Master Thesis in Software Engineering and Management

Measuring the Usage of Open Source Software

Mathias Bronner

Göteborg, Sweden 2007

IT University of Göteborg

CHALMERS | GÖTEBORG UNIVERSITY

Department of Applied Information Technology

REPORT NO. 2007/66

Measuring the Usage of Open Source Software

MATHIAS BRONNER

Department of Some Subject or Applied Information Technology
IT UNIVERSITY OF GÖTEBORG
GÖTEBORG UNIVERSITY AND CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

Measuring the Usage of Open Source Software
MATHIAS BRONNER

Report no 2007:66
ISSN: 1651-4769
Department of Applied Information Technnology
IT University of Göteborg
Göteborg University and Chalmers University of Technology
P O Box 8718
SE – 402 75 Göteborg
Sweden
Telephone + 46 (0)31-772 4895

[Printer's name]
Göteborg, Sweden 2007

Measuring the Usage of Open Source Software

MATHIAS BRONNER

Department of Software Engineering and Management

IT University of Göteborg

Göteborg University and Chalmers University of Technology

SUMMARY

The rapid pace of development of software products, particular in Open Source Software (OSS), generates an increasing number of product releases. The success of such product releases relies on several factors, e.g. strategic timing, functionality, usability, market focus. In the end, what can often be measured is the number of users downloading the software product.

Measuring the usage of OSS, enables mapping the exact functionality exploited by users, thereby assisting developers to improve the product. Tailoring of the product based on user statistics, could optimize the product to its market. One option to optimize the product is removing sparsely used functionality and focusing on the elements central to the user. Following the same line of thought, potential investors of OSS can estimate the potential market of a software product by receiving statistical information on the usage of the software.

For acquiring feedback on the usage of an OSS product, this Master Thesis Study provides a proposal for a system measuring the usage of OSS. The proposal includes an investigation of usage in both traditional software -and open source development, breaking down the findings to measurable points, analyzing the most common interests of open source project stakeholders, and combining this into usage metrics specific for the proposed platform. In addition, a system infrastructure is created containing requirements, architecture and thorough analysis of important system qualities are performed.

The result is a proposal for a platform, capable of measuring the end users usage of OSS. This provides open source project stakeholders with statistics. The interests of open source project stakeholders will be investigated to determine appropriate measurement factors from both commercial software and OSS.

Keywords: open source, measure, usage, statistic.

Preface

This report presents the results of a master thesis study, conducted from the 19th of January 2007 to the 25th of May 2007 at the IT University of Gothenburg. The purpose of the study is to propose a method of measuring the usage of open source software.

This report is the result of a literature study partially accomplished with a collaborative effort involving a fellow Master student, Maria Josefsson, who shared the same research area although with a different focus. The collaboration resulted in a background chapter – *Open Source Software*, presenting a general picture of open source software including an economical perspective.

I would like to thank my supervisor Thomas Lundqvist and mentor Friedrich Bosch for their support and advice during this study. I would also like to thank Maria Josefsson, who I worked with on the literature study and who incorporated some questions in her interviews on my behalf. Furthermore I would like to thank the respondents for providing me with valuable information.

Mathias Bronner, Master Student
IT University of Gothenburg
May 2007
mathiasbronner@gmail.com

Table of Content

1 Introduction

This Master Thesis presents a new approach to measuring the usage of Open Source Software (OSS). At present time no such solution exists covering the most popular OSS platforms. Such a platform could provide information about the user-interaction of a specific OSS application, e.g. number of users, user activity or functionality used.

The rapid pace of development of software products, particular in OSS, generates an increasing number of product releases [3, 16]. The success of such product releases relies on several factors, e.g. strategic timing, functionality, usability, market focus. In the end, what is often measured is the number of users downloading the software product. Although what can be of interest, to both the developing community and potential investors, is the actual usage of the product.

The usage of a product can refer to the amount of time and functionality used, which can be measured to provide valuable feedback. For interested investors, private or venture capitalists, the feedback from an early release of the product can be utilized to foresee a potential product success. This enables them to make decisions based on statistics. On the other hand, the feedback used by the development community to monitor the functionality use, gives them a chance to improve, add or remove functionality for creating a better product.

The research question is the following:

How to measure usage of open source software?

The main research question leads to the following two questions:

1. What factors are of importance when measuring usage of open source software?
2. Is it possible to create general frameworks and development tools to support measurements for open-source applications?

The result of this Thesis Project will be a proposal of a system which measures the usage of OSS, including a definition of measurement factors with interest to the context, requirements and system architecture.

Currently one software solution exists to measure the usage of OSS, being the Debian Popularity Contest named after the Open Source Debian Linux distribution [21]. The purpose of Debian Popularity Contest is to measure the usage of programs or packages installed on computers running the Debian operating system. The Debian Popularity Contest, though functional, is limited to the Debian Linux distribution platforms, and is not thoroughly or systematically documented. This hinders further research to be performed upon this solution [21].

The traditional approach of software development referred to as *closed source*, is based on the assumption that software development is a specialized process. This process is best handled by a localized team of skilled developers and a manager [3]. This development results in the form of periodical releases. Open source is on the other hand based on inter-geographical collaboration between developers and users,

continuous improvement and frequent releases, and conformance to open standards using open source licenses [3].

Unlike the traditional approach of software development, open source users have free access to the source code. This enables users to modify the code and correct possible errors which might include porting the software to another hardware, or software platform. As a result the users can create add-ons to solve perceived problems or just use the software as it is [2]. In addition, unlike the traditional software development approach no broader OSS platform is available for measuring the usage. For traditional software applications different levels of usage measurement platforms and tools are available, providing concrete measurable information to the stakeholders of the software [27, 28]. This study will look at usage metrics, currently used for measuring traditional and OSS applications, and apply the knowledge to the proposed measurement system.

The development of a system for measuring usage could help to establish an open standard for collecting information of importance to the stakeholders. Following an open industry standard gives a certain degree of freedom, while easing the decision taking process. A good method is to offer a software solution in a number of proven industry-standard configurations, and let the users choose between them.

There is an interest from people in the open source community to acquire usage statistics. From an interview with representative for KDE Cornelius Schumacher, a genuine interest for usage measurement was expressed [13]. It was stated that currently, information regarding the number of users and other types of usage statistics, was unavailable to KDE. KDE is searching for methods or tools that can provide this information.

2 Methodology

This section contains the methodology used for conducting the Master Thesis study, as well as an explanation of the work process and motivation behind the steps taken to achieve the result. The reason for defining the work process is to make the study repeatable, thus reaching the similar result if the study was to be performed again.

The methodology used for the Master Thesis study is a Design Research approach. This method consists of the construction and evaluation of technological artifacts in relation to their area of use, and the proposal of new theories if seemed necessary.

The aim when conducting research is to make a contribution to academia as to practice. This implies that the research conducted should both add to the existing base of theory in order to make a contribution [11], and assist in solving current or expected practical problems [14].

The use of the Design Research approach is directly associated with the expected result, which is to create a platform for measuring OSS. The result of the study has to be evaluated both to ensure the validity and reliability of the study, and to determine to which extend the result can be used as a base for further research. For this purpose the result will be evaluated against the existing Debian Popularity Contest as a frame of reference.

As the initial step an adequate background on OSS will be consolidated, providing a foundation to the study. The background will contain the central idea of OSS and motivation behind it, as well as the economy and interests associated with an open source project - see chapter *3. Open Source Software*.

The next step is to investigate common used platforms supporting OSS, assess their market share and to which extent OSS is used on that platform – see chapter *4. Open Source Platforms*.

In order to consolidate the relevant measurement points for the technological suggestion, a set of requirements will be developed. These requirements will be a result of usage metrics developed from the operationalization of current and potential measurements of usage [13]. The usage metrics providing measurable indicators of usage will add a certain degree of validity to the research – see chapter *5.Usage Metrics*

The elicited requirements are used as a foundation for creating the architecture for the measurement system. The requirements will be refined into a concrete architectural solution, where the non-functional requirements will have a major influence – see chapter *6. System Infrastructure*. The non-functional requirements will be elicited based upon both the functional requirements and a systematically analysis on the importance of the various “ilities” to the proposed system [12]. The break down of the non-functional requirements is done to ensure technical reliability. The result will then facilitate the production of proper counter measures to help the system manage these constrains.

As a final precaution the proposed system, containing requirements and architectural design is to undergo an evaluation. This is to ensure its conformance to its purpose and to verify that the solution is more covering than possible alternatives – see section *7. Evaluation*.

A discussion of the study will elaborate the findings and put the methodology and proposed solution into perspective. Furthermore the discussion will contain subjects mentioned in the report which requires additional attention – see section *8. Discussion*. The conclusion will present a summary of the findings and their implications found throughout the Master Thesis study – see section *9. Conclusion*. Areas of interest to the current study which could enhance or extend it are found in chapter *10. Future Work*. The final chapter, *11. References*, lists the references used in the report and ensures that material and statements in the report are well founded.

3 Open Source Software

This chapter gives an overview of OSS and the economy behind, necessary to comprehend the context in which the proposed measurement system is to operate in. The chapter includes an explanation of the idea driving OSS, the hosting of open source projects, and the contributions they receive by the various project stakeholders.

OSS gives at one end the developers, the possibility to contribute in the form of source code and at the other end the users, the right to use or modify a program and its code base.

One basic requirement of an open source project is the availability of the source code [7]. That implies that open source refers to shared software code with open standards, and the collaboration between software developers and users, to build software [1]. In addition this includes identifying and correcting errors and making improvements to the software [1]. That means that OSS gives individual developers, the possibility to contribute in the form of source code while at the same time giving individual users the right to use or modify a program and its code.

