The portability of open source: a structural analysis of the modularity of the Apache web server.

by

Benjamin Ho

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 2000
[June 2000]

Author [illegible]
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by [illegible]
Eric von Hippel
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The portability of open source: a structural analysis
of the modularity of the Apache web server.
by
Benjamin Ho

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2000

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer [Electrical] Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Open source design is defined as the practice of distributing the rights to distribute and modify the source-code, or inner workings of a computer program, for free (see Appendix 1). It is a phenomenon that has allowed groups of independent individuals o create products of similar if not superior caliber to those created by traditional firms. One fundamental property of open source software engineering is modular design. The ability to divide up a system into different functional units is absolutely necessary to allow such massively parallel collaboration.

The Apache Web Server Project is one prominent example of the open source movement. Their main product, the Apache web server, written and distributed for free, and built originally in the spare time of individual independent users, commands a majority of the market for Internet web page servers, despite strong competitors such as Netscape and Microsoft [1]. By understanding how modular system design has allowed Apache to attain such success, it is possible to extrapolate the principles to be used to apply the open source paradigm to innovation in other systems.

Thesis Supervisor: Eric von Hippel
Title: Professor, Management and Innovation

Table of Contents

1 Introduction

An important trend in the software industry has been the recent success of the open source software movement. The advent of loosely organized, independent coalitions of individuals who volunteer their efforts toward the creation of a product that is disseminated for free, has led to the idea that companies may be becoming obsolete. Open source software is based on the principal that software can be written by individuals voluntarily combining their efforts by making individual contributions to a software project, contributions that are shared freely with the community as a whole and the public at large. Instead of companies motivated by the profit motive, there are individuals motivated by something unclear. The source of innovation has reverted to the individual, the user, the consumer. This movement overthrows traditional theories of the firm and industrial organization and has been hailed by many to be a revolution in product development. Historically, the open source software movement has been restricted to computer science academics and select professionals. Its usage has been limited to such obscure software products such as Linux, Sendmail, Perl, and Apache. Recently, however, these products have gained in prominence through their wide dissemination due to low costs and reliable reputation. This has attracted significant capital expenditure, whether through the initial public offerings of new companies like Redhat or VA Linux, or even from investments made by well established firms such as Hewlett Packard, IBM, and Sun Microsystems [2]. Some have even suggested that the entire software industry adopt the ownership-free communal organization of the open source movement. One radical remedy to Microsoft's monopoly power, proposed early during the Microsoft anti-trust suit, was for Microsoft to relinquish its ownership of its software, making itself open source. These radical musings inevitably stop there. The logical next step is rarely pursued. If the open source paradigm works so well for promoting innovation in software, can the same success be extended to other industries? Will we one day drive an open source automobile?

This is the impetus of our analysis. In the next section, we will examine the properties of the open-source movement and the salient features that make it work. These properties can be classified as being economic or architectural. This paper will focus on the concept of modularity, one of the architectural properties that make open-source work. To examine modularity in depth, we choose a specific open source project and analyze its architecture. We then use this to generalize these properties.

Apache is our method of investigation. As one of the most successful open-source projects, the architecture of the Apache Web Server makes it one of the highest rated and most popular web servers currently available [1] [3]. Its success was dependent on both the large cadre of dedicated programmers who worked for free, as well as the architecture and system organization that made their work efficient and worthwhile. We will examine their design choices with regard to the modularity of their system, and come to understand how this facilitated Apache's successful development.

Attempting to generalize these principles, we use the model of Apache's development to attempt to understand modularity in a broader context. We build on earlier work by von Hippel [4] who analyzes task partitioning. Is open-source development unique, or is it derivative of traditional software development? Is the software development paradigm applicable to other industries? How can we define or measure modularity in other industries? What is an open-source automobile?

2 Open Source

At the most basic level, the term open source refers to software in which the source code for a given piece of software is made available for free to the public. Software distributed without source code can be executed but not modified. (Software is normally distributed in machine code, as chains of 1's or 0's, which the computer can read but is essentially incomprehensible to humans. The source code is written in a programming language such as Sun's Java or Microsoft's Visual Basic, which is a language of instructions derivative of English and mathematics that can be compiled into machine code. The source code acts as the blueprint that specifies precisely how the software was put together.) Distribution of the source code allows anyone not only free use, but also the limited or even total rights to modify and redistribute the software. What this implies is that it is basically impossible to receive monetary compensation for the sale of the software itself, and that consumers receive the product for free. This perverse organization has interesting implications. Its existence is dependent on unique economic institutions that have arisen from the unique nature of information.

2.1 Definition

"The open source definition" by Bruce Perens along with the GPL (General Public License) is a statement published by the Free Software Foundation [5]. Together, they form a manifesto that formally defines the copyright rights of the open-source movement. These copyrights form the basis of the open-source movement in software.

This document formally defines the open-source idea on nine ideas:

- Free Redistribution – the right to freely modify and redistribute the software
- Source Code – the mandate to keep all source code for the software open
- Derived Works – the mandate that all derived works be subject to these same rules
- Integrity of the Author's Source Code – the mandate that the original authors' code is not modified
- No Discrimination Against Persons or Groups
- No Discrimination Against Fields of Endeavor
- Distribution of License – the mandate to distribute the license with the code
- License Must Not Be Specific to a Product
- License Must Not Contaminate Other Software

The full definition written by Bruce Perens is included in Appendix A. The document essentially mandates that not only is the currently available software free, but all modifications to the software also be open and free. This ensures the longevity of the project and prevents the project from fragmenting into commercial competitors.

2.2 Economics

Eric Raymond's seminal work, "The Cathedral and the Bazaar," provides an informal analysis of the open source software phenomenon [6]. His work describes mechanism that not only allows the open source paradigm to work, but also mandates its existence. His analysis provides a useful starting point in understanding the ideas of open source. In it, Raymond contrasts the "Cathedral" mentality of the traditional software firm, where software has a master plan and all the workers build toward the glory of the plan, to the "Bazaar" mentality of the open source movement where there exists a crowd of energetic individuals, each shouting for their own voice to be heard and each contributing to the din of the whole. We will use Raymond's work as a launching point into various parts of this paper.

In order to understand the architectural properties that allow an open source project to function, it is useful to understand the motivations of the people involved. The interactions of these actors form the economic institutions that have emerged to accommodate the participants' motivations as well as the goals of the project.

More formal analysis can be found in Lerner and Tirole's analysis of the economics of open source. Their work gives economic reasons behind open source's dramatic departure from conventional industrial organization, and attempts to bring its structure in line with conventional economic thought [2]. This follows earlier work on "user-driven innovation" in other industries such as Rosenberg's studies on the machine tool industry [7] or those of von Hippel in scientific instruments and ASIC chip design [8].

Open source software is a phenomenon that defies traditional economic models. When considering the economics of such a system, an alternative framework must be used. Some of the explanation of open source can be attributed to the "new economics of information," explored by the likes of Varian and Shapiro [9]. In their book, *The Information Economy*, they outline seven main points that differentiate information from other commodities. Innovation is an increasingly central part of product development and, like any other form of information, must conform to these rules:

- Information
    - Cost of Producing Information – Information has high fixed costs to create, but is virtually free to distribute. Or in other words, zero marginal costs. Innovations once invented can be shared at no cost.
    - Managing Intellectual Property – Since information is easy to copy, it is difficult to protect property rights.
    - Information as an Experience Good – The quality of information cannot be ascertained until it is already obtained. Often, an innovation can be easily copied once it is seen.
    - Economics of Attention – Information is only valuable if someone sees it.
- Technology
    - Systems Competition – Technology depends on the interrelationships of multiple systems. Innovation depends on others to be successful.
    - Lock-In and Switching Costs – Once a standard is reached, it becomes expensive to change. This creates incentives to be the first to innovate.
    - Positive Feedback, Network Externalities, and Standards – There is value added in creating standards. Information often becomes more valuable when there are more users. This creates disincentives for a company to innovate in a market where a leader already exists though the need may exist.

