not work actively in the project since 1996.

2.2. The Debian Structure

Everyone can apply to become a Debian member. In order to be accepted in the project one has to demonstrate the control of the basic skills needed to manage software packages and the understanding of the "Debian Free Software Guidelines" and the "Debian Social Contract"[3].

By joining the group one gives his or her consent to contribute to the project according to the Debian *Constitution [8]*. The Constitution defines a lean organization with a *Project Leader (DL)*, a *Project Secretary (DS),* a *Technical Committee (TC)*, and *Individual Developers*. The DL, DS, and the chairman of the TC has to be three different persons. The work is entirely voluntary: nobody is obliged to do anything and everyone chooses freely to be assigned to a task he or she does find useful or interesting. A new DL is appointed every year by a general election involving all the individual developers that vote with a Condorcet's mechanism. The DL can make urgent decisions and he or she appoints the DS and, together with the TC, renews the members of the TC itself. The TC is composed up to 8 members, with a minimum of 4 people, and it decides technical policies and it composes developers disagreements. Individual developers can override any DL and TC decision by issuing a general resolution with a qualified majority. The DS is appointed by DL and the previous DS every year. The DS is in charge of managing elections and other calls for vote and it adjudicates any disputes about interpretation of the constitution. The properties and financial activities are managed by "Software in the Public Interest, Inc" (SPI), in which every Debian member can be a voting member.

The consequence of this organizational structure is that no single individual can take personal control of the project. Even better, "Any individual Developer may make any technical or nontechnical decision with regard to their own work." [8] However, since coherence of the final product is one of the goals on which members agree, this absolute freedom has to be temperated by coordination achieved by a number of *policies*, that, after discussion on the mailing lists (most of them are public

including collective rearing of children. More recently (by 1998) industries have taken over a significant role in the Kibbutz economy.

2 The official pronounciation of Debian is 'deb'~ee~en'. The name comes from the names of the creator of Debian, Ian Murdock, and his wife, Debra.

3 The Debian Social Contract states that the Debian project will be always free software (according to the definition of the Debian Free Software Guidelines), it collaborates with the free software community and it follows procedures open to the public.

and also non developers can contribute to the discussion -- see http://lists.debian.org), are defined by the TC, but they should meet a high degree of general consensus to not be overridden by general resolutions. I will discuss policies further in Section 3.2.

In order to study the Debian organization it is important to consider a number of actors that interact with the Debian galaxy, without necessarily being members of the project. Yet, they may influence the Debian work.

First of all there are *Upstream Authors*. They contribute to Debian by writing open source software. In theory they could not even know about Debian. In practice they are often in direct communication with Debian developers, because inside Debian a lot of work is done to discover and correct bugs. Thus, it is common that Debian developers (from now on, DDs) forward to upstream authors bugs, patches, suggestions, new features requests, etc.

Secondly, there are *Users*. Satisfaction of users is of course an important force that indirectly drives the project. Moreover, Debian systems provide a sophisticated infrastructure for bug tracking (Debian Bug Tracking System, DBTS). It is the main avenue through which users can report problems and propose enhancements. It is important to understand that it plays a critical role in the pursuing of coherence, since it is used also to report bugs of the distribution itself. For distributions the same Linus' Law [9] of generic software applies: when they are exposed to a great number of observers, with different needs and slightly different operating environments, all bugs are shallow.

A third category that is worth mentioning for its increasing significance is the one composed by people that use Debian systems to build their own *specialized distributions*. The openness and intimate coherence of Debian systems make them ideal candidates to be customized for specific purposes. The most successful customization is probably the Knoppix distribution, aimed at producing a system running entirely from a CD and able to recognize automatically a huge set of different hardware on i386 machines. As I will discuss in Section 4, the open architecture of Debian system is particularly apt to customizations without necessarily going out of sync with the mainstream Debian distribution.

3. The Debian Development Process

3.1. Debian Distributions

A distribution of a Debian system is composed by an installation program and a set of software packages. The installation program is able to set up the system from scratch on a large number of different hardware configurations: this makes the installation a quite complex operation. Software packages can be retrieved from a set of CDs, a local hard disk or the network.

All the Debian development effort is focused on the production of packages. A *package* is the minimal unit that can be installed or removed from a system. Consequently, each DD is responsible for one or more packages, and he or she is said to be the *maintainer* of that package[4]. When a maintainer has put together his or her package, it is uploaded to a public repository from where Debian users worldwide can try to install it on their systems. Since up to now the package was tested only on the DD's machine, its status should considered *alpha-testing* and the repository is called the *unstable distribution*. However, notwithstanding the scaring name, a considerable number of users (and virtually all the DDs) tries packages from the unstable distribution, thus the test is quite significant. If a package lives in the unstable distribution for ten days without any critical bug is notified, it is *automatically* uploaded to another, more stable, repository corresponding to a *beta-testing* status. This repository is known as the *testing* distribution. Approximately yearly, a *Release Manager* is appointed by the DL, and starting from the testing distribution a set of packages is *frozen*. This means that no new packages can be added to the set, included packages evolve only for bug correction, and eventually, when all *release critical bugs* are corrected, a new *stable distribution* is released to the public. The stable distribution is what is normally considered the official Debian distribution and included packaged are updated only for fixing security vulnerabilities.