The central element in open source development model is the open and collaborative environment in which software products are created swiftly. The cooperation between both developers and end users in the open source community encourages the building of products with a higher level of quality throughout the product life-cycle [9].

The exact degree of freedom included in the distribution of code, relies on the license type on which the software is released. Many of license types exist and are being constructed to support the interest of the producing community, meaning that OSS and the license applied to them are closely associated [1]. Certain restrictions are imposed on OSS licensing; an OSS license must not discriminate against any type of user group, field or endeavor [16]. In addition, an OSS license must be applied to all parties where the software is distributed, meaning that the Open Source distribution cannot be re-licensed by any user [16].

Most contributors in OS projects are driven by a personal motivation, not directly linked to the size of the salary, but rather factors of a more veiled nature. One explanation behind the personal motivation referred to in Maslow's hierarchy of needs as the category of self-actualization [15]. Other possible factors of motivation are proposed to be learning and skill opportunities, together with social and political factors [16]. Prior studies show that Open Source developers are the most talented and highly motivated software developers [24].

Using OSS in a commercial context has been explored for some time now, and unlike traditional software this revenue is not generated from the actual product. Open source business models in its many shapes tries to overcome the limitations gaining direct revenue by the product, often by using a less strict license or releasing the same software under several licenses. Thus, this implies that business models and license types are closely related and built to suit one another.

3.1 Hosting Open Source

The hosting of open source projects is a necessity and important foundation for the distribution of the code in an open Source project, among members of a community.

SourceForge is one of the world's largest OSS development web sites that hosts and provides services to more than 100 000 projects [9]. SourceForge.net is owned by OSTG (Open Source Technology Group, Inc.) which is a network of technology sites for IT managers and development professionals.

SPI (Software in the Public Interest, Inc.) is a non-profit organization which was founded to help organizations develop and distribute open hardware and software [10]. SPI is like OSTG a non profit organization.

Besides the hosting of the code, hosting organizations like SourceForge [9] and SPI [10] provide a starting open source project with an array of various tools for inter-group communication, version control and a donation system. Minimizing the interdependency between project members by focusing on a small mutual web based platform, enables members to utilize custom tools and techniques, ensuring their freedom of choice.

The result of gathering numerous open source projects at a central place, gives potential investors a possibility to search and contact open source projects developing software of interest. In addition private contributors can be members of the community and search for projects where they can join the development.

3.2 Open Source Economy

In OSS the main source of income is generated from what is around the product, rather from the product itself. Red Hat for example, charges for setting up an Apache Web server, developer training or "24-hour technical support for one year" [25, 29].

The financial model of conventional software development is mistakenly built upon the assumption, that software development is a manufacturing industry and not mainly a service industry [30]. The high purchase price and low service support fee is not correlated with the maintenance cost, which is estimated to be 70-80% of the total software development cost [31, 32]. This is acknowledged in the OSS model, where the purchase price is low and companies can contest on the service to the user, viewing the software as a commodity product where the ingredients are free [33]. Conventional software development companies can gain benefits from embracing open source development and distribution, and by such enhancing their reputation [29]. In open source development the research and development is conducted by the community members, from potentially hundreds of skilled developers, minimizing the costs and facilitating a rapid development and release pace [8].

OSS is often mentioned as being free. "Free" does not refer to the price tag it comes with, but instead to freedom [16, 17]. The founder of the Free Software Foundation (FSF), Richard Stallman, defines free software as the freedom to run, copy, modify

and redistribute the program for any purpose [17, 18]. The policy of FSF is formed by ideological tendencies, much like the Open Source Initiative (OSI), despite the significant differences in the definition of terms like “free” and “open” [22, 23]. According to FSF, the freedom to sell copies is of great importance to this freedom. Selling collections of free software on a media e.g. a CD-ROM will raise funds for free software development. The software that cannot be included in these collections is not free software according to FSF.
Even redistributing a program for a fee is defined as free software, since it again refers to the freedom of use and not the price. As a consequence, selling a free program is accepted and it can help to finance further development of free software [17].

The mechanism which ensures the adherence to the principles of the freedom of software is the General Public License (GPL) and copylefts, where the latter is copyrights with GPL regulations [16, 19]. Basically the GPL is a restriction forcing variations of free software to follow the same license, thus providing a guarantee that resulting software contains the same degree of freedom [16, 19].

3.2.1 Open Source Project Contributions

This section contains an overview of the contributions received by an open source project, which can consist of economical resources or time.

Economical resources are one of the types of resources that can be provided to an open source project. Economical funding is most of the time provided by venture capitalists but also private persons or companies.

The financial contribution of investors is a more unreliable source of funds for most open source projects but it does exist along side funding in the form of donations. Venture capitalists, private investors and companies all have their reasons to invest in certain Open Source projects.

For companies the natural reason for investing in an Open Source development project is that the company uses the software that is developed in that project, and wants to encourage the evolvement of that software. With the economical donation it might be possible to influence the direction of the development, e.g. the development of certain functionalities, covering the needs of the company.

The illustration below (see Figure 1) is based on the personal investigations made during the literature studies made in this area. The figure presents the approximate size of the contributions, money and time, as well as the two main factors that can be invested in an Open Source project. Time refers to the effort of developers, managers and coordination. The size of the arrows in the illustration below represents the approximate amount of the contribution that can be received from the two factors, money and time.

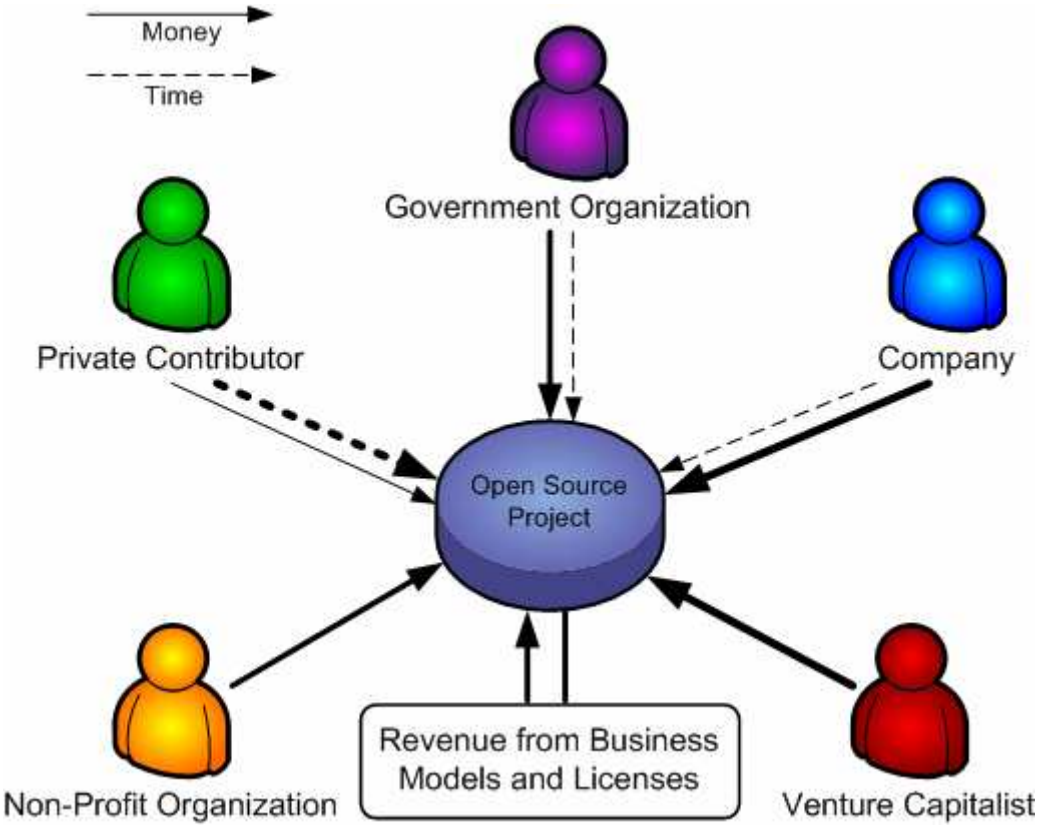

Figure 1 Describes possible sources of contribution to an open source project.

The private contributor can be both a private donator, giving money to an open source project or a developer contributing with time in an open source development project.

The government organization is a user of open source that is willing to contribute with money (and time) in order to achieve specific functionality beneficial to a need, e.g. language specific software support.

Companies can have different interests in open source project, investing both time and money for achieving a long term return of investment. Like government organizations, companies can have the need of specific functionality critical to their business goal.

Venture capitalists have a direct interest in open source projects, due to the possibility to gain attention by the possibility to add a successful open source project to their portfolio. Many Venture capitalists are interested in earning money from the success of the open source project.

Business models and licenses can bring financial support to an open source project. Open source business models and licenses focuses on retrieving money from services around the product, e.g. software support and distribution.

Non-Profit Organizations (NPO) does not generate a profit, but may receive money for the purpose of distribution these to projects which suit the goal of the NPO.

4 Usage metrics

In this chapter currently applied usage metrics are investigated, in order to elicit potential usage metrics which will be a part of the requirements for the usage measurements system.

For describing a conceptual variable such as usage, one has to break them into parts to capture meaning. In order to perform this process the variables will be operationalized into more measurable indicators, thus producing usage metrics [13]. This quantitative approach provides metrics which will be used, in correlation with the goal of the research to elaborate a comprehensive list of measurable points of interest to the research.

For gathering information on end-users interaction, in specific the practical usage of software products, measuring techniques can be used. The usage can be characterized in several factors depending on interest e.g. time, functionality or number of users.

4.1 Existing and potential usage metrics

The existing and potential usage metrics is a survey for the measurement points which could be applied when measuring the usage of software, in the area of traditional and open source development.