As the economy moves into an increasingly information driven age, Varian and Shapiro argue these effects become more and more prominent. This leads to the necessity for new methods for information creation, one of which is the move toward user-driven initiatives.

This has happened before in the case of the ASIC semiconductor chip design, scientific instruments, and many other industries, as studied by von Hippel [8]. In these industries, it was the *lead users* of a product, the users who were the most successful, who became the ones who create the innovations. Von Hippel's analysis is based on the concept of "sticky information," the idea that some information is difficult to encapsulate and therefore difficult to transfer. This can be applied when a user of a product has a problem with its operation, but that problem may be dependent on the environment in which it is used or its interaction with other systems that the user may have. These added variables that the manufacturer cannot see make it difficult for the manufacturer to come up with a solution. The information is "stuck" to the user side. However, it may be possible for the innovative know-how normally "stuck" to the manufacturer side, to also be available to the users. In this case, it becomes efficient for the user to innovate.

In software, this idea of lead user innovation becomes particularly relevant. Software is particularly susceptible to the "New rules" of the "Information Age." Further, the environment in which most software products operates are particularly complex. Typical software operating systems and hardware environments vary dramatically between users, they are typically known intimately by each individual system administrator. It would be difficult for the software manufacturer to address the concerns of each individual. However, since the users of much software are most likely to be software producers themselves, they tend to possess the same skill set as the original manufacturer. All they are missing is access to the source code that is also "stuck" to the producers by intellectual property concerns. However, the open source system eliminates these concerns, "unsticks" the information, and allows users to innovate.

Although this demonstrates the efficiency gains from allowing software to be open source, it does not explain the incentives that allow programmers to contribute to an open source project and share their work for free. There is no clear economic compensation to account for this altruistic behavior, especially since packaging for distribution incurs still further costs to the programmer's time.

The most common contribution to an open source project by a typical contributor is a fix to a specific programming error or bug, or the addition of a new feature. These arise from the immediate incentive for the programmer to solve his own problem. However, this is not enough to explain the wide scale adoption and success of open source, as well as the additional work required of the programmer to package the changes for distribution. It is difficult to explain why all programmers do not free ride, by using the changes of others as new updated versions of the software are released, without contributing himself. Though most programmers do free ride in most systems, significant numbers contribute as well as take. Typical reasons for this behavior by those involved typically refer to non-economic ideas such as the fun, altruism, ego, or reputation. On this issue, Eric Raymond writes the following:

> The "utility function" Linux hackers are maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers. (One may call their motivation "altruistic," but this ignores the fact that altruism is itself a form of ego satisfaction for the altruist) [6].

Lerner and Tirole, however, attempt to reconcile traditional economic theory with this apparently altruistic behavior among open source participants by appealing to the concept of market signaling [2].

But, there is still more to the story to be learned. Simple reputation or ego gratification is not a completely satisfactory explanation. A possible reason for a programmer to expend addition work to package his update for release is to ensure his changes are included and maintained in future versions of the software. Along similar lines, a recent study by Lakhani and von Hippel attempts to explain the success of the public Usenet help forum for the Apache web server, where users volunteer their time to answer the questions of others in a kind of open source help desk [10]. Lakhani and von Hippel find that although many of the contributors who voluntarily answer questions for others claim they do so for reasons such as fun or altruism, the actual reason is that many of them read the questions in order to learn, and that most of their time is spent reading the questions and answers of others. What they found is that although many question answerers do spend a considerable amount of time perusing questions on the public forum, this is not purely altruistic behavior. Most of the time spent perusing is for their own immediate benefit in terms of knowledge acquisition. The actual time spent answering questions is quite small, and though the most likely reason for this is indeed fun, altruism or ego, this is much easier to accept because the additional marginal cost to answer a question once time has been spent reading the list is quite small.

Even without an in-depth analysis, perhaps a more fundamental reason for the success of open source can be found in a simple analogy using Eric Raymond's Cathedral and the Bazaar metaphor. The Cathedral is a mighty symbol of the power of the state, a central authority attempting to create something by centralized fiat. The Bazaar, on the other hand is an example of freewheeling capitalism in its most pure form. History has shown which it thinks is best.

Economic incentives do exist to promote open source. However, as seen in the Lakhani and von Hippel example, these incentives depend on incremental change. Changes must be able to be made in small sub parts of the project that are independent, and must each solve a problem with meaningful results. Changes that are either too time consuming or lacking tangible outcomes require a much higher activation cost than the incentives provided to the typical user in an open-source economic system would allow. This does not disallow the possibilities of a catalyst, but in general, it does make open source development much more difficult.

3 Modularity

In their analysis of the automobile industry, Fine and Whitney partition product architectures distinctly between integral and modular design: "A product with a modular architecture has components that can be 'mixed and matched' due to standardization of function to <u>some</u> degree and standardization of interfaces to an <u>extreme</u> degree" [11]. Any product that has interchangeable parts provides good examples of *modular* architecture. Standard prototypes include home stereo systems where the CD player and the speaker can be made by different companies and have their own design, but both share a highly specific and standardized interface for their interconnections. The key to modular design is independence between components. At the other end of the spectrum along this axis, an *integral* architecture depends highly on the interdependence of all subsystems. The standard example of an integral architecture is an airplane. An airplane design battles constantly to maintain the balance to keep the plane efficiently aloft. Any change in one component has a dramatic effect on the functioning of the plane as a whole.

However, most product design architectures typically do not fall cleanly into one of these compartments. A completely integral architecture is complex and lacks flexibility. A modular architecture lacks coherence. Even home-stereo systems can have some dependence between different components causing benefits when different components are purchased from the same company, if for no other reason than consistency of user interface. This interdependence has been exploited by the home stereo manufacturer, Bose Corporation, who has recently been successful in selling their Lifestyle™ system, an integrated stereo product. Also, even in airplane design there is leeway in redesign of certain key components. A passenger jet does allow modularity in terms of its relative ease in retrofitting to carry different passenger seat configurations or cargo loads.

3.1 Kernel-Module Architecture

In computer architecture, this mutability of the classifications can be better characterized by using the concept of the kernel. According to *The New Hacker's Dictionary*, "The *kernel* is the essential center of a computer operating system, the core that provides basic services for all other parts of the operating system. A synonym is nucleus" [12]. The term kernel originates from the layered approach to operating system design. An operating system is the software loaded onto a computer that makes it possible for other programs to access and share a computer's resources. It is one of the most complicated pieces of software on a personal computer. Therefore, a modular design that allows programs written for the outer layer, the *shell*, to be written independently greatly controls complexity. However, fundamental control of the computer's processes require a great deal of interdependence. This interdependence is encapsulated in the integral part of the system, the kernel. The kernel creates the framework and acts as a sort of skeleton upon which the remainder of the program comprised of the individual modules, can be placed.

