Open Source Watershed: Studying the Relationship Between Linux Package and Distribution Releases

by

Scott Shawcroft

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science
With Departmental Honors

Computer Science & Engineering

University of Washington

June 2009

Presentation of work given on June 4 2009
Thesis and presentation approved by ________________________________
Date ______________________________

Contents

Abstract

Open Source Watershed is a research project aimed at gathering data to study the relationship between individual software projects, also known as upstream, and GNU/Linux distributions, also known as downstream. This is done by tracking package names, versions and release dates both upstream and downstream. Data is collected, or crawled, through a number of Python scripts. Upstream sources are commonly crawled by listing http or ftp directories or RSS feeds. Eight different distributions are crawled. They were chosen because of their unique release and package management styles. The eight are Arch, Debian, Fedora, Gentoo, OpenSUSE, Sabayon, Slackware and Ubuntu. After the data is gathered it is then normalized before analysis. Normalization consists of linking package names together across different sources and interpreting the version numbers.

After normalization, the data can be analyzed in a number of different ways. This report analyzes the data in four different ways. First, the number of upstream releases over the data collection timeline is graphed as a histogram. Second, the fraction of obsolete downstream package versions is studied over time for each distribution. Third, the average number of newer upstream versions for each package is studied over time. Fourth, and last, the average time since the oldest new release of a number of downstream packages is graphed over time. These four methods of analysis provide a visible distinction between fixed and rolling release cycle distributions. Furthermore, the analysis fixates common anecdotal knowledge about how up-to-date the different distributions are.

In the future, further work will lead to a larger set of quality data and a corresponding increase in the quality of the analysis. Additionally, more of the research data will be exposed to the large open source community as a tool for improving the efficiency of both upstream and downstream development.

1 Introduction

Creating large software systems out of many individual pieces is an engineering feat. There are many different approaches to maintaining these large systems and some techniques are better than others. One difference between techniques is the frequency of integrating components. For example, some projects continually upgrade the larger system while others integrate once every fixed period of time. These two different techniques lead to different evolutionary properties of the larger system. However, this integration does not happen in isolation. The effectiveness is a result of many other additional factors such as the developer community, integration tools and task management techniques.

In this paper, I will discuss the techniques for studying the coalescence of a variety of GNU/Linux systems. The results are then two-fold. First, the data gathered in this research is used as an aid to distribution maintenance through the research website. Second, the aggregate data is analyzed to discover packages which are maintained quickly in all distributions and distributions that maintain their packages in a timely fashion. These two techniques will then lead to a faster iterative development cycle between upstream and downstream.

2 Background

In 1991, Linux Torvalds began creating a new operating system kernel. However, at first it was very difficult to run as part of an entire operating system environment because the necessary components, such as a standard library and compiler, were scattered all over the Internet. As a result, GNU/Linux distributions appeared. They aggregated all of the standard components of a fully fleshed out operating system into a single installation medium, then floppies and now CDs and DVDs. This creation of distributions allowed for those without Internet to run the free operating system created from the Linux kernel, GNU tools and other software. At first, upgrades were centered around new media releases. However, as Internet use broadened and bandwidth increased, more distributions began to distribute additional software over the Internet to supplement the installation media. This eliminated the restrictions on the number of programs a distribution could distribute. However, while the repository of software size grew larger, there was still an emphasis on large simultaneous releases of new version within this repository. In fact, this technique is still largely visible in today's distributions. However, there are also newer distributions that continually release software such as Gentoo and Arch. These distributions are relatively new and feature new software management systems designed for continual, or rolling, release.

This variation in distribution creation and maintenance methodology is the focus of my research. In my opinion, open source software development is being held back by the lag time of the distributions behind upstream developments. Reducing this lag will reduce the time it takes for user feedback to reach upstream developers which in turn will reduce the time for a new upstream release.

Some will argue that reducing this time between upstream and downstream releases will decrease the stability of distributions. However, I firmly believe that, on the whole, upstream developments are improvements. Although there are times where regressions occur in upstream packages, the extra overhead of maintaining the old versions is not worth the effort. For example, a distribution may work to patch a problem fixed in a newer upstream version instead of updating the entire package because the process for integrating a completely new version is costly. This research hopes to find methods for reducing this cost. This savings will then lead to a distribution whose development is tied closer to upstream developments, creating a more responsive feedback cycle of downstream and upstream development. As a result, more development effort will be spent on upstream progression which benefits all distributions.

3 Procedure

The process of analyzing the upstream and downstream relationship is a three step process. The first step is gathering the data from the various distributions and upstream packages. Second, the gathered data must be normalized so that comparisons can be made between a distribution's package, an upstream package, and another distribution's package. Lastly, this data is analyzed, aggregated, and made available on the research website, *oswatershed.org*.

3.1 Gathering Data

The amount of data relevant to this problem of analyzing the relationship between upstream and downstream is enormous. My research focuses on gathering release dates and times for packages upstream and downstream. So a record in this data set consists of:

- a package name
- a package version
- a package release date

Additionally, revision numbers are tracked for distribution releases. No data is collected about the contents of various releases and as a result no distinction is made between major releases and minor releases and between unstable and stable releases. The scope is narrowed to only version numbers. This data is then gathered from distributions and upstream sources. However, the data collected is clearly a subset of all the open source software data. Distributions were chosen for their unique properties such as release cycle and package management system. This criteria meant that derivative distributions such as KUbuntu were not studied. Upstream packages were largely chosen based on their appearance in many or all of the distributions. If they were available from a source that provided many different packages such as the GNOME, KDE or X11 projects they would also be gathered. Each of these sources were then crawled by a

Python script and the data was stored in a MySQL database. More details are below.