The close relation between a central service or product and a price in traditional software development, makes some measurement points specialized for their context and not applicable for contexts where the *cause* and *effect* is more complicated. In OSS development however the product is not so evident, it's rather what lies around the software itself which is in focus. By consequence some measurement points might not be exactly the same.

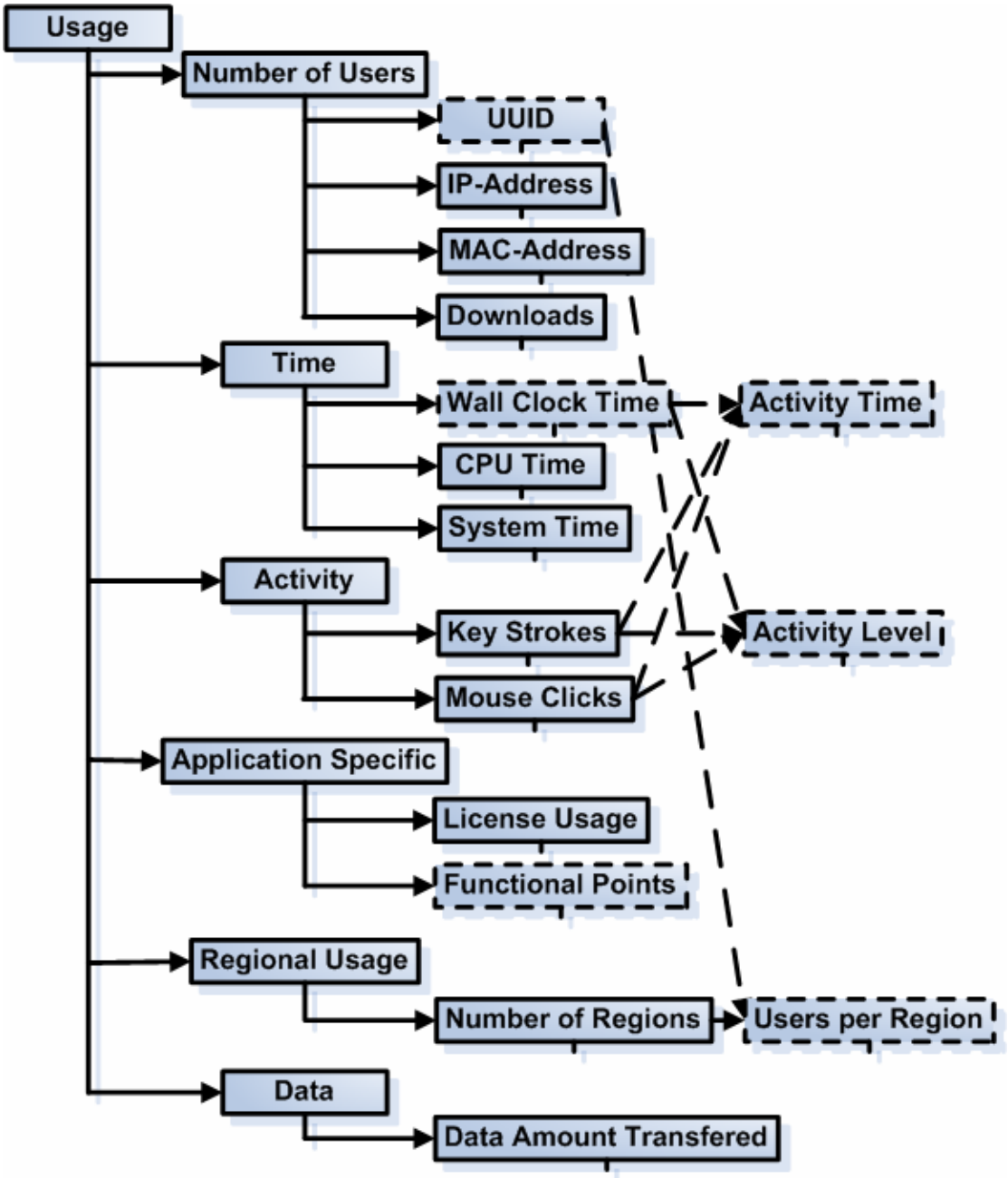

Figure 2. An overview of the existing and potential usage metrics in the area of traditional software development and OSS development found or developed during the literature study. The dotted boxes illustrate potential usage metrics which has been chosen to be a part of the requirements for the usage measurement system. Only two of the chosen usage metrics where found, the UUID and Wall Clock Time, the rest is proposed. The dotted lines show that the resulting usage metric is dependent on more than one usage metric source. Further explanation of the chosen usage metrics can be seen in the following sub-figures.

In traditional software development many of the measurement points are closely tied to a business model, where the measurement is done to formalize a usage which is to be paid for by the user. The telecom industry has had a large influence on how usage is viewed and therefore measured. Measurement factors like *Data Amount Transferred* and *Time* are typical for the telecom industry (see Figure 3), e.g. when using a mobile phone for making a call or using a GPRS service. The same factors are important as well for Internet Service Providers for measuring the end users usage and thereby the size of the bill.

Figure 3. Data usage.

Several measurement points can be specialized into more precisely described measurement point, facilitating the practical set-up of measuring. An example of this is the *Time* usage (see Figure 4) which can be measured using different time scales, and seen from different perspectives. The *Time* can be the *CPU Time*, *System Time* or *Wall Clock Time* usage in a software program, where the latter represents the total execution time of a program [38].

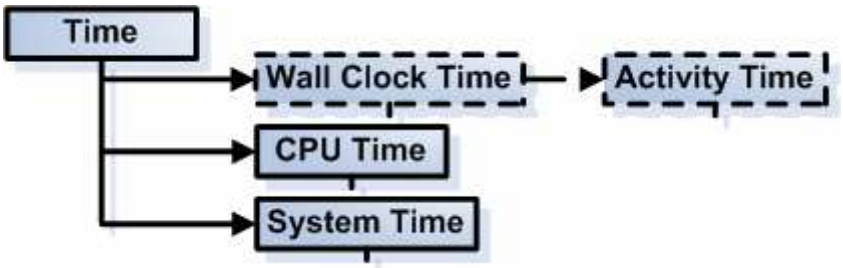

Figure 4. Time usage. *Wall Clock Time* – The accumulated time usage of a program, which can be divided time intervals, e.g. for viewing program time usage over a week. *Activity Time* – The accumulated time usage of a program, where *Key Strokes* or *Mouse Clicks* is inputted by the user. The dependency between *Key Strokes*, *Mouse Clicks* and *Activity Time* is depictured in the overview - Figure 2.

While prior existing measurement points addresses time, e.g. *Time* usage (see Figure 4), the typical viewpoint has been from the computers' logically. The time usage is usually measured in *CPU Time* - the time the program code uses on the CPU, *System Time* – the time used by the program code running kernel code, also referred to as I/O operations, and finally *Wall Clock Time* – which is a combination of the first two and the time spent idle, including waiting for resources [38]. Even though these time usage metrics can be valuable, the first metric *CPU Time* is proportional to the performance of the CPU, *System Time* is dependable on the system configuration and operating system, and *Wall Clock Time* is itself a measurement of how long the program has been running.

The measurement of the end users input, e.g. *Key Strokes* (see Figure 5), are often used to characterize various factors of efficiency. This measurement point could contain the potential to uncover the users behavior at different points in the software program, showing the raw activity of the user.

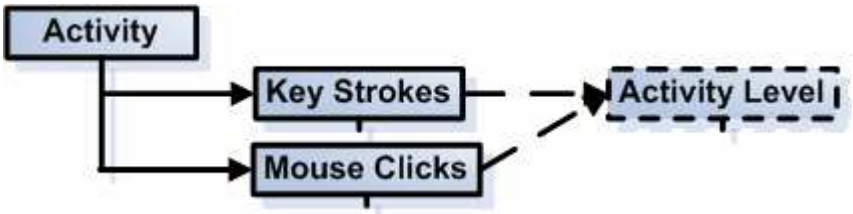

Figure 5. Activity usage. *Activity Level* – A relationship between the accumulated time usage of a program and the time where *key strokes* or *mouse clicks* has been inputted by the user. A higher

level of user input produces a higher activity level. The dependency between *Key Strokes*, *Mouse Clicks* and *Activity Time* is depicted in the overview - Figure 2.

The prior mentioned usage metrics *Data Amount Transferred*, *Time* and *Key Strokes* (see Figure 2) are by themselves measuring basic factors. These alone might not provide any significant information but functions like building stones to acquire the desired information.

Switching the viewpoint from the computers *Time* towards the *Activity Time or Activity Level* (see Figure 4 and Figure 5), where the two latter focus on the users actual time usage, or activity level. The time usage of a program measured in the *Wall Clock Time*, does not show the users usage of a running program, but probably measures a large amount of idle time. Different software application requires different amount of user interaction, while some are merely a background service showing the status of something other applications like a word processor requires a larger amount of user input to produce the intended result. The *Activity Level* would be usage metric to be applied to whole software programs or specified parts in the program which includes a high level of user interaction. For creating a generic usage metric for measuring the *Activity Level* the relationship between *Key Strokes* extended with *Mouse Clicks* and *Time* measured in *Wall Clock Time* (see Figure 4), would provide an indicator for the level of *Activity Level*. The reason for not naming the usage metric *Efficiency* is because the metric does not describe what information the user provides the software program with or relevance of the input. Put in another way, the *Activity Level* metric is too generic to be able to distinguish right input from wrong and hence measuring the efficiency of the user; the metric simply describes the user's activity level. The scaling of the *Activity Level* should be arranged so that many user inputs over a short amount of time should produce a higher number, or level of activity than the inverse.