It is worth noting that DDs normally produce their package on a specific architecture (the most common is i386). However, unless the package control file specifies explicitly that its use is restricted to a single architecture, every package inserted in the unstable distribution is automatically build for all the architectures considered by Debian (eleven, to date) and it can enter in testing only if the build process is successful.

3.2. Coordination

The goal of obtaining a coherent distribution where all programs can interact smoothly is a very complex one. The problem seems without a solution if a distribution is obtained by aggregating thousands of packages produced by hundreds of developers on dozens of different systems configurations. Nevertheless, Debian systems were able to obtain a quite good overall user satisfaction, as testified by several awards won in 2003 (Linux journal readers' choice, Linux enterprise readers' choice, Linux new media award). In fact, the main effort carried on by DDs is directed to ensure that their packages are fully compliant to Debian *policies*.

Policies are key in the Debian approach to software distribution. Freedom of DDs is unlimited as long as they comply to their collectively agreed policies. Policies are often based on international or community standards (e.g., the Filesystem Hierarchy Standard [3]) and they concern all the global issues that affect the coherence of a system: i.e., libraries deployment, environment variables,

4 A few complex applications, i.e., the XFree86 package, are maintained by a team of four or five people

shared services, scripting languages. They sometimes take the form of general principles ("Maintainer scripts must be idempotent"), but more frequently they assert some automatically checkable property of the installed package ("Link targets like foo/../bar are deprecated"). For complex subsystems special sub-policies exist: for example, the Emacs extensible editor has its own policy that reduce possible conflicts among the huge number of emacs-specific packages coming from different sources.

Policy enforcement is pursued at different levels, in order to exploit cross validation to minimize inconsistent packaging:

- during package assembling: most of the policies are associated to a tool (collectively called "debhelpers") that ensures the correct application. For example, documentation can be introduced in a package by using the script *dh_installdocs*; it guarantees that when the package will be installed, the documentation files will be put in */usr/share/doc* and compressed with *gzip.*
- during package testing: several tools exist to check policy compliance before uploading the package to the public repository. The most important one is *lintian*, a script that analyze a package for about thirty categories of policy violations. Moreover, when a packaged is uploaded to a public repository some critical checks are repeated and the package is refused if checks fail.
- during package deployment: every user that detects an incoherence can issue a bug with an automated procedure (*reportbug*). Since policies are public and available on every Debian system, also not harmful violations can be in principle discovered (and bug reports show that they often are) and notified to DDs.

Every package implicitly assumes a working environment providing to it some services. DDs should make explicit these assumptions by defining a set of *dependencies* for each package. The richness of Debian dependency language enables fine tuning of installed systems: if A *depends* on B, B must be installed in order to install A; if A *recommends* B, most users would not run A without B; if A *suggests* B, B may enhance A functionalities, but A can be used in most cases also without B. Moreover, two packages can *conflict*, a package may *replace* another, and a package A can *provide* the functionalities of B. The latter relationship makes useful the existence of *virtual packages* (e.g., a generic mailer application) that can be required by others. In order to foster reuse and avoid duplications, Debian promotes *micro-packaging,* therefore it is common that from a single source package several binary packages are generated. Thanks to these dependency relationships, installing a new application on a running system can be as painless for users as typing a "apt-get install application" command: all the required packages are retrieved from a public repository (possibly on a set of CDs), installed and configured. In most cases even running services can be upgraded in this way, since Debian policies define standard mechanism for stopping and restarting daemons. Moreover, when an application is removed, it is possible to check which libraries were "orphaned" (i.e., they are no more requested by any package) by this removal and remove them too.