3.1.1 Filename Processing

One technique used often in gathering data both from distributions and upstream involves parsing a package name and version from a tarball or rpm filename. Unfortunately, there is no standard for filename formats. It is common, however, to separate parts of a filename by a single character that is usually a dash. This does not make processing trivial because there is not a separate distinction between name and version. Therefore, the contents and placement of a token must be used to determine whether a token is a part of a package name or version. For example, `gxine-0.5.1.tar.bz2` is version `0.5.1` of `gxine`. In order to process this, originally, all symbol and alpha tokens where part of the name and symbol and numeric tokens were versions. However, more difficult names exist such as `xine-lib-1-beta1.tar.gz`. This breaks the parsing because `beta1` is both numeric and alphabetic. Furthermore, some filenames include name elements both before and after the version. For example, `webmin-1.080-minimal.tar.gz` is version `1.080` of `webmin-minimal`.

The solution is to process the filename tokens left to right. The first will always be part of the name. Furthermore, all tokens up until a token with a numeric character is package name. Once the core version is hit, which has no alpha characters, all tokens with numeric characters are appended to the version and purely alphabetic tokens are part of the package name. There are exceptions still however, the tokens `alpha`, `beta` and `BETA` are considered elements of a version string. `src` is dropped all together. Despite this complexity, this parser cannot handle filenames which lack any sort of separator between name and version. Overall, though, it works pretty well based on informal tests.

3.1.2 Distributions

Distributions consist of a number of repositories. Therefore, when gathering data a number of versions and repositories must be crawled. Additionally, most distributions feature separate repositories for different architectures. How these distinctions are embodied in each distribution varies. As a result, a common model for a distribution's structure was created. Most importantly, I created a generic model for distribution branches as a basis for comparison. It consists of these categories of repositories:

past
: old version that is no longer supported

long term support (lts)
: old version that is supported despite it not being the newest

current
: newest stable version

future
: future stable version currently under development

experimental
: initial version which will form the next future version

Not all distributions have all of these categories but most have at least current and future versions. Below are the details for each distribution about how the data is gathered and classified.

Arch Arch is a smaller distribution which does rolling releases using its Pacman package manager to distribute binary packages. Four different repositories are crawled. The core and extra repositories are grouped under the current branch. The testing repository is considered the future versions and lastly the community repository is the experimental branch.

Debian Debian is one of the oldest active Linux distributions. It has its own file format (.deb) and package manager that distributes its binary packages. It has both past, lts, current, future and experimental versions. These are statically defined in the crawl based on the codenames of each release. Development is done in a fixed cycle that lasts approximately 18 months.

Fedora Fedora is a relatively recent community spinoff of RedHat. It uses the RPM package manager to manage binary packages. It also has a single XML file that stores the metadata for all of these packages in each repository. These XML files are parsed for the desired information. The most recent version is considered current and the development version is considered future.

Gentoo Gentoo is a distribution centered around the Portage package management system. It manages the packages using the sources and then compiles them on the user's machine. It also features a rolling release cycle. Data is gathered using an rsync'd version of the portage tree which features single files that store individual package metadata. Gentoo uses keywords to defined the stability of each package. Gentoo's three levels of stability are: masked, unstable and stable on a per architecture basis. In gathering the data, masked packages are ignored, unstable packages are the future version and stable packages are the current version.

OpenSUSE Similar to Fedora, OpenSUSE is the community version of SUSE. It uses rpms along with its own YaST manager. It is developed on a fixed release cycle that was recently expanded from a six month cycle to an eight month cycle. Novell, the company behind SUSE and OpenSUSE, runs a build service that creates packages for SUSE, OpenSUSE and other distributions such as Ubuntu. The data is gathered in a rudimentary form from directories which contain rpms. The filenames are then parsed with custom code to get the package name, version and revision. The release date is assumed to be the file's modification time.

Sabayon Sabayon is a young binary derivative of Gentoo. It utilizes Gentoo's ebuilds to create binary packages through its Entropy package manager. It is a hybrid rolling and fixed release in that packages are updated in a rolling fashion but the system as a whole, because of its dependency on Entropy, is released periodically. The crawl determines the current and future versions based on hard coded data about the periodic releases. With a fixed release no distinction is made between stable and unstable packages.

Slackware Slackware is also one of the oldest active distributions. Unlike the other distributions studied, it is developed by a single person, Patrick Volkerding. The package management system is very basic. It is based on tarballed binary files per package. It has a current version which is the highest numbered version and future version labeled current. There are also four component repositories that are traversed: *extra*, *pasture*, *patches*, and *slackware*. The data comes from the PACKAGES files in each of these repositories.

Ubuntu Ubuntu is one of the most popular distributions today. It features a six month fixed release cycle with a metacycle of releasing a long-term-support (lts) version every two years or four releases. Ubuntu is technically a derivative of Debian and utilizes the .deb package manager to distribute binary packages. However, there is a large community surrounding Ubuntu which leads to a distinction between it and Debian. At any given point, there is a lts release, a stable release and a development release which correspond to my lts, current, and future versions in the data.

3.1.3 Packages

In the beginning of the research, each script gathered data about a single package. It quickly became apparent that this technique was extremely inefficient for gathering a large amount of upstream data. The next approach was to target large projects that produce multiple packages found in distributions such as GNOME, GNU, KDE and X.org. This did not solve the problem of scaling to many individual sources. Therefore, scripts were created that gathered package information for a single style of source based on a few configuration parameters per source. Below are more detailed descriptions of these sources and the crawl techniques used to gather their data.