Most usage metrics in traditional software development can be categorized as a service due to the close association between what is measurement and what the end user is debited for. However what is described as an *Application Specific* usage (see Figure 6) is a software application which provides a service to the end user, where the content of the service is of a more generic nature, e.g. downloading music over the internet. This implies that more basic services measured in factors such as *Data Amount Transferred* and *Time* are not included in *Application Specific* usage.

The context which the measurement is conducted in, either in traditional software or OSS development, as mentioned earlier, influences the measurement of interest. As a result, the measurement of *License Usage* (see Figure 6) has a valid position in the traditional software development context, where in OSS development the same measurement lacks a genuine sense of meaning [35].

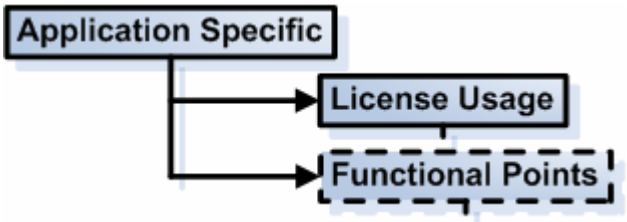

Figure 6. Application Specific usage. *Functional Points* – A measurement counting the number of times a specific functionality of program has been used. The definition of a functionality is

defined by the program being measured. The number of times a functionality has been used can be divided into time frames of e.g. 24 hours or one week.

The behavior of the user includes measuring what kind of information the user chooses to gain access to and not only what is presented to every user that utilizes the software program by default. The information which is of interest to the user in a program depends on many factors, e.g. for what purpose did the user start the application, or what can the program do for the user beside the initial intended use. Nevertheless the user's program experience could be *recorded* to chart which functionality was used or not used, providing valuable feedback to the software developers. The feedback could subsequently be employed to allow software developers to improve the existing software by streamlining the software product depending on market strategy. For the users of the software, this could mean that very sparsely used functionality was removed and the popular parts of the software was extended or improved, thus creating an overall better product.

For measuring the *Functional Points* (see Figure 6) the various functionality should have an indicator attached to them, in order to receive a notification when specific functionality is used. For making the functionality measurable in the first place, the software application vendor should provide the definition of what a block of functionality is. This is needed in order for the measurement of the functionality to have a meaning for the vendor. The granularity of the block of functionality should be course enough, to have a logical reference to a user of the software, but at the same time fine enough to provide the software vendor with feedback to improve the software.

Several software applications currently perform measurements regarding the end users behavior, and with the conformance of the end user, sending this over the internet to a receiver. An example of this is the media player application Winamp, which supports anonymous program usage statistics, consisting of "*how much Winamp is used*" [39]. The content or results of such measurements are in such applications a grey area to the end user, not providing in-depth information regarding what, how, when and why the measurement is conducted.

In both traditional software development and OSS development the number of *Downloads* (see Figure 7) is measured to give an estimate of how many users that are downloading and potentially installing and thus using the software program. Though being a measurement of usage, no clear analogy exists between the number of *Downloads* and users, due to the many uncertainties involved. When the user wishes to download a specific piece of software, he or she presses the link which has a reference to a file containing the installation program, and executes or saves the file locally for the purpose of installation. Several scenarios are possible, the user might not install the software even though downloading it, or the user might install the same software on many machines to save time and not downloading it again. Even if the user installs the problem, the usage of the software is unknown by only looking at the number of *Downloads* making this measurement a rough estimate.

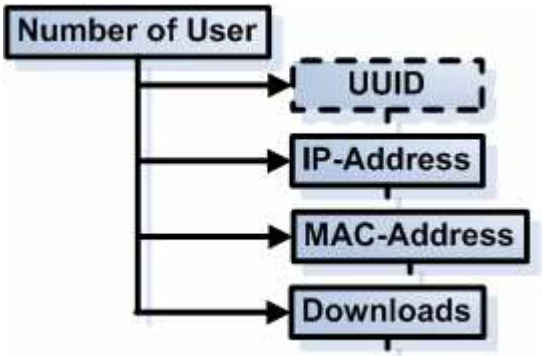

Figure 7. Number of Users. *UUID* – Universally Unique Identifier to distinguish each user for the system and is used to calculate the total number of users of a program.

In a usage measuring system where the usages of software on many computers are monitored, the *Number of Users* retrieved by counting the number of *UUID*s (see Figure 7) is a logical measurement point. Besides being a viable alternative to another measurement point *Downloads* (see Figure 7), it could provide precise and accurate information regarding number of users and possibly their geographical distribution. A number of paths could be presented for how exactly the measurement is performed. One solution is to let every user who is using the usage measuring system have a unique number or name, and simply count the number of unique names. Another way to measure the *Number of Users* in a distributed measurement where the communication is conducted through a network adapter is to employ the in-built *MAC-Address* of the network adapter. An uncertainty when relying on information provided by a network adapter is that the user uses several channels of communication, e.g. a wireless network adapter and a cabled network adapter, resulting in a higher number of users. A third way to identify users of the internet, or in a distributed usage measuring system is their external *IP-Address*. Though reading the *IP-Address* might identify a user, it is not possible to measure the number of users sorely by counting the IP-addresses used in relation with the usage measurement system over time. But the external *IP-Addresses* provided can be mapped to a geographical region, often a country, giving some indication on where the users are. A unique user identification, e.g. a number of a name, combined with the external *IP-Address*, could be used to map the number of users in a specific country or even region, providing government organization with possible valuable feedback.

Figure 8. Regional Usage. *Users per Region* – A number identifying the number of users of a program in a specific region. The dependency between *UUID* and *Regional Usage* is depicted in the overview - Figure 2.

An area when measuring software usage is performance, influencing both availability and usability. In this study the *Time* usage is measured to some degree. While this study acknowledges the importance software performance as a quality, the measuring of performance itself is not the primary focus of this study. The usage is seen from the direct interest of the project stakeholders, shown in the following section, and not from the perspective of the computer. In addition, to conduct covering performance

measurements it would require extensive configuration of the measurement set-up customized for each application.

4.2 Interests and Usage Metrics

This section establishes a connection between the interests of the project stakeholders, the open source context, and the usage metrics used for the usage measurements system.

When looking back in time, complete business models founded upon a direct connection between the actual user usage of software and the price has been suggested without any success [36]. What exists today for non-OSS, is typically a license based agreement where the price of the license gives the users the right to install and use a product how frequently he or she wishes. Most license based software products includes a service, both in the form of documentation, support in case of problems, and future updates to the software. What is sometimes defined is the exact quality of the services included, while software updates might be supplied at a regular basis for several years, but again further development of the product might be discontinued by the supplier. This issue is a legacy from a time where the software was less complicated and in general did not change as much during its lifetime. The dynamic nature of current software makes the expression *product* and the borders around it increasingly unclear and hard to define.

The recognition of the dynamic nature of software is a natural part of OSS development, with its user driven focus, frequent releases and where maintenance is shifted for evolution. For improving existing software a usage measurement system can be utilized, collecting a variety of information, and returning this information to OSS project stakeholders who will be able to benefit from this feedback. The measurement factors suggested in this thesis report which could be used to optimize an OSS product are shown in Figure 2.

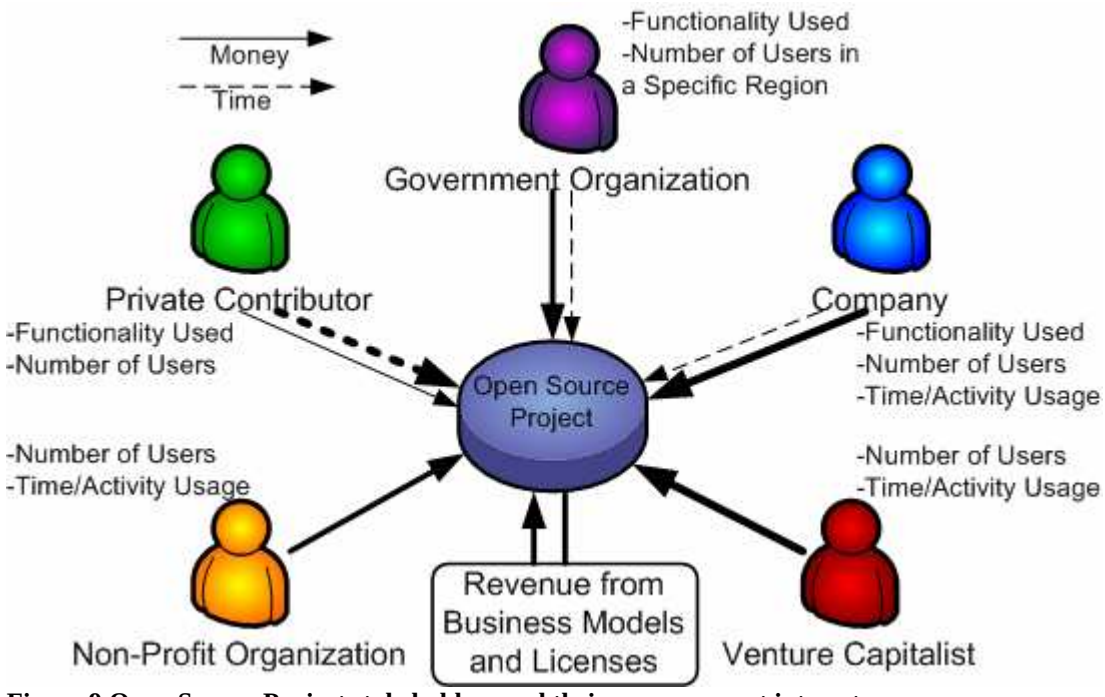

Figure 9 Open Source Project stakeholders and their measurement interests.