Traditional systems engineering methodology of a modular system typically involves the step of *decomposition* where each level of the process is divided into its constituent parts. This decomposition must have a "clear and terse interface with each other and with levels above them" [11]. The division of labor between modules is assigned to minimize complex interactions between systems. This division must occur and may occur only with some expense. The decomposability that this division allows becomes indispensable when companies seek to divide labor and particularly when companies seek to outsource.

In the kernel-module framework, the focus shifts from the inter-module interface, to the interface between each module and the kernel itself. In this framework, the responsibility is passed to the level above; the kernel design is responsible for maintaining the independence of interactions. Interactions between modules go through the kernel as the intermediary. Therefore, the interface between kernel to module is then responsible for the interactions.

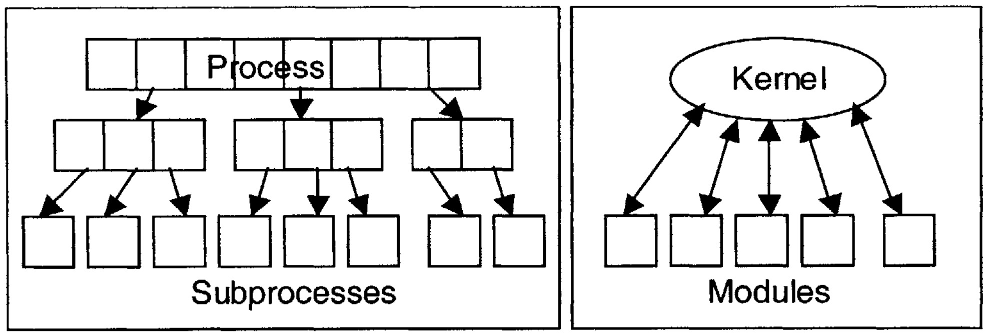

Figure 1 – Traditional decomposition vs. Kernel/Module

In many ways, the kernel represents the limits of modularity and task partitioning. It has two main purposes. First, it is useful as one method of layering in system design. The use of different layers is important in establishing hierarchy. The kernel encapsulates the low-level system calls hidden from the module developers, thereby eliminating one level of complexity and replacing it with a clearly and simply defined interface. This hierarchical approach simplifies development, and helps to isolate problems. Secondly, the kernel represents from the point of view of the module developers, the portion of the system that cannot be subdivided. The purpose of modularity is to subdivide complexity into independent functional units. The kernel in system design thus represents an indivisible core, able to handle all of the complexity of module interdependence by providing tools for the modules to use, thus making the job of the system designer to ensure module independence easier.

The benefits of modularity come from the ability to distribute the labor, the ability to isolate problems, and the reduction in complexity for later development and integration. The benefit of an integral system is only that it makes the initial system design easier, and allows possible optimizations that arise from exploiting module interdependence. The kernel system, therefore, serves as a compromise, allowing the benefits of modularity while giving the system designers the ability to handle certain problems with integral solutions. However, since the kernel still suffers from the design flaws of an integral system, it is still preferable to minimize the kernel's role. In operating systems, one of the latest developments is in *micro-kernel* architectures. These architectures attempt to improve modularity by minimizing the dependence on the integral architecture. As much functionality is delegated to the modules as possible.

3.2 Modularity in the Software Industry

> "Good programmers know what to write. Great programmers know what to rewrite (and reuse)." – Eric Raymond "The Cathedral and the Bazaar" [6]

This quote from Raymond's "Cathedral and the Bazaar" echoes the central mantra of computer programming. Abstraction is fundamental in computer architecture design. Modularity is the primary way to abstract a problem into simpler components. The "divide and conquer" mindset makes decomposition fundamental practice in software engineering [13]. This modularity, inherent in the computer culture, is a feature that facilitates the collaboration necessary for an open source project. Understanding why this is so is necessary for insight into how open source design ideas can be exported to other industries.

The main focus in software modularity derives from code reuse. The software industry is one in which specifications may vary little across different projects. In what is fast becoming America's largest industry, software reuse on a larger scale will be necessary to efficiently continue to create new software. Most software problems can be broken into simpler steps that are fairly atomic and consistent between different problems. Much time is wasted rewriting these sub problems when they arise.

Further, in the limited world of a computer system, small, simple mistakes can lead to dramatic consequences. For this reason, the compartmentalization of modules that helps in isolating mistakes is particularly important. Systems are especially prone to mistakes when arising from unexpected interactions between different functions of the program. Modules with well-defined barriers breached only by tight interfaces can effectively partition the program such that unexpected interactions cannot take place.

A third advantage of a modular system is the ease of updating. When new technologies are developed regarding a specific module, if the interfaces are designed properly and interdependence is appropriately minimized, then work must only be done in that particular module. This is evident in an area particularly relevant to open source software, portability. *Portability* is the ability for a given piece of software to be used across a wide variety of different types of computer systems, or platforms. Modularity allows a programmer attempting to adapt a program to a different computer system, to only change the modules that behave differently in the new system.

3.3 Modularity Tools

Modularity is necessary in order to make sense out of the complexity large software project by nature create. As a result, the industry has matured enough to develop a large sub-industry dedicated to establishing the interfaces that promote modularity between systems. These industries take design concepts and hardcode them.

Modularity's importance in computer software is underscored by the large variety of devices used to instantiate its principles in substantive outputs. At the lowest level, one method that codifies modularity is found within object-oriented programming languages based on object-oriented design (OOD). The most basic description of object-oriented programming is a methodology that organizes the lines of code that form the action of a program with the data it acts upon. The analogy is of a real world object that has physical properties, the data, as well as actions it can perform, the code. The concept is that by combining the data with code, it creates a unified package that can fit the properties of modular independence. An object-oriented programming language such as Java or C++ can enforce this by requiring the programmer to very specifically define the interface and prohibiting the use of data in other objects. Following the rules of the language ensures these rules are met, and therefore encourages modular design.

To extend this concept, there exists in software design the concept of design patterns. These are concepts that attempt to use the object analogy to put together programs based on an extended metaphor that proves useful for organization. This allows real world methods of task partitioning to be extended into software design. A simple example is the producer-consumer pattern, where one object is responsible for producing the data, and another, for consuming or using the data in a meaningful way.

As we move upwards still on the hierarchy, modularity becomes dominated by competing commercial interests. In terms of commercial application of these ideas, there exist software packages designed to aid large-scale computer system architects in the design process by keeping track of modularity in large systems. Whereas OOD methodologies are used to design individual software components and products typically intended for a single computer, larger scale systems that can encompass global networks of computers processing enormous amounts of data require more general module tools. Tools to design such systems come in several forms. They range from Computer-Aided Design (CAD) programs that merely provide graphical tools that help a designer visualize a large modular system. These include programs such as SELECT Enterprise. These packages are designed to coordinate the numerous competing standards, and provide graphical tools to aid in visualization.

The commercial interests vie to provide methods to promote the writing of reusable code. Each company has its own proprietary inter-module interface that it attempts to establish as standard. Most of these typically only specify the interface, leaving the workings of the module unspecified. At the level directly above objects and design patterns, falls the category of widgets and/or libraries. The two most prominent players in this niche are Sun's Java Beans, and Microsoft Visual Basic Custom Control (VBX). A typical widget may be code that draws and animates a certain type of button for a Graphical User Interface (GUI) or one that performs a specific type of calculation, such as a financial one. Both are tightly linked to each company's proprietary programming language, and therefore are somewhat constrained.