Single Package Source One of the packages that is crawled by itself is the Linux kernel. It is one of the most important components and is therefore very useful for analysis because each distribution utilizes it. To gather the data, two levels of directories are traversed from http://www.kernel.org/pub/linux/kernel/ and the common filename parser is used to get a package name and version in addition to the file modification time which is stored as the release date. A further stipulation is that the resulting package name must be *linux* to prevent other possible tarballs from being interpreted as packages.

Multiple Package Source Projects like Gnome, GNU, KDE and X.org all host a variety of software packages. Furthermore, they each have one directory under which all tarball releases can be found. Therefore, a traversal of the directory structure can provide data for many releases. Some of the crawl scripts limit the depth while others do a full recursive exploration. It is common in these scripts to add limitations on acceptable package names and versions. The scripts also limit which directories are traversed.

SourceForge SourceForge is a commercial site that makes hosting for open source projects available for free. As a result, it hosts many of the programs included in Linux distributions. However, they are hosted through a mirroring system that does not allow a simple traversal to find all of the packages. So, the SourceForge RSS feed system is used to gather information about the releases. This is done through a feed that provides the full filenames of released files. The released files for a specific project are processed by the filename parser and those with the expected package names are kept. This filtering system prevents odd package names from being added into the database that originate from the project or a misinterpretation of the filename.

Single Directories All of these crawl techniques do not handle packages hosted on different independent servers. However, many projects place releases in a single directory on a http or ftp server. Therefore, these can be crawled by a script that lists a single directory, parses the filenames using the generic parser and then accepts package names that are explicitly desired.

3.2 Normalization

After gathering all of the data, there is still data grooming to be done. The largest problem with the data is that names for packages are not standardized. Furthermore, version formatting also varies from distribution to distribution.

3.2.1 Naming

Distributions tend to change the names of packages for a number of reasons. Sometimes, the distribution wants to package parts of an upstream package in multiple packages. The canonical example is splitting packages into its normal package and a developer package which is usually the version plus the suffix *-dev*. Other times, an upstream package is split into its multiple components for individual installation. Another motivation for changing package names is to allow multiple versions of a package to coexist in a distribution at once. An example of this is Ubuntu which splits PHP into three packages: php3, php4, and php5. However, Gentoo does not do this because Portage features a notion of slots which allows for the installation of multiple versions while maintaining the name. Lastly, distributions may change a package's name because the upstream name does not match the standards of the distribution.

Since a package can only be used for comparison if it's in all of the distributions in the comparison, package names are linked together in a table that functions like an up-tree to associate names into a larger set. This table is currently populated by hand. In the future, it could be populated in an automated way by comparing package names, version numbers or additional information such as homepage. During analysis, all of the packages names will be used to determine the versions and their release dates in the particular scope.

3.2.2 Version Ordering

One major tool of analysis which also normalizes the data is the process used to understand the relationship between versions. This is primarily focused on ordering the versions so that one can determine if one release is obsoleted by another. Originally, versions were ordered purely by release date. That meant that anything older than the newest release was obsolete. Furthermore, in this system of analysis, versions in distributions that did not match up exactly with an upstream version could not be reliably analyzed. This is due to issues with matching prefixes. This is also an issue that distributions are faced with but their solution is usually to sort version strings alphabetically. This breaks down with different length numbers. For example, 23 would be sorted after 123 despite the numeric ordering. As a result, I developed a more elaborate analysis technique.

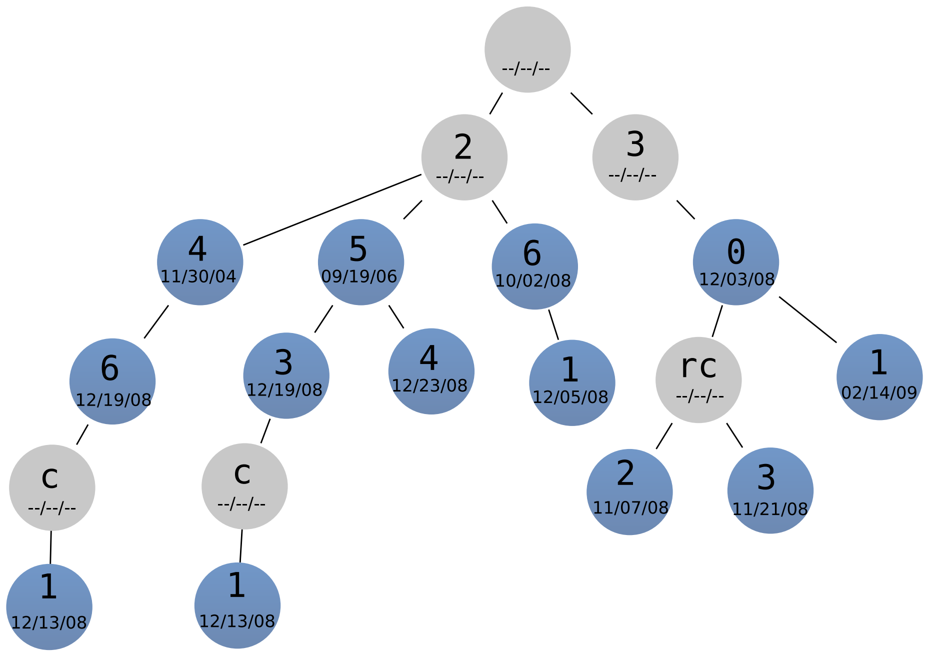


Figure 1: Partial Python version tree.

This technique uses a tree structure to analyze versions. The first step to constructing the tree is splitting the version string into tokens. The tokenizing process splits the string between numbers and letters and on symbols that are dropped. For example, `2.6.29_beta1` is the tokens `2`, `6`, `29`, `beta` and `1`. Once tokenized, the tree is constructed by placing the higher order tokens towards the empty root and lower order tokens towards the leaves. Child nodes are then sorted by the earliest release date with that token. This solves the obsoletion problem because even if a subrelease is released after a newer major release the subrelease will be placed to the left of the new major release. Figure 1 shows the resulting tree for a subset of the Python releases. In the figure notice that `2.6.2` is still left of all `3.x` version despite it being released afterward.