Identifying the open source project stakeholders and their interests in terms of measurement points is the first step towards consolidating a set of requirements, for a usage measurement system (see Figure 9). The connection between the stakeholders and their interests where investigated during the literature study.

The *Private Contributor* can be subdivided into a *Private Developer* and a *Private User* (see Figure 9). The *Private Developer* is the main contributor of code (time*)* to the open source project, making this stakeholder an important decision maker. The code is the central artifact of an open source project. The *Private Developer* could be interested in feedback on which parts of the functionality that are used by the *Private User*, which can be used to improve the product. Furthermore the *Number of Users* can be a factor of motivation to the *Private Developer*, realizing that their creation is actually used. The *Private User* uses the OSS, donates money to the project from time to time and could want to see how popular different products are.

The *Company* and *Government Organization* will have similar measurement interests (see Figure 9). The *Number of User/ Number of User in a Specific Region* is as mentioned before a measurement for the popularity of the product. *Companies* and *Government Organizations* can have a strategic dependency on a specific open source product. They might want to extend functionality important to them, by contribution with development of the code base (time) or invest in the project (money). An example is *Government Organizations* wanting an open source product to support a specific language.

Non-Profit Organizations and *Venture Capitalists* are interested in the same measurements point, although their intended agenda can be quite different. While both *Non-Profit Organizations* and *Venture Capitalists* support an open source project financially, the latter does it for the expectations of significant return of investment. Both stakeholders will have an interest in promoting their company names and gain a positive reputation. The *Number of Users* determines if product is used and the

Time/Activity Usage to which extent the product is used, helping to *Non-Profit Organizations* and *Venture Capitalists* to select open source projects to invest in.

The six usage metrics found beneficial for OSS development are: *UUID*, *Users per Region*, *Wall Time Usage*, *Activity Time*, *Activity Level* and *Functional Points* (see Figure 2). These types of usage measurements are distilled from the interests of stakeholders in an open source project found during the literature study (see Figure 9). The aim of conducting these measurements is to improve the software produced in the open source project during its lifespan.

5 System Infrastructure

The System Infrastructure section contains a description of the proposed system, including a list of system requirements, an overview of the system and an explanation of the important qualities of the system. This section is important for understanding how the system operates and the functionality supported.

5.1 Requirements

In this section the requirements of the system are listed to provide an overview of the functionality and the manner in which they collaborate. The requirements are the basis for the usage measurement system, specifying the types of measurements and how the system should support the actual measuring. As the requirements are extracted from the usage metrics, the requirements are a part of a suggested solution for a usage measurement system.

The usage measurement system consists of two main parts. The first is installed on the client machine and the other positioned on a server. The part installed on the client machine, referred to as the *client program*, is a standalone application measuring the usage of other programs on the client machine. The programs that are to be monitored for usage activities have to first register to the *client program* in order for the measurement to take place. The part installed on the server, referred to as the *server program*, is an application which can serve several *client programs*, receive, organize and store the data sent from the *client programs*.

1. Client program
1.1 The program shall run as a background service in the operating system
1.2 The program shall provide a service for other programs to register to, for performing the usage measurement
1.2.1 The other programs is when registering, able to choose the usage measurements the they wishes to have performed
1.3 The program shall provide a service for other programs to deregister from, for no longer perform the measurement
1.4 The program shall be able to measure the six measurements point specified in the subsections
1.4.1 The program shall be capable of measuring the Wall Time Usage of a registered program
1.4.2 The program shall be capable of measuring the Activity Level of a registered program
1.4.3 The program shall be capable of measuring the Activity Time of a registered program
1.4.4 The program shall be capable of measuring the Number of Users of a registered program, by counting the unique identifier of the user.
1.4.5 The program shall be capable of measuring the Users per Region of a registered program, by counting the unique identification of the user registered in a certain region.

1.4.6 The program shall be capable of measuring the Functionality of a registered program, which has defined functional measurements point
1.5 The program shall be capable of storing the usage data measured on the same computer which the program is executed on
1.5.1 The program shall encrypt the stored data
1.6 The program shall be able to connect to the server program over the internet, if the server program is available
1.6.1 If the server program is not available within 20 seconds, the client program shall attempt to retry the connection procedure every 6 hours if an internet connection is available
1.6.1.1 If an internet connection is not available the program shall attempt connection every hour
1.7 When connected to the server program, the client program shall be able to authenticate itself to the server program
1.7.1 The program shall have a unique identifier which is available only to the program itself and the server program
1.8 When connected to the server program, the client program shall be capable of sending the stored data
1.8.1 The data sent shall be sent using an encrypted protocol
1.8.2 If the transfer is disrupted or error prone the transfer shall be cancelled, and the a new connection is to be established
1.8.3. If the transfer was successful, the transferred data shall be deleted from the client machine
2 Server program
2.1 The program shall be able to connect to the client program over the internet, if the server client is available
2.2 When connected to the client program, the server program shall be able to authenticate the client program
2.3 When connected to the client program, the server program shall be capable of receiving the data sent
2.3.1 If the transfer is disrupted or error prone the transfer shall be cancelled,
2.3.2 If the transfer was successful, this should be communicated to the client
2.4 The program shall encrypt the stored data
2.5 The program shall provide a service for other computers to register to, for them to receive usage measurement statistics
3 Additional constrains on the client –and server program (for detailed description on important system qualities – see section 5.3)
3.1 Modifiability – Highly modular design [1]
3.2 Security – Establish audit trail for legal purposes in case of connection attempts followed by a failed authentication procedure
3.3 Availability – The server program shall be executed in a distributed server environment, supporting load balancing, migration and transaction control
3.4 Portability – The client program shall be implemented on all of the following major platforms: Linux, Windows and MacOS

5.2 System Overview

The System Overview gives an architectural description of the system both in graphic and text, showing the various nodes, software components and interfaces which the system consists of.

The usage measurement system consists of two parts situated at the *Client System* and the *Server System* (see Figure 10).

The client system contains the *Measurement Program* which measures the usage of programs which are registered using the *Internal Interface*. The *Measurement Program* was previously named the *client program* in the requirements (see section 5.1), but has been renamed in the Figure 10 in order to not confuse it with the programs installed on the *Client System* being measured. In the figure, two programs: *Program 1* and *Program 2*, are monitored by the *Measurement Program* which collects data, stores it encrypted, and sends the data through the *External Interface* towards the *Server Program* using an encrypted communication protocol.

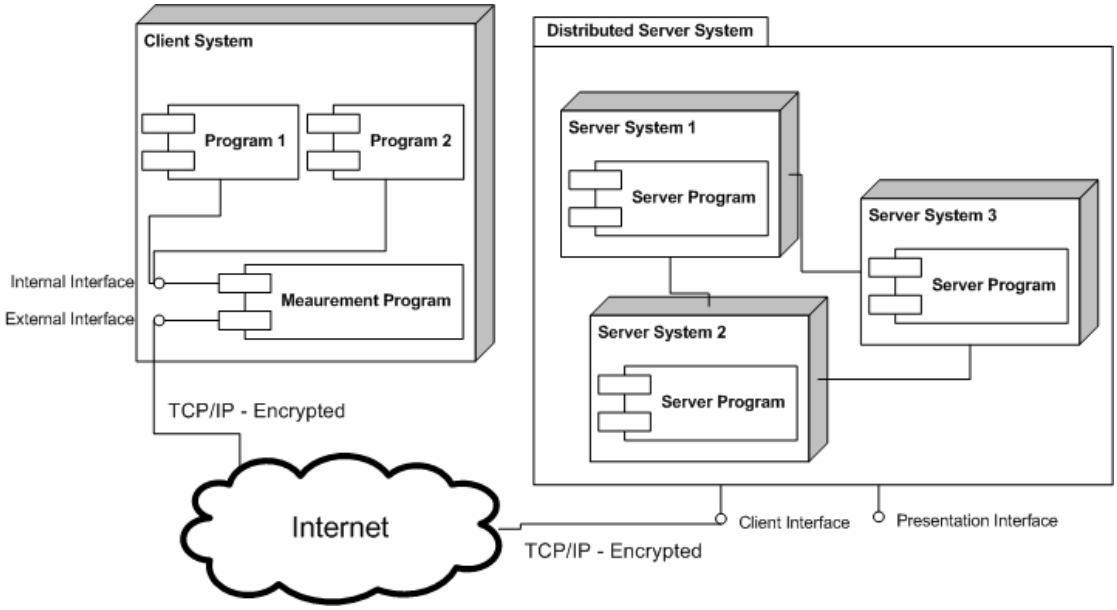

Figure 10 – Component diagram of the usage measurement system, illustrating a setup of one Client System and three Server Systems, united in a Distributed Server System.

The *Server Program*, here situated on three *Server System* machines, receives the measurement data through the *Client Interface* and stores it (see Figure 10). The *Server System* machines themselves are interconnected in *Distributed Server System*, e.g. using middleware which can enhance location transparency, availability and scalability [40]. Even setups enlisting multiple *Distributed Server Systems* are a feasible option for spreading the risk of system failure, connected to the geographical position of the *Server Programs*. The *Server Program* provides a public *Presentation Interface* for everyone to use. This interface provides access to the statistics from the users of the usage measurement system.

The usage measuring system's client part could have several forms, of which the four main candidates are explained below.