At the next level, the component level, this constraint is dropped. A component, more so than a widget, offers a complete integrated subsystem that is identified by functional task. A component may be responsible for managing the bankroll of a company, or for calculating a parameter in a highly specialized manufacturing process. Commercial interests are involved in setting the interface. Interfaces such as Microsoft

DCOM, OMG's CORBA, and Sun's Java/RMI are software packages that systematize the interface between different components.

The momentous growth of the German company SAP, exemplifies the rise of another recent trend in software, the use of Enterprise Resource Planning (ERP) systems. ERP systems attempt to integrate all of the disparate computing systems that a large multinational may have by establishing common interfaces and common protocols. The idea is to allow a company's once disparate systems, such as customer billing or inventory management, to communicate. What companies like SAP offer is consistence of interface.

At a higher-level still, development is underway to establish eXtensible Markup Language (XML), perhaps the ultimate interface. This is a language designed represent any type of data in a machine independent way. This creates an interface that facilitate e-commerce by allowing computer systems from different companies to communicate.

Traditional software design practices, however, have focused on peer-level interfaces between modules. XML is designed for company-to-company interactions. Components are widgets designed to be connected to each other. Hierarchy only exists as a way to organize groups of peer-level subparts into large pieces. One essential development used extensively in open source design that is not standard software practice is the kernel-based approach. Kernels have long been a part of operating system design. Strictly speaking, the term kernel bears the connotation as the core of an operating system. However, the similarity of function between a core of an operating system and the core we will describe here bears enough resemblance that we will appropriate the term. The carry over of the operating system approach to other systems has proven essential in maintaining the viability of open source projects. In order for incremental improvement to be possible, there must be some initial integral structure that establishes and then maintains the framework for development. Someone needs to spend the effort on a strong initial design:

> "Smart data structures and dumb code works a lot better than the other way around." – Eric Raymond "The Cathedral and the Bazaar" [6]

A common story in the history of successful open source projects has been an early complete reworking of the fundamental system architecture to accommodate open source development. The most influential open source project, the Linux operating system germinated by Linus Torvalds, began in Torvalds own words as a *hack* – a quickly assembled, largely unplanned software program. However, as the number of contributors began to grow, one of the first revisions was to rewrite the kernel to accommodate a more modular design [14]. Similar occurrences happened with both Sendmail and Apache. Both started off as garage style replacements or adaptations of more traditional programs written originally by a single individual. When they began receiving the widespread attention of the community, the first major enhancement was the improvement of the modularity of the core software.

4 Apache

The Apache web server is, by some accounts, one of the most successful open source projects. According to Netcraft's survey, as of March 2000, Apache commands over sixty percent of the market share of public web sites [1]. It is the most popular program used to allow access to the content of web pages over the Internet. Aside from deriving its immense popularity from its free cost, it also has a reputation for being one of the most reliable web servers, one of the fastest web servers, and the one quickest to adopt new technologies. Far from being used by only the specialized or for small time projects, Apache runs many of the most trafficked web sites on the Internet.

4.1 History and overview

In 1995, the nascent World Wide Web was gaining its first users. The dominant web server in the at that time tiny web server market was the program developed by Rob McCool at the National Center for Super-Computing Applications (NCSA) and referred to as the NCSA server. The NCSA server was a piece of software in the public domain, meaning it was freely distributed. However, it lacked both a modular design and a group dedicated to supporting it. At the time, NCSA was in version 1.3, but bugs persisted. In an early email to what would become the founding members of the Apache Group, Robert S. Thau defines their original mission:

> However, 1.3 is not a perfect product, and in the absence of further visible development from NCSA, a lot of people have found themselves fixing or extending 1.3 to meet their needs --- in the process fixing the same bugs and deficiencies over and over, at different sites. For instance, there are now three initgroups() patches floating around. This group consists of people who've all had to patch 1.3 at one time or another, either to extend server functionality, fix bugs, or improve performance.
>
> Our goal is to produce a revised version of NCSA 1.3 which has all the popular fixes in it directly, in order to have a supported server which actually meets our needs [15].

He later goes on to say that their changes will be distributed freely under the name "Apache." The Apache Group's original vision was as a repository for *patches* or small fixes to the original NCSA code, that would become a central location where people could share their additions. The name "Apache" is a play on the words "**A PAtCHy** server." Soon, the success of the project snowballed, leading to a complete redesign, abandoning the NCSA core in favor of a modular one of Apache's own creation. However, the practice of basing improvement in Apache on incremental change has persisted.

One of the requirements that Eric Raymond lays out as necessary for an open source project is strong leadership. Significant development decisions must be made when the core is being designed. Much of the failure in open source systems occur because of conflicts among the founding members. Although Apache is unique among successful open source projects in that it has no single figurehead like Linus Torvalds for Linux, it had strong leadership in terms of a committed core group who had an efficient system for making decisions without requiring consensus. All design decisions and decisions regarding what changes are adopted and what modules are included are put before Apache Group members via their mailing list and voted on. Code changes require a minimum of three "+1" (yes) votes and zero "-1" (no) votes or vetoes [16]. All other decisions require three positive votes, and a simple majority. The Apache Group is based around this email list. From the beginning, they overcame distance by using email as the primary method of decision making. The following architecture is the grand design that this group has laid out.

4.2 Architecture

4.2.1 Web Servers

The Internet provides a wealth of information in the form of text, images, sound, video and more. The World Wide Web, created by Tim Berners-Lee, is one method of organizing information into web pages that can be accessed by a program called a web browser. We will use the term *document* to describe any information that is obtainable over the web.

The process that retrieves information via the web begins when a user makes a request using a program called a browser. This request is typically for the contents of a web page. It makes this request by opening a connection via its Internet connection to the address of the computer containing the document it is requesting. The full request is called a *Uniform Resource Identifier* (URI) or *Uniform Resource Locator* (URL), and looks something like http://web.mit.edu/bho/www.

It is the responsibility of the web server to listen to the Internet connection for browsers requesting documents. When a request is made of the web server, it is the web server's responsibility to decode the request, perform all processing as configured by the server administrator such as user authentication of decryption, retrieve the document, typically from a local disk, and finally pack the document to send to the browser that requested it.

It should be noted that the web server must perform these tasks for every request. A busy server may be asked to deal with thousands of requests at a time, if not more. In order to allow for this, typical web servers can be written to *clone* themselves such that a given computer serving as a web server is running multiple copies of the web server software. Other web servers are designed to deal with the multiple requests internally in a process called *threading* which has a single program perform multiple tasks simultaneously. A final solution, and perhaps the most modular, is for one server program to delegate responsibility to sub programs that it supervises. Apache, as we will see, uses a combination of these ideas.

4.2.2 Apache Design

Due to the paucity of published information on the subject matter, the information presented here is based on research from the information published on the official Apache Web Site, their documentation on the API, the paper on Apache API design by Robert S. Thau [17], the conversations between Apache Group members via their email list, and personal examination of the source code to the project. Thau, then a PhD student at MIT, was primarily responsible for redesigning the Apache architecture after its initial success. His primary contributions included a modular structure and API for better extensibility, a pool-based memory allocation, and an adaptive pre-forking process model.