3.2.3 Gauging Data Quality

Another one of the challenges while doing analysis on this data is determining what data is good. In particular, choosing packages over which to analyze distributions is a difficult task because some packages, with a specific name, exist only in a subset of distributions. Furthermore, those that do exist in all distributions may or may not have reliable upstream data. One technique to compensate for not having an upstream source is to approximate the package's history by taking the earliest date of each release in the repository independent of the source. However, this turns out to be quite unreliable because it leads to package release reordering and adds potentially bizarre version strings from individual distributions to the package history. Therefore, only packages with real upstream data are considered. So, the collection of good packages consists of packages which have two properties: *1*) an upstream source *2*) exist in all distributions. The second property is tough to achieve due to the naming issue described earlier.

To settle on a list of interesting packages for the analysis below, I started with the core packages of Slackware, because it tends to be one of the limiting factors for the second property above. Then, I found to what degree each package fit the above properties. If it was only missing an upstream source or existed in all but one or two distributions I fixed the data and added it to the good set of packages. Fixing the data involves either adding another upstream source or linking one package name to another. This resulted in the list of packages in Figure 9 in the appendix.

4 Results

Since October 2008 a lot of data has been collected. As of May 16, 2009, 2,300 upstream packages have been gathered, and nine distributions with 828,137 releases from 72,683 packages have been collected. It is important to note that these numbers include a significant amount of error due to the normalization challenges discussed previously. However, this is still a massive amount of data, but is only a small fraction of the full open source data space. This data is then used for two main purposes. Analysis is done over the aggregation of the data and the individual pieces of the data are made available on the website.

4.1 Analysis

Below, aggregate analysis is done over the different distributions with respect to a limited set of 137 good packages, unless otherwise noted. This set of packages is referred to as the good or groomed set of packages and was formed based on the metrics outlined above in section 3.2.3. The full list is in Figure 9 in the appendix. This data aggregation starts on October 17th, 2008 when data collection began and ends around May 29th, 2009 when the analysis was done.

4.1.1 Upstream Releases Count

One of the most basic aggregations of the data collected is to simply observe the number of upstream releases per day. This analysis puts a finger to the collective heartbeat of upstream. Below in Figure 2 observe the number of releases per day for the approximately 2,300 upstream packages currently tracked.

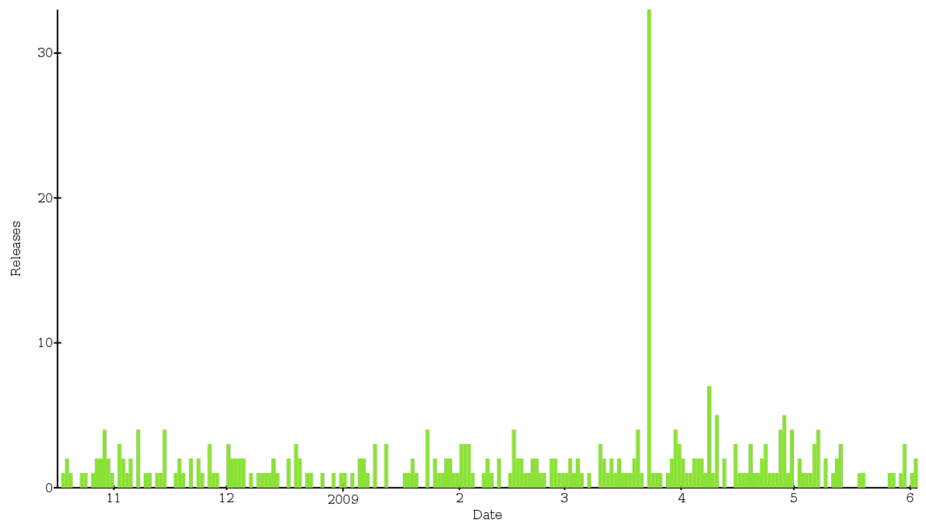


Figure 2: Upstream releases per day since October 17th, 2008.

Figure 2 has two interesting properties. First, the large spikes in releases in a single day show the influence of larger projects such as GNOME and KDE. This influence by the large projects could provide justification for synchronizing to their release schedule. This adjustment is further justified by the lack of any other aggregate cycles. However, the second interesting bit is the reduction of releases directly after Christmas. This is an example of a common release pattern for all upstream sources.

Similar analysis can be done over more limited sets of packages. Below, in figure 3, is the histogram of releases for the good set of packages. This graph has two features to note. First, there is an anomaly in late March where the number of releases for a single day was over thirty. This is an example of the potential for bad data even in the good set of packages. Upon further investigation of this spike, I found that it is 32 separate `openvpn` releases that day. This was caused by many files on their server having the same modification date and time. Second, there is a release lull at the end of the graph in the latter half of May. This lull will propagate into some of the further analysis below.