- Library – One solution is to install a library on the client, containing functionality for measuring and communicating with the server. Each program that supports usage measurements needs to include the library in order to use the functionality. Depending on the design of the library, a library solution leaves many decisions and options to its user. The user of the library will require knowledge to use the library correctly. In terms of *modifiability* a library works as a central repository providing functionality to its users, making updates to the library a small matter. This is only true while the library interface is not reduced in functionality, forcing users to change their implementation relationship to adapt to the changes.
- Plug-in – Using a plug-in containing measurement functionality, which the individual measured program installs before or after reaching the end-user. This solution is related to using a library, although some of the responsibility of configuring the library can be moved towards the plug-in, making the plug-in easier to use, and less configurable too. The initial distribution of the plug-in might be easy, although an upgrade could require an upgrade of all applications measured. An alternative would be to use a plug-in which uses a measurement library, to facilitate configuration and modifications. Either way, an upgrade to the plug-in itself would impose a *modifiability* issue.
- Stand-Alone Application – A traditional application which provides a specified interface to application wishing to have the usage monitored and measured. While a stand-alone application is similar to a library during installation and distribution, the *modifiability* of such a solution is more versatile. In a stand-alone application the GUI can be extended to support new functionality, e.g. visualization of personal usage statistics, where the same extension for the library solution would require modification on multiple places. Depending on the resources available by the operating system, a stand-alone application will have better control of the application environment than a library or plug-in solution. Better control of the application environment helps ensure that the intended quality constrains will be enforced.
- Two Part Application – In this solution the stand-alone application is divided into two parts, with the same responsibility, working in parallel. One part of the application operates closer to the operating system, e.g. in Linux as functionality embedded in kernel space. The other part operates as a normal application using the resources provided by the surrounding environment. The parallel work process provides a way for the *server program* to compare the measured result with each other, detecting inconsistencies. Most importantly the difference en the working environment provides a variance in the *security* level, serving as precaution against both application parts getting compromised. Installing the application part close to the core of the operation system can pose problems in regards to gaining access to the system core.

The choice of form for the client fell upon a stand alone application where the application can be controlled, facilitating the imposing of quality restrictions, and supporting *modifiability* by making functionality additions feasible.

5.3 Important Qualities

In this section selected system qualities are highlighted in order to elaborate certain choices and how these choices influence the system architecture.

5.3.1 Security

Security is a concern in most systems; threats may arise from within or outside the borders of an organization. For a system measuring and collecting data from clients to be used by various actors, the system must be trustworthy and that requires the system to deal with potential attacks.

The fist step against attacks is *resisting attacks* where the goal is to ensure integrity, confidentiality and assurance [12]. In the measurement system the *client program* will have to *authenticate* itself to the *server program*, using a unique ID and comparing the ID against existing IDs. When the *client program* is installed, a unique ID is generated and will initiate a registration process with the *server program* to store the ID and IP-address of the client machine. The registration of the ID and IP-address serves as a precaution against tampering with the intended use of the measurement system by having a large number of IDs per IP-address. In addition, the IP-address is used in the measuring of the number of users in a specific geographical region. Furthermore using *client program's* unique ID end users can connect to the *server program* which *authorize users* to access their usage statistics. For *maintaining data confidentiality* the data measured and stored should be encrypted, minimizing unauthorized access. Persistently stored data should be asymmetrically encrypted using two pairs of keys, meaning that both *client* and *server program* contains a private and a public key which they can use to encrypt and sign the data. The stored data must be transferred from the *client program* to the *server program* over publicly accessible network, escalating the potential threat level. For this purpose a Transport Layer Security (TSL) protocol can be used for preventing eavesdropping, message forgery and tampering, by providing communication encryption [12]. Besides confidentiality issues, the data should be delivered as anticipated, *maintaining integrity* of the data can be realized using a checksum, e.g. MD5. Checking for the correctness of the transfer by analyzing if the data is corrupt or incomplete, enables for the *server program* to request the data to be retransferred. The last tactic for *resisting attacks* is to *limit access* to the usage measurement system, limiting access to the implicated machines. The *server program* executing in a distributed server system will have a firewall situated for restricting the access based on where the request comes from and the destination port [12]. Although the server side environment can be controlled, the *security* level at the client machine is depending on the administrator of the machine, meaning that a firewall, hardware or software, might be available and in working state.

The second step to achieving *security* is *detecting attacks*, using an *intrusion detection* system, where network traffic patterns are analyzed and compared against prior patterns in history saved in a database [12].

The final step is the ability to *recover from attacks* made on the system, where the recovering involves identifying the attacker and to restoring the system [12]. The

usage measurement system has to identify the attacker in order for legal actions to be taken. To identify the attacker and collect evidence, the appropriate tactic is to *maintain an audit trail*, where the IP-address and a copy of the transactions made in the system are logged. Additionally, the copy of the transactions including the data in system that has been changed in the transactions can help system recovery [12]. The second part of recovering from the attack is to restore the system, setting the system in a working state using the same tactics described in the section regarding *availability* (5.3.3).

The overall *security* goal for the usage measurement system is for the users to trust it. The trustworthiness perceived by a user is related to the *dependability* which the system exhibits. Besides the *availability* and the *reliability* of the system, the *dependability* of the usage measurement system relies on the security, which for the users is expressed as the personal integrity. Personal integrity is typically maintained by applying various encryption techniques, making it difficult to view the data. Even well protected, a system is never unbreakable, thus posing a risk for the systems perceived trustworthiness.

5.3.2 Modifiability

Modifiability is a key factor in a broad range of systems, in open source development a modular architecture is especially important [1]. A modular architecture enables developers to extend the functionality of the system without being forced to change the core of the system [1]. In addition, a modular architecture enables increased parallel development, due to the alignment between the architecture and the physical distribution of the person involved in an open source project.

Tactics can be used to control the time and cost to implement, test, and deploy changes, to accomplish modifiability. One set of tactics which will be applied to the system is to *localize modifications*. This has the overall goal to assign specific responsibilities to modules, which limits consequences if the module is replaced by another [12]. *Maintaining semantic coherence* is a tactic to ensure that a module functions without being too dependent on other modules, and a method to realize this is to *abstract common services*. Abstracting common services, e.g. with middleware is to be used for the distributed server system, providing the *server program* with services such as transaction control and load balancing.

Another modifiability tactic could be used when designing the system to avoid a ripple effect. A ripple effect occurs when a change to a module initiates changes to another module and so forth. In this tactic the existing interface of a module is maintained, keeping the name and signature of the interface unaffected by an internal change to the module. Nevertheless, *maintaining existing interfaces* can not solve module inter-dependency problems regarding semantics or quality of service [12].

By improving the *testability*, e.g. using *separate interface from implementation*, the *modifiability* would be improved, facilitating the testing process when modifying the application [12]. Creating separate interfaces for testing purposes allows substitution of implementations, which can be useful when testing different measurement techniques in the *client program*.

The possibility and essentially the effort required for making a modification to the measurement system, e.g. adding new measurement factors, is important for the success and survival of the system. Open source products are released more frequently than non-OSS products, making *modifiability* a quality to consider in detail.

5.3.3 Availability

Availability is about being able to control faults arising in the system, making sure that faults do not create a system failure, thus causing the system to be unavailable to the user [12]. The consequences associated with system failure are not simple unavailability; on the contrary a system failure can manifest itself by providing a service which is different from the expected response. Ideally the system should be maintained by rectifying the faults thereby ensuring the continuous correctness of the system.

For the usage measuring system, *availability* in terms of 24/7 uptime is not the highest priority. However a lower *availability* e.g. at least 80%, would be sufficient. A client failing to connect and transfer the measured data will attempt to retry periodically, and transfer the measured data later. If the retransfer of the data is successful, the delay must be taken into consideration, in order for data measurements based on time intervals to be adjusted accordingly. This will of cause create an increased standard deviation of the measurements results.

What is of importance to the system in terms of *availability* is the correctness by which the system makes measurements and handles these, both in the *client* and *server program*. Faults in the measurement system creating malfunction of system functions, must be handled by the system to ensure the perceived trustworthiness of the system.

To handle availability issues several tactics need to be exploited to prevent, detect and recover from faults before they lead to system failure, thus compromising the *availability* and possibly the correctness of the system. Many tactics for managing the availability of the system are in-build options in operating systems, application frameworks and middleware [12]. Nevertheless, detecting the uprising faults as they appear is a crucial step for dealing with faults in a controlled manner. To detect faults using *exceptions* is a tactic where the faults are dealt at the place they rise, which facilitates isolating faults causing as little consequences for the remaining system as possible. Though many faults can be detected, the system should recover from these by preparing for possible faults and have means for repairing them. Most tactics regarding *fault recovery* include having redundant software components or even spare hardware, which need to be synchronized in order for them to relieve the faulty part system from work [12]. The *client program* should have a redundant measuring component which could compare the results, to establish whether the results are correct and if a misalignment was caused by a fault in one of the components. For the *client program* the application environment is harder to control, and a redundant hardware or a spare computer is not possible. For the *server program* a distributed server system using middleware, can help to ensure *availability* and even have in-build functionality for *fault prevention* like transaction control and advanced process monitoring.

Availability is closely associated with *performance* in practice, e.g. when many clients are in contact with the server, the server can become overloaded and thereby unavailable. As shown many *availability* tactics consist of adding more parallel computer resources, which is the same tactic used for achieving higher performance.

A simple technique for improving the *availability* of the *server program* and the distributed server system is to use DNS round robin, where a DNS request is handled by a DNS server, mapping the request to a list of IP-addresses sequentially. This means that the *client program* needs only to have one point of contact, specified in the communication information as one IP-address. The round robin increases the *availability* by providing load distribution which can be enhanced to providing load balancing by routinely asking the servers in the list if they are available and not overloaded. Subsequently the servers which are overloaded or unavailable are then temporarily removed from the server list, ensuring that operational servers receive requests.

5.3.4 Portability

Portability is important for the wide acceptance and usage of the proposed measurement system, where users can have a diversity of operating systems. *Portability* is closely associated with *modifiability*, where *portability* is modifications made to the platform. A platform in this context is an operating system.