4. Customizing and Mantaining a Debian System

One of the added value of open source systems is that they can be customized to better satisfy user needs. However, customization is also risky. A highly customized system can be very difficult to keep in sync with the mainstream open source development. Suppose for example that a user wishes to use a program *java-local* rather than the program *java* provided by the Debian "java" package. If the user overwrote */usr/bin/java* with *java-local*, the package management system will not know about this change, and it will discard the customization on upgrades. For this reason, Debian introduces the concept of *package diversion*, by which users can maintain their *diverted* versions of programs, while enjoying mainstream upgrades. For example, by issuing the command *dpkg-divert --divert /usr/bin/java.debian /usr/bin/java* all future installations of the Debian "java" package will write the file */usr/bin/java* to */usr/bin/java.debian*. Moreover, several alternative equivalent programs can be installed in a system and simple infrastructure can be used to keep a generic name linked to the preferred alternative (e.g., *x-www-browser* may point to *galeon*, even if both *mozilla* and *galeon* are installed. These facilities make Debian systems ideal to be used as a starting base for specialized distributions: successful examples are the Knoppix distribution (running entirely from a CD), and the Familiar distribution (intended to be run on PDAs): while very different, they all share the same packet infrastructure and they keep reusing the daily work of DDs notwithstanding their customizations.

Another problem that sometimes hurdles users in upgrading their customized systems, is that configuration options can be discarded by the new version of applications. Roughly speaking, the configuration of an application is a three steps process. Major options are set system wide when the application is installed; they can affect major issues (for example, how a program is started: if it is an *inetd* daemon or if it is *SUID root*) and they are only rarely modified. Other less important options are more frequently changed by editing configuration files. User options are changed by users themselves and settings are stored in their home directories. In Debian systems, preservation of major options across upgrades is achieved by exploiting the *debconf* database. When a new application is installed for the first time some questions are asked to the user. The answers provided by the user are stored in that database and when a new version of the application is going to be installed only new options are presented to the user. Users' choices are preserved for unchanged options and the installation script is responsible for traducing them in

the possibly new syntax of configuration files. Moreover, every time an upgrade affects a system wide configuration file, a warning is issued, asking which version the user wants to keep and, if the files are human readable text files as it is common in the Unix world, differences can be merged together. Instead, no support is provided at the moment to evolve end-user options. However, these are often just cosmetic ones and therefore much less critical.

Another approach is worth mentioning in this paper is what can be called *aspect oriented package maintenance.* In any system complex enough, there are issues that *cross-cut* the whole system and cannot be easily packaged in an isolated module. The Debian solution to this problem follows an aspect oriented approach: special events of the package life cycle are exposed to other packages and they can, *obliviously* from other packages points of view, introduce actions that will be performed when these events occur. For example, the *localepurge* package aims at not installing all localized files (i.e., files specific for different languages) not explicitly preserved by the user of the system (it is a big waste of space to install non useful Japanese documentation files if nobody reads Japanese!). Other packages know nothing about *localepurge*, but, when it is installed, its execution is needed during their installation. Therefore, the package installation system (clearly a cross-cutting issue) can be customized by specific programs, that may subscribe themselves to be executed when well defined events occur (basically installation and removal of a package)

5. Conclusions

The goal of obtaining a coherent distribution of software packages where all programs interact smoothly increases its complexity with the number of applications, the number of architectures involved, and the number of system configuration supported. The Debian project copes with this complexity with an approach that does not resemble neither the *cathedral* model with a single architect with unlimited power, nor the *bazaar* model where the only coordination force is mutual interaction. Instead, freedom of action is preserved, and a democratically decided coherence is pursued as far as possible by technical means. The Debian coordination effort to manage complexity and heterogeneity should be studied in depth in order to understand which techniques can be applied conveniently also to commercial organizations.

Acknowledgments

I would like to thank all the members of the Debian project for their voluntary work. I am a member of the Debian project since 2000. However, the opinions expressed in this paper, except the ones taken from cited official documents, are my own only responsibility.

6. References

[1] *Linux weekly news*. http://www.lwn.org/distros, 2004.

[2] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley publishing company, USA, anniversary edition, 1995.

[3] Freestandards.org. *Filesystem hierarchy standard*. http://www.pathname.com/fhs/, 1998.

[4] B. Garbee. *Where would you like 100,000 users to go today*. http://www.gag.com/~bdale/talks/2004/adelaide/keynote/bdaleLCA.html, 2004. Linux Conf. Australia, Adelaide.

[5] Open Source Initiative. *Open source licences*. http://www.opensource.org/licenses/, 2004.

[6] A. Mockus, R. T. Fielding, J. Herbsleb, *Two case studies of open source software development: Apache and Mozilla*. ACM Trans. Softw. Eng.

[7] I. Murdock. T*he Debian manifesto*. http://www.debian.org/doc/manuals/project-history/ap-manifesto.en.html, 1993.

[8] Debian Project. *The Debian constitution*. http://www.debian.org/devel/constitution, 1998.

[9] E. S. Raymond. *The cathedral and the bazaar*. http://www.tuxedo.org/esr/writings/cathedral-bazaar/, Nov. 1998.

[10] R. Stallman. *The GNU manifesto*. http://www.gnu.org/gnu/manifesto.html, 1985.