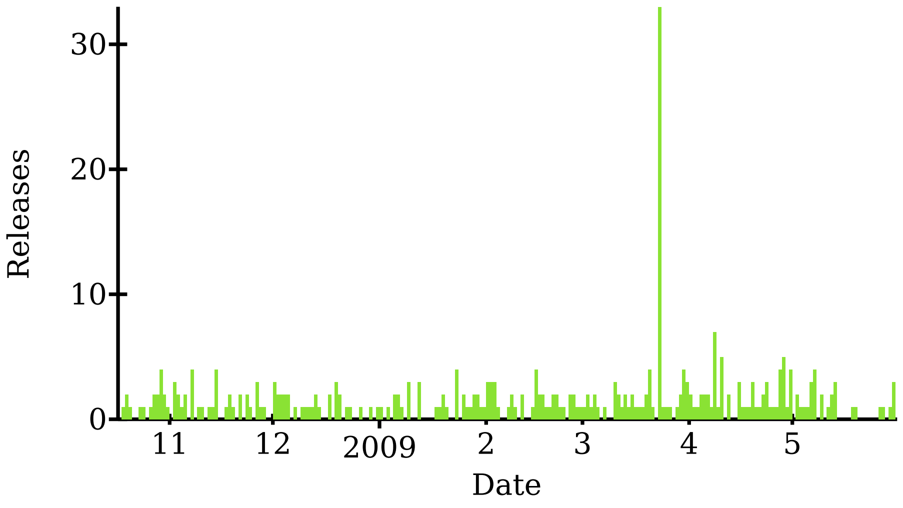


Figure 3: Groomed upstream releases per day since October 17th, 2008.

4.1.2 Obsolete Packages

The next step to analysis is relating the upstream releases to downstream releases. This is done in its simplest form by tracking the fraction of obsolete packages over time. This can also be thought of as the percentage of packages with obsolete versions in a distribution out of the total set of good packages. Figure 4 shows the analysis of the different distributions over time.

In Figure 4 one crucial observation can be made. In this, the difference between fixed-release cycle distributions and rolling release distributions is apparent. These two types of release cycles are distinguished by their different shapes over time. Rolling release distributions meander around some sort of average obsoletion level over time. Conversely, fixed release cycle distributions feature drastic drops on release days but then increase until the next release. Arch is an anomaly because it sustains about 20 percent obsolete while all the others are within 40 to 60 percent. This may be a reflection of Arch's small

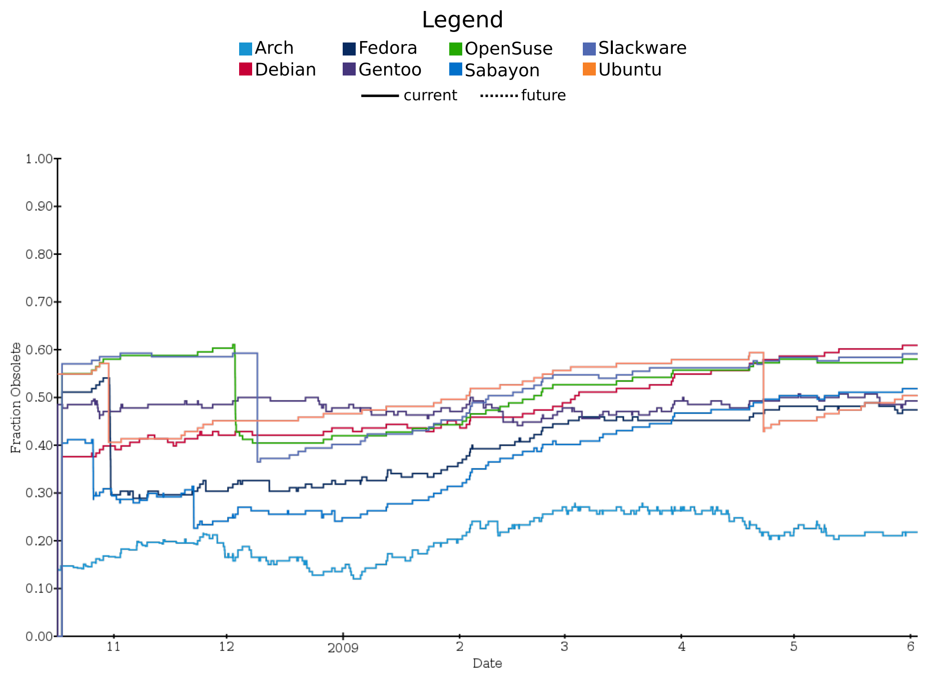


Figure 4: Percent of old packages out of standard set.

repository size, which allows for focused development, and their small, technical community, which does not rely heavily on system stability.

4.1.3 Versions Behind

One missing element from the previous form of analysis is the notion of how obsolete a distribution's version is. In the graph below, this is taken into account by counting the number of newer releases at any point in time for each good package and averaging this count together. The result is in Figure 5.

Figure 5 is fascinating for a number of reasons. First, it provides a much clearer distinction between each distribution. It shows that Debian and Slackware packages are, on average, more obsolete than other distributions. This is as expected because Debian focuses on extreme stability and Slackware is maintained by a single person. The graph also shows a plateau in the latter half of May that corresponds to the reduced number of releases in May pointed out earlier in section 4.1.1. Furthermore, the shapes of the fixed release distributions are surprising similar between February and May when none of them released a new version. Lastly, this fixed release cycle shape contrasts the rolling release distribution's constant incremental improvement.