The most significant of the modifications made to the initial Apache design was the creation of the Apache API by Robert Thau. The API is the component of the design that defines how modules are created. It was something unavailable on the original NCSA server, and is perhaps the most significant feature of Apache that led to its early success. The API is the heart of Apache modularity.

An *API* (*application program interface*) is the method prescribed by a complex computer program such as an operating system that allows a programmer to write add-ons that interact with the kernel or the core. Strictly speaking, the API is the formal definition that specifies the interface programmers must use when accessing the core. The primary reason Thau provides for creating a API for Apache is to allow the server administrator extensibility. Basic web server design only allows extensibility through the *Common Gateway Interface* (CGI), a protocol that allows a browser to request a document that is actually a program. The CGI interface, however, is a very strict interface, allowing very limited access to the Apache core. Also, components written with the CGI interface are actually separate programs from the server, thus requiring additional system resources. Code written via the API becomes embedded into the Apache program process, thus reducing the load on system resources.

When Apache hears a request from a browser for a document, it will handle this request in eight phases:

- URI to filename translation – converts the browser request to the filename of the corresponding document on the local server
- Authentication ID checking – validates the identity of the user of the browser who made the request.
- Authentication access checking – ensures this user has permission to access the requested document
- Other Access checking – this additional phase for access checking multiple modules the opportunity to authenticate
- Determining the MIME type of the requested entity – the *MIME* (Multi-Purpose Internet Mail Extensions) is a protocol that specifies the type of file being requested, i.e. text, audio, video, images, application, etc.
- Fix-ups – a phase for things that do not fit elsewhere
- Sending data back to the client – fulfills the actual request

- Logging the request – Apache logs everything even if only an error message is returned

Modules written for the API interact with the server core through handlers. Each module can include up to one handler for each phase of the request cycle. At each phase, the Apache core will look at all handlers that are registered for that phase. It will then execute these handlers in arbitrary order (order unspecified by the design). Each phase can be thought of as a hook, upon which a module can hang its handlers, and thereby attach itself into the Apache execution cycle. When each handler is executed, it is given a pointer that directs it to a request data structure stored in the core that contains all pertinent information about the request. This information begins with only the URI, but handlers in various phases will modify/add information such as document type, user access, directory configuration, etc. Each handler can only return one of three things: OK, DECLINED, or an HTTP-error code. Returning a message of OK indicates that the request was successfully processed. A DECLINED message indicates that this handler was not designed to handle the type of request it was presented with. The core will then try to find another handler to process the message. An OK message indicates that the handler completed successfully, and that the request structure was appropriately updated. Depending on the phase, this may complete the phase's task, and the server moves to the next phase. The URI to filename translation only needs to occur once, so the server advances as soon as a handler returns successfully. Alternatively, for a phase such as the Request Logging phase, it is beneficial to ensure that every piece of information is properly logged, so all handlers registered with the Logging phase are executed. The third option is an HTTP-error code. These codes are specified by the W3C Consortium which establishes the standards for World Wide Web interactions. This aborts processing of the request. The kernel sends bypasses the remainder of the phases, sends to the requesting user an error message, and logs the transaction.

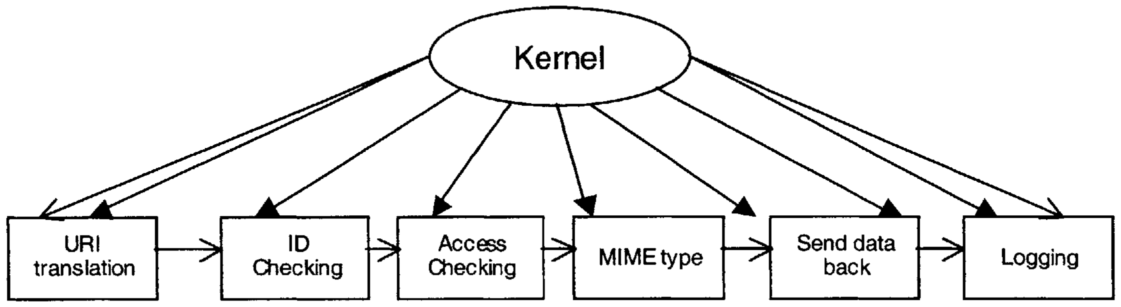

Figure 2 - The dotted lines are the abstract phase transitions. In actuality, the kernel maintains explicit control and runs each phase in order.

The arbitrary ordering of handler choice is one of the problems Thau acknowledges. He suggests future versions to use a priority based system where a module can specify the priority its handler should be given, or one in which the server administrator has some control over the ordering.

This is one of the keys to Apache's design philosophy. A handler has the potential to preempt another module and prevent it from executing is because the structural enforcement of module independence is relatively weak. The reason for this is that structural fences to enforce modularity rules are expensive in terms of performance. Apache depends on human oversight instead. Like most open source projects, the leadership has final say over what is added to the Apache distribution. Designated experts examine all code revisions and all new modules. The Apache modularity rules are maintained by supervision as opposed to hard coding.

Thau's other early design innovation was a pool-based resource model. The pool-based resource model is a system that centralizes the resource distribution. Software needs access to the resources of the computer it is running on, particularly access to the computer's memory. The operating system is responsible for providing those resources. The problem is that when processing many requests simultaneously, each module, running for each request will occupy its own space in memory. This risks the possibility of memory leaks occurring in memory in more places, as well as suffers the consequence that each module reserves a segment of memory that cannot be used by others even when not in use. In order for Apache to be scaleable, this had to be changed. When a module's handler is being run, instead of requesting memory from the operating system, it requests memory from a pool of memory obtained by the core. The core can then manage the memory so that leaks by the modules using core memory can be handled. Also, it can dynamically reallocate resources so that memory, while not in use by one module, can be used elsewhere.

4.2.3 Concrete architecture

The above ideas form Thau and the Apache Group's module design considerations at a conceptual level. Using program analysis software *cfx* and *grok* to decompose the source code of Apache, Jean Preston provides an analysis of the concrete architecture of Apache, specifically Apache v1.3.4, as it is actually implemented [18].