The measurement system needs to be available on the most common operating systems, such as Windows, Linux and MacOS. An aspect to consider when mentioning software platforms and OSS, are the correlation between the number of users of a platform and the amount of OSS used on that platform. On some software platforms, e.g. Linux, UNIX and FreeBSD, most of the software is OSS, while the market share of the platforms are quite small on desktop machine. While machines running Windows as a platform typically contain a significant amount of non-OSS and less OSS, Windows has a desktop market share of nearly 90% (ref). This makes the Windows platform an important measurement platform and *portability* a quality worth prioritizing.

The realization of the usage measurement system on multiple software platforms requires for the implementations to be tailored and optimized on each platform. While some practical measurement details are specific to a platform the general architecture remains the same for all the platforms. This means that depending on design choices, much of the design and implementation can be reused as well.

6 Evaluation

This section contains the evaluation of the results, consisting of the usage measurement system. The evaluation is important for analyzing and expressing the forces and weaknesses of the proposed system.

The usage measurement system and its architecture is a broad solution, providing a platform for extracting usage measurements from clients, and making this available to stakeholders of an open source project being measured or simply anyone with an interest. Supporting six isolated usage metrics this solution is a good proposal for a measurement system capable of measuring OSS. While true, details of measurement system need to be specified in depth to assist the realization of the system. In specific, the connection between the usage metrics of interest found (see section 4) and the actual system infrastructure (see section 5) should be defined in detail. This involves creating a description in which manner the system conducts the actual measurement of one of the proposed usage metrics'. Conducting the actual measurements in isolation has been done before with success, what is interesting is the way the information is collected and managed by the usage measurement system. Of cause parts of the issue is solved by performing the design phase, specifying the proposed architecture, thus creating scenarios for measurement process.

In order to evaluate the usage measurement system further, a frame of references aids to compare and articulate the forces and weaknesses, and the choices leading to these differences. For this purposes the proposed usage measurement system is evaluated with the Debian Popularity Contest (DPC) as a point of reference [21].

6.1 Evaluation against the Debian Popularity Contest

The results presented in this study, contain a proposal for a usage measurement system with a similar functionality as the existing DPC. The two systems have different scope and the proposed system is not constructed as an extension of the DPC. The evaluation structure is the same as section 5.3 – Important Qualities combined with the requirements and measured usage metrics of both systems (see section 5.1).

The s*ecurity* issue is addressed in the proposed system, by tactics ensuring resisting attacks e.g. using multiple encryption techniques, detecting attacks and recovering from a potential attack both using *audit trail* and various *availability* tactics (see section 5.3.1 and 5.3.3).

In the DPC the *security* issue is mentioned in regards to the encryption of the communication but it is not enforced or implemented [21]. Other tactics regarding detection or recovering from attacks are not mentioned in the documentation available.

The *modifiability* of the proposed system is regarded an important system quality, especially in open source development [1] and tactics for achieving *modifiability* are

documented for the proposed system. The usage measurement system is to be implemented as a stand-alone application, where each application monitored needs to define functionality usage points and use an interface to register, provided by the system (see section 5.3.2).

The DPC documentation states no specific goals for reaching a higher level of *modifiability* [21]. What can be observed of the DPC is the way that it is distributed and installed. The DPC is often included as a package in distributions based on Debian, e.g. Ubuntu [26], which makes facilitates the installation process. Furthermore, the DPC generates a list of installed packages on the client machine and monitors these, making it possible to monitor all installed packages without making modifications to them for supporting the measurement. On the other hand the DPC does not define interfaces which can be used to support the removal of specific parts of the system and replacing them with other parts, thus making it possible to enhance it without changing the boundaries.

The *availability* of the usage measurement system is defined and tactics for ensuring a certain degree of *availability* are in place. For the detection of faults, *exceptions* are used. Having at least one distributed server system using middleware and DNS round robin, facilitates the prevention and recovery from faults (see section 5.3.3).

The DPC degree of *availability* is not defined, although the server part of the DPC is described, in the frequently asked questions, as a server responsible of receiving the collected information [21]. This would imply that no distributed server system or hardware redundancy is in place, having significant negative implications on *performance*, *availability* and *security*. The one positive factor, making the necessary *availability* less than 24/7 for the DPC, is that some part of the communication is done using email, where the unavailability of the server would result in later received information.

The *portability* has been of great importance when developing a proposal for a usage measurement system, monitoring and reporting usage on major platforms used for open source software. The issue of having cross platform *portability* is more important for the *client program* since the end users have the authority of the client machine. By consequence the system should be able to support Windows, Linux and MacOS (see section 5.3.4).

The DPC approach towards *portability* designed specifically for supporting the Debian Linux distribution and distributions founded on Debian. The narrow platform support and the specific implementation for Debian Linux, where the DPC measures the time usage of installed packages, make it hard to reuse the architecture, design and implementation for new platforms at a later point.

The usage measurement system supports six individual *measurements points*:
UUID – Universally Unique Identifier to distinguish each user for the system and is used to calculate the total number of users of a program.
Users per Region – A number identifying the number of users of a program in a specific region.
Wall Clock Time – The accumulated time usage of a program, which can be divided time intervals, e.g. for viewing program time usage over a week.

Activity Time – The accumulated time usage of a program, where *key strokes* or *mouse clicks* is inputted by the user.
Activity Level – A relationship between the accumulated time usage of a program and the time where *key strokes* or *mouse clicks* has been inputted by the user. A higher level of user input produces a higher activity level.
Functional Points – A measurement counting the number of times a specific functionality of program has been used. The definition of a functionality is defined by the program being measured. The number of times a functionality has been used can be divided into time frames of e.g. 24 hours or one week.

The DPC supports one *measurement point* being the accumulated *access time* for an installed package (referred to as *Wall Clock Time* for the usage measurement system) [21].

7 Discussion

This section contains the discussion where the methodology used for conducting the research is discussed in conjunction with the proposed solution. Furthermore, matters mentioned in the report, deserving additional attention are discussed in this section.

The used methodology utilized in the study is closely related to the result, creating a system measuring usage. The scope of the research is rather broad. First, an investigation of usage in both traditional software -and open source development, breaking down the findings to measurable points, analyzing the most common interests of open source project stakeholders, and combining this into usage metrics specific for the proposed system. Secondly, creation of a system infrastructure containing requirements, architecture and thorough analysis of important system qualities is performed. This broad scope of the methodology has some consequences for the result, meaning that some issues are not addressed in particular some details of proposed system (as described in the *Evaluation* - see section 6). The consequences for the result are the following issues I shall address below.

In the area of *security* as an important system quality, no system is completely secure; however a great deal of effort should be directed towards ensuring a high degree of *security*. For enabling an increased level of *security* in practice, the communication between the *client* –and *server program* should be documented thoroughly. Examples of the exact communication messages, at bit-level, should be illustrated for guaranteeing the proper use of encryption and other methods to ensure *security*.

In the area of *modifiability*, among others, the interfaces mentioned in the system infrastructure should be described in details, giving an overview of the actual data exchange through the interfaces. Especially the public interface from the *server program* (see section 5.2), named the *Presentation Interface*, should be specified to illustrate the services provided by usage measurement system.

An issue in regard to the system architecture is how to manage and ensure data consistency between geographical distributed *Server Programs* (see section 5.2). The issue becomes relevant when multiple *Server Programs*, or even multiple *Distributed Server Systems* need to synchronize or store the measured client data at a repository in order to provide a uniform service from the *Presentation Interface*.

Prioritizing the content of the study and thus report, as a result of the choosing to use a methodology with a broad scope, is necessary and crucial for reaching the expected result. The utilized methodology served its purpose well, achieving a result which fulfills the estimated result. And the result, a usage measurement system capable when realized, of providing open source project stakeholders with valuable usage statistics enhances open source products.

7.1 Measuring OSS – Ideology and Corporate Interests

The uprising of OSS development (FSF – see section 3.2), is founded upon some essential ideas, insuring openness and freedom in manners how the software can be used. Although large corporations and venture capitalists invest money and time in open source projects as a positive input for the development of the project (Investment Flow – see section 3.2.1), they tend to have an agenda.

Corporate interests can be compared to the more unspoken interests of the many developers, who in their private time contribute to developing OSS. Both obviously have interests, but where private contributors seek glory and knowledge from the development, corporations are merely driven by money. The company tactics can be versatile, making long term investments for branding or ensuring goodwill, or more eminent investments to influence the direction of the development. By controlling the direction, investing money in open source projects depending on others or staffing people for implementing a specific functionality, corporations can align the software development with their own strategy. Goals in a corporate strategy could be to ensure a certain quality of an OSS product, adding in feature imperative to the corporation or supporting an OSS product to diminish the competitive power of a commercial product.

The interests of a project or of a corporation or a venture capitalist are frequently the same, since both sides are interested in enhancing the open source project product and infrastructure. Even though a strong correlation in interests exists between private contributors, corporations and venture capitalist, the motivation behind the interests differ. Driven by personal motivation, private contributors strive to put their accumulated knowledge and experience into the open source project, habitually with the honor of doing the job as payment. Seeing their work being measured for usage statistics and used by corporations for achieving their goals, even if the result of measurements would benefit the project, they might reduce the involvement of private contributors. In such a scenario the outcome would be a possible loss of productivity.

The statistics collected and presented by the usage measurement system would be publicly available and would have significant value to the future development of open source projects, since feedback, e.g. on used functionality, number of users and user activity, is gained.

7.2 Trustworthiness

The success of a system such as the usage measure system relies in the end on the support of the users. The end users decide whether or not they feel comfortable with installing the application and letting it collect information regarding the usage behavior of the user. For ensuring a continuous positive experience when using the usage measuring system, an important factor is the systems trustworthiness, mentioned under the important qualities section (5.3.1).