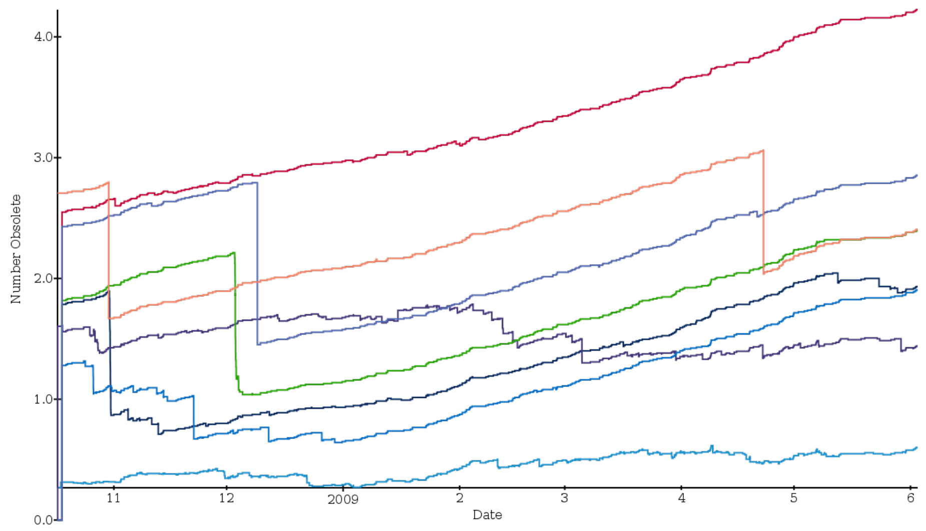


Figure 5: Average number of newer packages over time.

4.1.4 Lag

Another way of adding resolution to the obsoletion metric is by tracking the length of time a newer version has been available. This metric was derived from a desire to gauge the time between upstream and a particular distribution. The result was a notion of lag which is rooted in the idea that no lag occurs until a new upstream version is released. At that point, as time goes by, the lag increases until downstream releases the latest version which leads to a lag of zero. In practice, this is the time between the date of analysis and the oldest new release. That is, given a downstream release how long has passed since a newer version was released upstream. If multiple newer versions exist upstream the time is based on the oldest of those. Therefore, the graph will not be affected by subsequent upstream releases. However, distributions may partially catch up by releasing any newer release before the newest. On the next page, in Figure 6, this analysis is done for all current distributions over the groomed set of packages.

Figure 6 features similar trends found in Figure 5. In particular, the contrast between fixed release cycle distributions and rolling release distributions still exists. However, this lag graph further does not reflect upstream activity to the extent that the previous graph does. As a result, the package release lull in May is not visible. There are also some differences in ordering of the distributions. In terms of lag, Slackware is the highest but in terms of the number of newer releases, Debian is highest. Despite this inversion, the majority of relationships hold true. For example, Arch is still the minimum in lag.

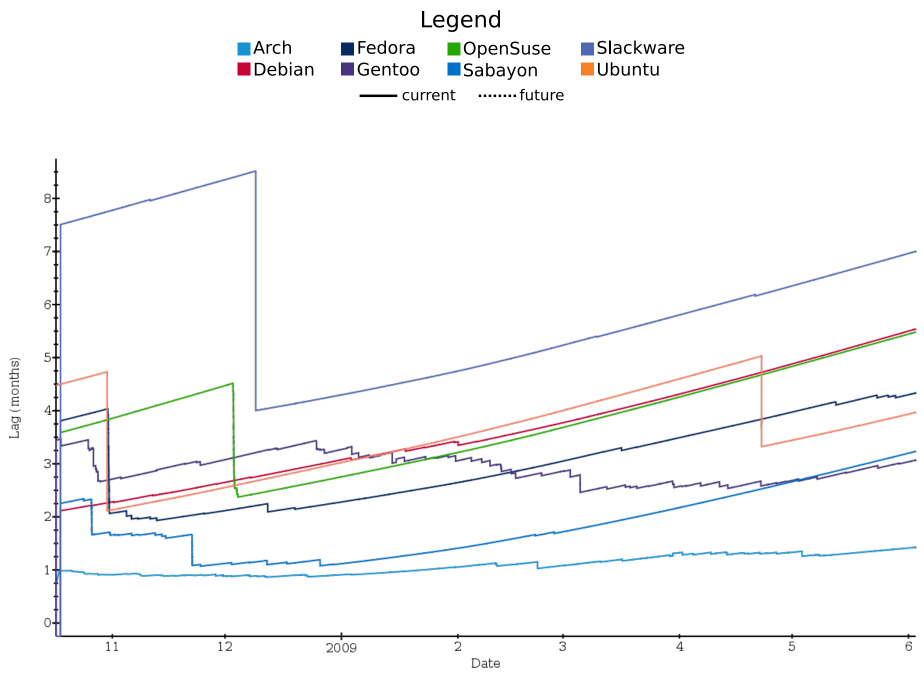


Figure 6: Average lag of current distributions.

4.1.5 Fixed Versus Rolling Release Cycles

One of the biggest benefits of this research is the quantification of the relationship between upstream and downstream. When visualizing this data the effects of the different choices of distributions become apparent. One example of these effects is visible when Gentoo, a rolling release source distribution, is contrasted with Ubuntu, a fixed six month release binary distribution. Below, in Figure 7 both the current and future versions are graphed for Gentoo and Ubuntu.

Adding the future versions of each further emphasizes the vastly different development techniques. When creating a new future version, Ubuntu focuses updating packages into the first month or so of release development. Conversely, Gentoo is constantly adding to the future version just like they are constantly adding to the current version. Most interestingly, the current versions of each average out to about the same lag over the entire time. However, Gentoo's future version is much less laggy than Ubuntu's future version. This is both an effect of the differing release cycle choices and the different package management styles. Source package management greatly eases dependency issues because code is configured and compiled specifically for each user's setup. With binary package management, the dependencies are much more rigid because all packages are compiled to specific versions. Therefore, if package X is compiled to library Y version 1, library Y cannot be updated without updating package X also. As