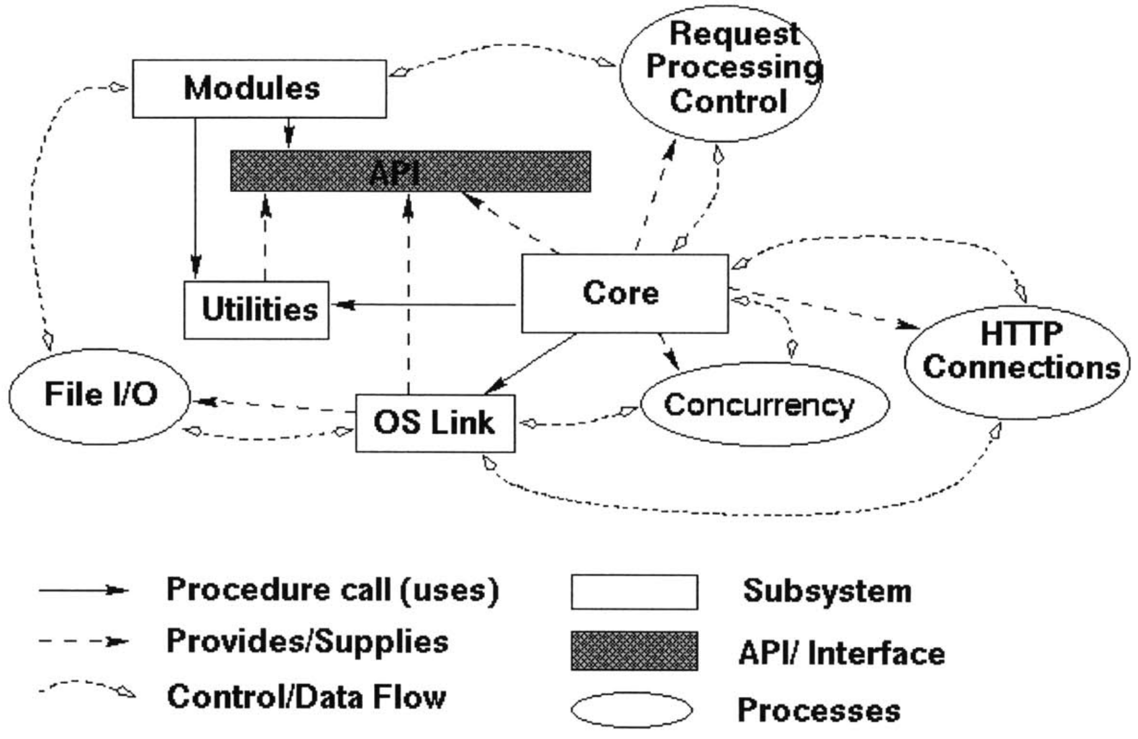

Figure 3 - This is the result of Jean Preston's analysis of Apache's concrete architecture [18].

In the actual implementation, there is a strong connection in terms of code interdependence between the subparts of Apache. Though a high level of modularity is encouraged, the structural mechanisms used to partition the modules do not even go as far those found in object oriented programming, much less the higher levels of encapsulation found in widget design or component interfaces. Apache, like Linux, is written in the C programming language, the language from which the popular object-oriented language, C++, derived. The use of structural devices to ensure modularity tends to hinder performance. Therefore, all code is compiled into one program, and modularity is instead maintained through adherence to protocol and enforced by the other programmers in the Apache Group.

Conceptual rules for modularity are for the most part maintained, but pure independence is sacrificed for the sake of performance and practicality. Aside from the performance requirements, an early design decision was for Apache to be completely compatible with the functionality of the original NCSA server. This caused problems because of the NCSA server's *monolithic*, or integral, design. NCSA, however, was defined as a monolithic product, allowing it to perform functions that require interdependence between different modules in Apache. To accommodate these functions, extra functionality is undertaken by the Apache core.

This gives Apache module designers extremely flexible control over their operations. They are given more access to the Apache core functionality than should be permitted within the bounds of their modularity interface. This again puts added responsibility on the leadership for enforcement.

Occasionally though, even this flexibility in the API was insufficient to accommodate all the demands that have since been required of it. There have been extensions that were written that had to bypass the API commands, and patch the core code directly. One such extension is SSL. SSL (Secure Sockets Layer) is an Internet protocol that allows information sent over the Internet to be encrypted. However, in the original API design, the server kernel was responsible for all communication back to the user's browser, and so there was nowhere in the API that allowed the response to be safely encrypted.

The code dependencies found from Preston's concrete analysis corroborates the connected nature of Apache design. Ideal modularity would have independently packaged units connected by finely defined precise interfaces. However, practical considerations often win out. Resources must be shared, and inter-module efforts occasionally coordinated.

Apache was designed with modularity in mind. However, it originated with a small group of developers, a group small enough to undertake the monolithic decision making necessary to build the Apache core. The decisions that the founding team made to build the kernel of the project, and the leadership they maintained over the project, have been responsible for its success. Despite the modular design that engenders a distributed development team, supervision is necessary to ensure that these rules are followed.

4.3 Analysis of Apache

Apache was certainly successful in mobilizing thousands of developers across the country in a joint software project. It was also clearly successful by becoming the majority web server in public networks. However, there is an old saw in computer development formalized in Brooks' *The Mythical Man-Month*, that productivity per worker decreases exponentially as the numbers of programmers in a project increases linearly [19]. Therefore, though there have been thousands of programmers at the task, this innovation has primarily been limited to small, incremental improvements, improvements that are within the scope of a single person. Work is discretized into small enough independent pieces so that to the point of view of each individual developer, there is only one developer. This allows open source projects to avoid the pitfalls of Brooks' exponential decay. Apache's and much of open source's method of improvement and innovation is patch based and incremental. It is this understanding, upon which the small module architecture is based.

Name	*Description*	*Number of Lines of Code*	*Number of Predicted Man-Hours, based on COCOMO model*
mod_access	Host based access control.	350	121.1508
mod_auth	User authentication using text files.	276	94.40812
mod_cgi	Invoking CGI scripts	612	217.8431
mod_imap	The imagemap file handler.	855	309.4705
mod_log_config	User-configurable logging module	1103	404.3516
mod_mime	Determines document types	695	248.9653
mod_proxy	Caching proxy abilities	846	306.0509
mod_speling	Automatically correct minor typos in URLs	509	179.5182
mod_unique_id	Generate unique request	354	122.605

Table 1: Table of select modules and their code size. The number of man-hours is based on the COCOMO model of software engineering which gives Man-house=152 * Man-months = 2.4(KDSI)$^{1.05}$ where KDSI = thousands of lines of code [20]. As of Sep 1996, there are 47 standard modules written for Apache. Whereas there have been 6105 fixes between Sep 1996 and May 2000. Of these, 2068 are of modules. This averages 44 per module. (Note, the number of lines of code do not include the 58 lines of each module devoted to restating the Apache license, but they do include all other comments and documentation in the module.)

However, this has given rise to much criticism of the open source movement that innovation occurs only in areas where tangible improvements can be made with only relatively little work, work within the scope of a single person. The incentive is not strong enough for one individual to devote time to large and dramatic innovations. At the same time, there is too little coordination to allow one individual to work on a project that depends on others for success. There may be vast numbers of people, each putting in time to solve problems, but the problems they solve are shallow. These are problems such as bug fixes, speed enhancements, or adaptation of ideas pioneered by others. These are the innovations from which open source derives its strengths.

Enhancement	*Description*
Unix Threading	Allows both multithreading and multiprocessing. The two main methods for efficiently handling multiple requests simultaneously.
New Build System	The build system has been rewritten from scratch to be based on autoconf and libtool. This makes Apache's configuration system more similar to that of other packages.
Multi-protocol Support	Apache now has some of the infrastructure in place to support serving multiple protocols. mod_echo has been written as an example.
Better support for non-Unix platforms	Apache 2.0 should be faster and more stable on non-Unix platforms such as BeOS, OS/2, and Windows. This is based on the Apache Portable Runtime.
New API	The API for modules has changed significantly for 2.0. Many of the module-ordering problems, and module ordering is now done per-hook to allow more flexibility. Also, new calls have been added that should allow modules to do more without requiring patching of the core Apache server.

Table 2: Table of major enhancements to the core, as reported in the documentation provided with the Apache 2.0 distribution. The Apache 2.0 distribution is the first major update of Apache since 1996.

Apache was quick to adopt the technology of others, such as *cookies* and SSL technology from Netscape, or the CGI scripting of NSCA. Apache is also quick to port itself to a vast number of different platforms, ranging from Linux, to Solaris, to NT, to any number of other obscure platforms that would not be worth the time of a commercial development team. Finally, Apache is fast. Given the compromises in modularity for performance, as well as the time these developers put in to optimize the code, Apache is among the fastest servers in the market. Speed is a tangible result, and therefore is sufficient to hold the attention of Apache contributors. Incidentally, the few web servers that are faster, thttpd, mathopd, and boa, tend to give up modularity for the sake of speed.