Seen from the end users perspective, the usage measurement system will be an application which will have access to the client machine, collecting information and sending this information to a foreign actor. By consequence the system needs to fulfill some expectations in terms of functionality, the quality of the functionality, and more importantly the actor receiving the client information must be trusted.

Trusting an actor means trusting that the information is handled carefully in the data gathering process, in addition to using the information for a matter that is supported by the end user. The first issue is maintaining data integrity, for the end user this relates to the protections of personal data. For the receivers of statistical information it relates to the correctness of the statistical information. In practice, it has been proven that most encryption techniques can be defeated using proper techniques. As a result, complete data integrity can never be guaranteed; nevertheless a strong level of effort should be upheld for maintaining a high level of data integrity.

The second issue in trusting the usage measurement system is associated with whether the end user trusts or believes in the purposes which the measurements are used for. A large influence for conducting this research was the conviction that measurements statistics, on e.g. functionality usage, from the end users could help developers enhance the software. Although counting the number of users of a promising OSS product, making potential investors interested in the further development of the product, this interest might not be approved by the people involved in the open source project. This second issue is closely related to *Ideology and Corporate Interests* (7.1).

The trustworthiness of the usage measurement system as mentioned earlier, is a central matter to the wide acceptance of the system. For reaching a high level of trust from the user of the system, a high level of *security* is necessary, e.g. making sure that data integrity is sustained. A method to accomplish a higher level of security is to let the end users of the system register with a user profile when installing the *client program*. This could provide additional means of authentication, secure information used in case of a security breach and gain more information on the user. While this might seem as a positive control by the system and the consumers of the statistical information, the end users might regard it as a restrain on their personal freedom. In addition, though the registration will add to the security and perceived trustworthiness of the usage measurement system, the willingness to register requires a high level of trustworthiness in the first place. The goal is to strike a good balance between freedom, control, and trustworthiness for an open source application. As a consequence registration is not included for the proposed usage measurement system.

8 Conclusion

The rapid pace of release of many OSS products enables a constant improvement of the functionality and quality. On the other hand, this development requires an existing knowledge of what improvements are of the essence.

Measuring the usage of OSS, enables mapping the exact functionality exploited by users, thereby assisting developers to improve the product. Tailoring of the product based on user statistics, could optimize the product to its market. One option to optimize the product is removing sparsely used functionality and focusing on the elements central to the user. Following the same line of thought, potential investors of OSS can estimate the potential market of a software product by receiving statistical information on the usage of the software.

For acquiring feedback on the usage of an OSS product, this Master Thesis Study provides a proposal for a system measuring the usage of OSS. In the proposal, the requirements and architecture for such a system is presented together with a logical explanation of the decisions made and their consequences in relation to the resulting system.

The usage measurement system can measure six separate measurement points from the end users: UUID, Users per Region, Wall Clock Time, Activity Time, Activity Level and Functional Points. The result is a usage measurement system, capable when realized, of providing open source project stakeholders with valuable usage statistics for enhancing open source products.

With the knowledge that an interest exists, from actors like KDE in the open source community, it would be interesting to realize the proposed platform for measuring usage.

9 Future work

This chapter describes possible issues emerged from this research which deserves further investigation.

- Realize proposed platform – Implement proposed requirements and architectural description into a concrete solution. This can be followed by an analysis on additional usage metrics.
- Propose a structure for an independent organization, responsible of administer the proposed platform, which includes the *Client* and *Server Program*, the hosting and continuous evolution of the measurement platform. This is point is associated with the discussion about ideology and corporate interests, and trustworthiness (see section 7.1 7.2).
- Propose and develop a monitoring mechanism for the *Client Program*, which ensures that the measured programs are authenticated and digital signed.
- Extend the proposed platform to support the measurement of non-OSS, including an analysis of the implications.

10 References

[1] T. O'Reilly, *Lessons from open-source software development*, Commun. ACM, vol. 42, no. 4, pp. 32-37, 1999
[2] G. Von Krogh, *Open source software development*, Sloan Manage. Rev, vol. 44, no. 3, pp. 14-18, 2003.
[3] S. Raghunathan, A. Prasad, B. K. Mishra, H. Chang, *Open Source Versus Closed Source: Software Quality in Monopoly and Competitive Markets*, IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans, Vol. 35, No. 6, 2005.
[4] G. P. Dwyer, Jr., *The Economics of Open Source and Free Software*, Federal Reserve Bank of Atlanta, 1999.
[5] I.L Hann, J. Roberts, S. Slaughter, *Why Do Developers Contribute to Open Source Projects? - First Evidence of Economic Incentives*, Graduate School of Industrial Administration Carnegie Mellon University, 2002.
[6] J. Lerner, J. Tirole, *The Simple Economies of Open Source*, Journal Industrial Economy, vol. 52, no. 2, pp. 197-234, 2002.
[7] Cristina Gacek and Budi Arief, Januari/Februari 2004, *The Many Meanings of Open Source*, IEEE Software, www.computer.org/software
[8] Michel J. Karels, *Commercializing Open Source Software*, 2003, 1542-7730/03/0700, ACM
[9] OSTG Open Source Technology Group, Inc. 2007, *Source Forge*, http://sourceforge.net/
[10] SPI, Software in the Public Interest, Inc. 2007, *SPI*, http://www.spi-inc.org/
[11] R. Baskerville, "Conducting Action Research: High Risk and High Reward in Theory and Practice," in Qualitative Research in IS: Issues and Trends, E. M. Trauth, Idea Group Publishing, Hershey, 2001
[12] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice", 2nd edition, Carnegie Mellon, Software Engineering Institute, Addison Wesley, 2005
[13] G. David Garson, "Guide to Writing Empirical Papers, Theses, and Dissertations". Marcel Dekker, Inc, New York, pp. 121-162, 2002
[14] A. R. Hevner, S. T. March, J. Park, S. Ram, "Design Science in Information Systems Research," MIS Quarterly (28:1), pp. 75-105, 2004
[15] A. H. Maslow, "Motivation and Personality", 2nd edition, Harper and Row, New York, 1970
[16] J. Feller, B. Fitzgerald, *A Framework Analysis of the Open Source Software Development Paradigm*, University College Cork, Ireland, ACM, 2000
[17] R. Stallman, "Open Sources: Voices from the Open Source Revolution", 1st edition, O'Reiley Media, 1999
[18] Free Software Foundation (FSF), Plone Copyright by Alexander Limi, Alan Runyan samt Vidar Andersen, Accessed Mars 2007, http://www.fsf.org/
[19] The GNU Project, Copyright by Richard Stallman, Accessed Mars 2007, http://www.gnu.org/gnu/thegnuproject
[20] Tim O'Reilly, "Open Source Business Model Design Patterns" O'Reilly Media Inc., EclipseCon, March 1, 2005
[21] Debian Popularity Contest, A. Pennarum, B. Allombert and P. Reinholdtsen, viewed January 2007, http://popcon.debian.org/
[22] Open Source Initiative, *Open Source*, www.opensouce.org

[23] Raymond, Eric. *Shut Up and Show Them the Code*, Linux Today, 1999, http://linuxtoday.com/stories/7196.html, viewed March 2007
[24] Raymond, Eric. The Cathedral & the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary, First edition, 2001, O'Reilly Media.
[25] Red Hat Inc, *Red Hat*, www.redhat.com, viewed March 2007
[26] Canonical Ltd, *Ubuntu*, www.ubuntu.com, viewed March 2007
[27] salesforce.com, Inc., *salesforce.com*, www.salesforce.com, views March 2007
[28] Open IT Inc., *OpenIT – LICENSEAnalyser*, www.openit.com/products/licenseanalyzer.html, viewed March 2007
[29] Wu, Ming-Wei, Lin, Ying-Dar, Open *Source Software Development: An Overview*, National Chaio Tung university, Taiwan, 2001
[30] Raymond, Eric. The Magic Cauldron, 1999, found in The Cathedral & the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary, First edition, 2001, O'Reilly Media.
[31] Boehm, Barry. *Software Engineering*, IEEE Transactions on Computers (25:12), 1976, pp. 1226-1241.
[32] Flaatten, Per, McCubbrey, Donald, O'Riordan, P. Declan, and Burgess, Keith. Foundations of Business Systems, Chicago, Dryden Press, 1989.
[33] Young, Robert. *Giving it Away: How Red Hat Stumbled Across a New Economic Model and Helped Improve an Industry*, in Open Sources: Voices from the Open Source Revolution, Sebastopol, CA, O'Reilly & Associates, 1999.
[34] Omidyar Network, *Omidyar Network*, http://home.omidyar.net/index.php, viewed January 2007
[35] Open iT, *LicenseAnalyzer*, http://www.openit.com/products/licenseanalyzer.html, viewed March 2007
[36] Box, Brad, *Superdistribution?*, Wired Magazine September 1994 Idea Fortes, http://www.virtualschool.edu/cox/pub/94WiredSuperdistribution/, viewed February 2007
[37] KDE e.V., *KDE e.V.*, Accessed May 2007, http://ev.kde.org/
[38] Brunch, Jr., John. *SDSC DataStar – Parallel Programming Tools*, UC Santa Barbara, College of Engineering, http://www.engineering.ucsb.edu/~stefan/perf-tools-06.ppt#318,17,Timers, viewed May 2007
[39] Nullsoft, *Winamp*, General Preferences, http://www.winamp.com/support/help/50/General_Preferences.htm, viewed May 2007
[40] Bray, Mike, *Middleware*, Software Technology Roadmap, Carnegie Mellon University, 2007, http://www.sei.cmu.edu/str/descriptions/middleware_body.html, viewed May 2007