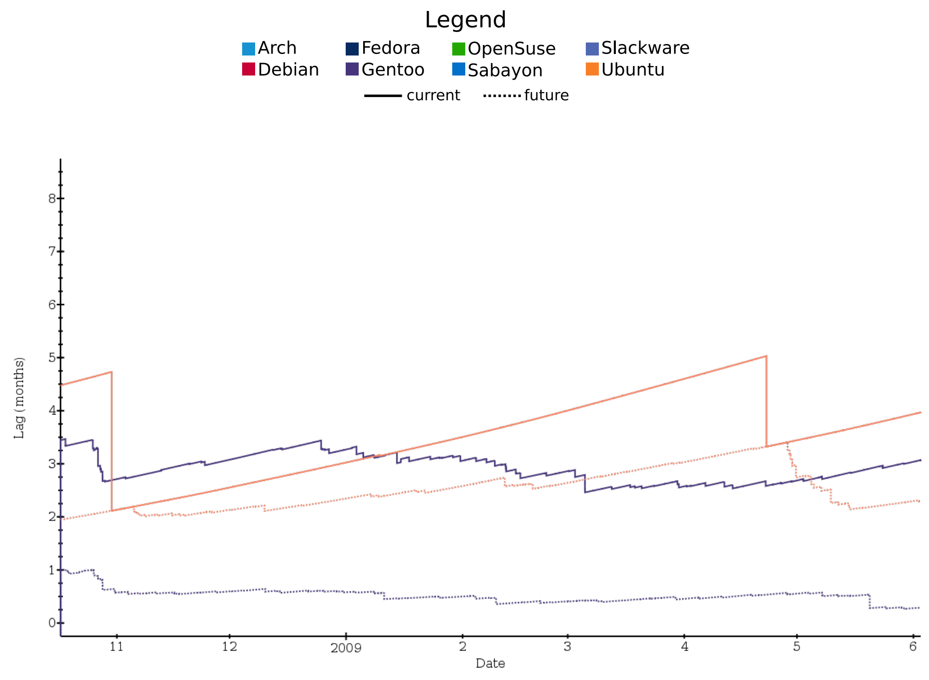


Figure 7: The lag of Ubuntu's and Gentoo's current and future versions over time.

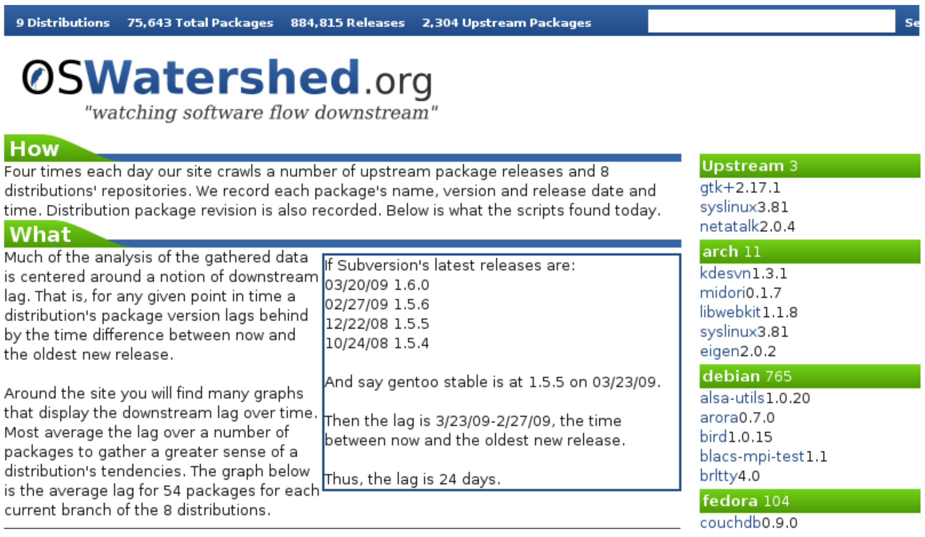
a result, binary distributions must make large changes once in the beginning of development to avoid these maintenance issues.

4.2 Website

This research would not have been possible without the openness of the open source community. Therefore, in the spirit of openness, much of the research data is publicly available on the research website, *oswatershed.org*. Hopefully, the availability of this data will further promote the study of open source development. Additionally, this data can aid the development process directly. It can help both upstream and downstream developers. The site is split into two sections, packages and distributions.

4.2.1 Packages

The packages section aims to provide two things. First, the package section provides current information about what version of a particular package is available in a particular branch of a distribution. This enables package developers to put effort into promoting the use of the new version into areas where it doesn't exist. Second, the page provides information about the latest release of the software and will, in the future, provide an analysis of a package's release tendencies such


9 Distributions 75,643 Total Packages 884,815 Releases 2,304 Upstream Packages
Se
OSWatershed.org
"watching software flow downstream"

How

Four times each day our site crawls a number of upstream package releases and 8 distributions' repositories. We record each package's name, version and release date and time. Distribution package revision is also recorded. Below is what the scripts found today.

What

Much of the analysis of the gathered data is centered around a notion of downstream lag. That is, for any given point in time a distribution's package version lags behind by the time difference between now and the oldest new release.

Around the site you will find many graphs that display the downstream lag over time. Most average the lag over a number of packages to gather a greater sense of a distribution's tendencies. The graph below is the average lag for 54 packages for each current branch of the 8 distributions.

If Subversion's latest releases are:
03/20/09 1.6.0
02/27/09 1.5.6
12/22/08 1.5.5
10/24/08 1.5.4

And say gentoo stable is at 1.5.5 on 03/23/09.

Then the lag is 3/23/09-2/27/09, the time between now and the oldest new release.

Thus, the lag is 24 days.

Upstream 3
gtk+2.17.1
syslinux3.81
netatalk2.0.4

arch 11
kdesvn1.3.1
midori0.1.7
libwebkit1.1.8
syslinux3.81
eigen2.0.2

debian 765
alsa-utils1.0.20
arora0.7.0
bird1.0.15
blacs-mpi-test1.1
brltty4.0

fedora 104
couchdb0.9.0

Figure 8: OSWatershed.org's homepage. Much of the data gathered is made available, in aggregation, here.

as how far apart particular releases occur and what time of year. This second analysis is aimed at helping distributions know what the latest version of the software available is. It does not, however, provide any information about where to acquire a package or its dependencies like Freshmeat does. The goal of this website is not to duplicate data efforts.