Nevertheless, Ko Kuwabara's comparison of open source software to evolutionary practices is very apt [14]. Though the scope of Apache's contributions to innovation may seem limited, this may not necessarily be case. In evolution, change occurs very slowly, and the only changes that stay are ones that have immediate payoffs.

However, given enough tries, dramatic advances such as the development of the eye or of wings can come about. Whether Apache is responsible for dramatic innovation remains in debate.

However, private networks such as those used for intranets continue to distrust Apache. In networks behind a firewall that prevents public traffic, Apache is only used in 7% of systems. There are still a number of disadvantages to Apache that make it inaccessible. The primary problem with open source is the lack of accountability. There is no guarantee of security or reliability, and little technical support. Also, for the casual user, Apache is significantly more difficult to deal with than packages put out by Netscape or Microsoft. However, this is not necessarily an insurmountable problem. Public Internet discussion forums such as those on Usenet studied by Lakhani and von Hippel have long provided a user-based answer to the technical support problem. Other institutions such as companies such as Redhat may develop to fill the role as guarantor of reliability. These will be necessary in order for open source to continue to grow.

5 Discussion

This analysis of the power of modularity has generated a number of questions that need to be addressed.

5.1 How do we generalize modularity? How do we measure modularity?

As it is, modularity is not unique to software. The definition I used for modularity derived from Fine and Whitney's analysis of the auto parts industry [11]. Modularity is already standard engineering practice. In Fine and Whitney's paper, they study modularity as it is used in the make-buy decision for the car industry. As outsourcing of work becomes a more and more important component of manufacturing as a way of reducing costs, careful modular design with very clear specifications for interface and function must be used. Aerospace is one industry that almost best exemplifies decomposition into subsystems. Aircraft design is so complicated, modularization and layering is the only way to deal with the complexity.

One useful analogy to take from Apache, however, is the kernel model used for operating systems. This is a sub-case of modular design with layering in which the focus from inter-module dependence in peer layers shifts to the interface between each individual peer module with the encapsulated monolithic layer above known as the core or kernel. This level of abstraction allows us to differentiate between the modules designed specifically for open source innovation from the monolithic kernel within that contains the portion of the design indivisible into pieces small enough for mass consumption.

5.2 How specific is open source modularity to Apache? To Software? To Innovation?

The key to modular design in Apache that has made it successful as an open source project is the ability to divide the problem into sub-problems that have individual goals. We have already seen that the modularity built into the design of Apache is not particularly robust, rather it is riddled with interdependencies and incomplete specifications. These problems are resolved in Apache by the Apache leadership. However, the feature of the design that Apache's success can be attributed to is the ease by which a sub-component can be tangibly improved. A feature that can be called its *upgradeability.*

This property of upgradeability can be used to define criteria for using the kernel-module design. An appropriate kernel-module design would contain modules that have pieces simple enough to be upgraded. These modules must be independent, but have tangible effects on the output of the product as a whole. The remainder of the functionality of the design as well as the functionality that would integrate the workings of the modules would then be part of the kernel of the system. In order to be successful, all significant functionality of the product must lie with the modules. The kernel's main job should be to operate the integration of the individual parts. Otherwise, the coordination institutions as they now are are not strong enough to allow sustainable growth.

Automobiles and aerospace are just two of the many examples of modular design in other industries. They can be used to examine and evaluate these design criteria and decide whether open source has a chance to propagate. Cars are in fact already quite modular. In early car design, say in the 1950's, the kernel could be thought of as the chassis and little more. The rest of the components – the engine, the carburetor, the transmission, etc. – could be thought of as modules mounted together in an interface whose only external requirement was that it must fit within the mounts of the chassis. However, at the time there did not exist the widespread communication medium of the Internet to allow massive collaboration to share new ideas. Otherwise, open source may have started then. Perhaps there might have been enough word of mouth to achieve some level of standardization and sharing – this remains to be studied – but certainly there was not enough to supplant the commercial auto industry. Today, the communications medium exists, but cars are now significantly more complex. There is today, either the lack of interest within a large portion of car users, or enough knowledge for users of today's highly integrated car systems to introduce their own designs. This however, is an institutional problem, and not necessarily one with the structure.

Aerospace, however, has other features that prevent open source from emerging. Aside from also requiring a high degree of specialized knowledge, an airplane has a high degree of interdependence between parts. The interface between is highly specified, and there is no flexibility for error. This makes independent user innovation difficult to achieve. As seen in Apache, some flexibility in design is preferable.

5.2.1 Architecture Design Prescriptions

Von Hippel's study of modularity using the idea of task partitioning focuses on how managers divide a task so that it can be worked on by multiple units [4]. The paper partitions this process into two methodologies: the reduction of sub-process interdependence versus the encouragement of inter-unit cooperation. Here, we attempt to specify the structural requirements to achieve this task partitioning of innovation, and more specifically, the requirements that the partitioning is sufficient to attract an open source development community.

As seen form the Apache example, the basic requirement is for modules to be independent, and that their interfaces must be are well defined but not overly constraining. The sub-problems that these modules represent must be tractable to a large enough population of possible contributors. This may be difficult, but these requirements can be mitigated by the use of a well-designed kernel. The kernel can be used to hide the complexity that is necessary to the design. It can be used to encompass all the parts of the architecture that do not satisfy the properties of modularity.

There are two competing constraints on the kernel-module design at this point. Modules must be designed to ensure their simplicity to find an open source community willing to contribute. Overly complex modules will not be able to attract enough contributions because the effort required to make a tangible improvement would be too high. However, care must be given so that the kernel remains in the supporting role. Innovation can only occur in processes based in the modules. For innovation to occur in a kernel driven function, the benefits of modularity would be lost.

5.3 So what is cause of Apache's success? Is it structure or is it economics?

Above, we defined a prescription for architecture design using modularity that is necessary for open source. However, the question remains as to whether modularity or more specifically, kernel-module design is even a sufficient measure for guaranteeing the success of an open source project. A successful project such as Apache may owe its success not to its architectural design, but to the economic institutions and conditions that occurred around it.

The most commonly cited reason for failure of open source project is a failed leadership, not a failure of structural design, but of institutional design. The most difficult aspect of open source to control is to understand the incentives that cause those to volunteer their efforts for little obvious gain.

But perhaps structure does play a role in creating these incentives to join open source projects. If the primary incentive is for achieving notoriety or at least self-satisfaction for achieving tangible results, then the relative difficulty of module upgradeability becomes important. If the activation energy for success is beyond the potential contributor's threshold, then he chooses not to join.

Netscape makes for a good case study to understand the power of modular design in open source software. In 1998, in an attempt to regain the initiative in the Web browser market from Microsoft, Netscape released their browser to the public in the form of Mozilla. Unlike the germination states of Apache or Linux when they began, Netscape was not the product of one person's efforts. By the time the Netscape browser source code was released, it had already matured into a complex product worked on by numerous Netscape employees. This necessitated it to have a modular design. The phase system upon which the Apache API is based was used by Netscape for their server design before Apache was even conceived [21].