4.2.2 Distributions

The distribution section does not exist yet. However, it will feature the graphs presented earlier for different package sets such as GNOME, KDE, Office Apps and Developer Apps. This information will be aimed at assisting people in choosing different distributions by showing the strengths and weaknesses of each distribution. It can also be used in finer-grained comparisons between distributions similar to those done in the previous analysis section.

5 Conclusion

This research is just beginning to study the open source ecosystem. It takes the first steps toward understanding the open source development process, and the development process of larger systems of software more broadly. Previously, these notions of responsive and unresponsive distributions with respect to upstream were only based upon intuition and anecdotes. Now, this research has begun to nail down these amorphous ideas for greater study. Furthermore, the data gathered has been exposed back to the community as a tool for improving the process of creating large software systems out of small and large independent pieces.

It is clear now that popular distributions feature only subtle distinctions about their relationship with upstream. Despite the large differences in the development process between Gentoo and Ubuntu, they both manage to maintain approximately the same level of lag in their current stable releases. Other distributions vary from these somewhat but no one attribute of distributions leads to a significant advantage. Hopefully, the data culled by this research will benefit all distributions such that all are less obsolete over time.

6 Future Work

As hinted at previously there is much more work to be done. As each day passes more and more data is collected automatically. Furthermore, more crawl scripts for downstream and upstream will be added with the goal of logging a larger portion of the open source ecology. Also, further normalization work will occur to increase the size of the good package subset used for analysis. And, as this set of good packages grows, smaller groups of related packages can be used to analyze release relationships pertaining to particular classes of packages. All of this new and better data will improve the quality of the existing analysis techniques.

Much of the analysis above is done with respect to the current version of each distribution. Missing from this analysis is discussion of the subjective notion of when a distribution is stable. Therefore, Arch may have a large advantage over other distributions like Ubuntu in that the Arch developers expect a different level of stability than the Ubuntu developers. To remedy this, some notion of target stability should be introduced. One solution would be to have a number of users rate the perceived stability of the distribution. Integrating a notion of stability into the analysis will strengthen the comparison between distributions.

Much of the research so far has studied the data from a distribution's perspective. However, there is an equally important package perspective. Instead of learning which distributions are freshest, one could find out what packages are included in distributions the fastest. This perspective can provide package developers similar insights to those that distribution developers can already gather from the analysis. Unlike the distribution analysis, however, this package based analysis depends on a large number of upstream and downstream releases to show particular insight. On the website, there is the beginnings of this analysis but some distributions only feature one or two downstream releases which does not provide a thorough portrayal of the package developer's efforts to promote adoption of new versions of their software.

Overall, much more work will be done in the next few months in preparation for the Open Source Convention (OSCON) in July. This will include an increase in data gathering and data normalization at its center. But, it will also include exposing more data through the website and other forms of analysis. Despite all of this work, there will still be more work to do because the scope of the open source software ecosystem is constantly growing. Perhaps by gaining the support of the large open source community, the majority of the available data can be collected, normalized and analyzed.

7 Acknowledgments

First, I would like to thank David Notkin for supporting me in this original research. I would also like to thank Jason Kivlighn for being the endless sounding board for all of my ideas. Many thanks to the open source community at large which has allowed me to consider computers, copyright and community from whole new perspectives. Lastly, I'd like to thank my parents for instilling a fierce desire for knowledge in me while allowing me to explore areas of knowledge sometimes tangent to school and unknown to them.

- a2ps
- acpid
- alsa-utils
- aspell
- aspell-en
- atk
- audacious
- autoconf
- automake
- bash
- bc
- binutils
- bison
- blackbox
- bridge-utils
- cairo
- ccache
- cdrdao
- clisp
- compiz
- coreutils
- cpio
- cscope
- curl
- cvs
- desktop-file-utils
- diffstat
- dirmngr
- dosfstools
- e2fsprogs
- ed
- emacs
- enscript
- esound
- ethtool
- expat
- file
- findutils
- flac
- flex
- fontconfig
- gawk
- gcc
- gdb
- gettext
- gimp
- git
- gkrellm
- gnome-icon-theme
- gnuchess
- gnuplot
- gperf
- gqview
- grep
- groff
- gucharmap
- gv
- gzip
- hal
- hdparm
- hicolor-icon-theme
- hplip
- icon-naming-utils
- imagemagick
- indent
- intltool
- iptables
- iptraf
- joe
- kbd
- less
- lftp
- libpng
- libtool
- libxml2
- m4
- mailx
- make
- mc
- mdadm
- mercurial
- module-init-tools
- mtr
- mtx
- mutt
- nano
- nasm
- netpbm
- nmap
- ntfsprogs
- obexftp
- openssh
- openssl
- openvpn
- pan
- parted
- patch
- pciutils
- pcmciautils
- perl
- pidgin
- pm-utils
- ppp
- python
- rcs
- rdesktop
- rsync
- samba
- scim
- scim-m17n
- screen
- sdparm
- sed
- shared-mime-info
- sharutils
- slrn
- smartmontools
- sox
- strace
- subversion
- sudo
- sysfsutils
- syslinux
- sysvinit
- tango-icon-theme
- tar
- tcpdump
- tcsh
- texinfo
- tin
- udev
- usbutils
- vorbis-tools
- wget
- xchat
- xcompmgr
- xterm

Figure 9: List of packages used in analysis.