Since 1998, the Mozilla project has been a qualified success. It has maintained some hold in the browser market. However, it has not attracted widespread support, nor has it achieved more than a foothold in the market. Raymond believes that the most important reason for Mozilla's failure is economic. Mozilla's licensing agreement which was not truly open source because it required proprietary software to use [6]. However, Raymond also reports that one of the project's principals resigned because "going open will not necessarily save an existing project that suffers from ill-defined goals or spaghetti code or any of the software engineering's other chronic ills." This is an indictment of the core design. Perhaps the core structure of Mozilla, and the ill design of its modularity are indeed to blame.

The Microsoft and Intel collaboration, so called "Wintel" makes for another case study on the importance of modularity. While not an open source product, they benefited from an open standards design: the IBM PC. In a departure from their standard business practices, IBM in an effort to save money, the IBM PC was designed entirely off of off-the-shelf parts. IBM included only one proprietary component, the BIOS. However, since the rest of the design was of standardized parts, Compaq was eventually able to reverse engineer the BIOS, and hence the open standard IBM-compatible PC was born. This can be compared to the Apple Computer model that created from outward appearances, a monolithic system. Apple computers were designed and created by Apple who controlled rights to everything from the computer architecture to the operating system. This, in contrast to the virtuous synergy generated from the interplay between Microsoft, Intel, the numerous IBM-compatible makers, and the individuals who build their own systems. Though Apple was typically cited as a superior product, they eventually fell to the success of Wintel systems.

It is ironic that today, one major criticism of the Linux operating system and other open source projects is their lack of consistency. Today, Microsoft controls office productivity software by presenting a consistent monolithic interface that users are familiar with.

One of the radical proposed solutions to Microsoft's antitrust trial was to release Microsoft's software to the public. Their Windows operating system software most certainly has a modular design. It is already the product of a collaboration between thousands of people, the employees of Microsoft. Will releasing the software to the public make it succeed? Are the necessary economic institutions for its success as open source in place? The results of Netscape's open source experiment are unfortunately inconclusive.

6 Conclusion

There is much to be learned from studying the structure of open source design projects. The open source movement introduces a powerful concept into industrial organization. Some of its most ardent proponents foresee a future where all software is collaboratively created and distributed freely. However, even the most fanatic may miss the full implications that the open source movement may bring: a world where all companies become obsolete, and the users become the source of all innovation.

Acknowledgements

Thanks to Eric von Hippel for sharing his dream.

References

[1] "Netcraft Web Server Survey," [Online Document], (May 2000), Available HTTP: http://www.netcraft.com/survey.

[2] J. Lerner and J. Tirole, "The Simple Economics of Open Source," NBER Working Paper 7600, (Apr 2000), Available HTTP: http://www.nber.org/papers/w7600.

[3] Y. Hu, A. Nanda and Q. Yang, "Measurement, Analysis and Performance Improvement of Apache Web Server," *Performance, Computing and Communications Conference 1999* IEEE International , (1999) 261 –267.

[4] E. von Hippel, "Task Partitioning: An innovation process variable," *Research Policy* 19 (1990) 407-418.

[5] B. Perens, "The Open Source Definition," (retrieved May 2000), Available HTTP: http://www.oreilly.com/catalog/opensources/book/appb.html

[6] E. Raymond, "The Cathedral and the Bazaar," (retrieved May 2000) Available HTTP: http://www.tuxedo.org/~esr/writings/cathedral-bazaar

[7] N. Rosenberg, *Perspectives on Technology*, (Cambridge University Press, Cambridge, 1976).

[8] E. von Hippel, *Sources of Innovation*, (Oxford University Press, New York, 1988).

[9] H. Varian and C. Shapiro, *Information Rules*, (Harvard Business School Press, Cambridge, 1998).

[10] Lakhani and von Hippel, "Apache Help System," Unpublished working paper, Massachusetts Institute of Technology (2000).

[11] C. Fine and D. Whitney, "Is the make/buy decision a core competence?" Presented at MIT Symposium on Technology Supply Chains (May 10-11, 1995).

[12] *New hackers dictionary.* (retrieved May 2000) Available HTTP: http://earthspace.net/jargon/jargon_toc.html

[13] B. Liskov, *Program Development in Java: Abstraction, Specification, and Object Oriented Design*, (1999).

[14] K. Kuwabara, "Linux: A Bazaar at the Edge of Chaos," *First Monday* Vol. 5 No. 3 (March 6 2000).

[15] R. Thau, Email record of Apache Group mailing list, (March 1995), Available HTTP: http://www.apache.org/mail/new-httpd/199504.gz

[16] R. Fielding, "Shared Leadership in the Apache Project," *Communications of the ACM* Vol. 42, No. 4, (April 1999).

[17] R. Thau, "Design considerations for the Apache Server API," From the proceedings of the Fifth International World Wide Web Conference, May 6-10, (1996).

[18] J. Preston, "Software Architecture of the Apache Web Server," [Online Document], (Jan 1999), Available HTTP: http://se.uwaterloo.ca/~je2prest/ap_architecture/tbl_contents.html.

[19] F. Brooks, *Mythical man month*, (Addison-Wesley, Reading, Mass., 1975).

[20] B. Boehm, *Software Engineering Economics*, (Prentice-Hall, Inc., Upper Saddle River, 1981), 59-62.

[21] Netscape server API documentation, http://home.netscape.com/newsref/std/server_api.html, 1995.

Appendix A: The Open Source Definition

(Version 1.7)

Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:

1. Free Redistribution

The license may not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license may not require a royalty or other fee for such sale. (rationale)

2. Source Code

The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost -- preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed. (rationale)

3. Derived Works

The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software. (rationale)

4. Integrity of The Author's Source Code.

The license may restrict source-code from being distributed in modified form *only* if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software. (rationale)

5. No Discrimination Against Persons or Groups.

The license must not discriminate against any person or group of persons. (rationale)

6. No Discrimination Against Fields of Endeavor.

The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research. (rationale)

7. Distribution of License.

The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties. (rationale)

8. License Must Not Be Specific to a Product.

The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution. (rationale)

9. License Must Not Contaminate Other Software.

The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software. (rationale)

Conformance

(This section is not part of the Open Source Definition.)

We think the Open Source Definition captures what the great majority of the software community originally meant, and still mean, by the term "Open Source". However, the term has become widely used and its meaning has lost some precision. The **OSI Certified** mark is OSI's way of certifying that the license under which the software is distributed conforms to the OSD; the generic term "Open Source" cannot provide that assurance, but we still encourage use of the term "Open Source" to mean conformance to the OSD. For information about the **OSI Certified** mark, and for a list of licenses that OSI has approved as conforming to the OSD, see this page.

Change history:
1.0 -- identical to DFSG, except for addition of MPL and QPL to clause 10.
1.1 -- added LGPL to clause 10.
1.2 -- added public-domain to clause 10.
1.3 -- retitled clause 10 and split off the license list, adding material on procedures.
1.4 -- Now explicit about source code requirement for PD software.
1.5 -- allow ``reasonable reproduction cost" to meet GPL terms.
1.6 -- Edited section 10; this material has moved.
1.7 -- Section 10 replaced with new "Conformance" section.

Bruce Perens wrote the first draft of this document as 'The Debian Free Software Guidelines,' and refined it using the comments of the Debian developers in a month-long e-mail conference in June, 1997. He removed the Debian-specific references from the document to create the 'Open Source Definition.'