Learning from Linux

Internet, innovation and the new economy

Part I: Empirical and descriptive analysis of the open source model

Working paper, April 15, 2000. Please do not quote without permission.

Ilkka Tuomi

ituomi@uclink4.berkeley.edu

Abstract:

This paper describes some key principles that underlie development of the Linux operating system, and discusses their implications for innovation research. Using Linux as a case example, I will discuss organizational, institutional, economic, cultural, and cognitive aspects of the open source development model and technological change.

Introduction

According to user surveys, the Linux operating system is rated as the best operating system available. It is considered to be more reliable than its main competitors. Its functionality is claimed to be better, and according to many experts, new releases of Linux implement innovative ideas faster than its competitors. In other words, it is argued that Linux development creates complex new technology better and faster than the biggest firms in software industry.[1] Yet, Linux also seems to break many conventional assumptions that underlie research on innovation and technological change. Linux is developed by an informal self-organizing social community. There is no well-defined market or hierarchy associated with it. Most of Linux development occurs without economic transactions. Instead of getting paid for their efforts, the developers often spend a lot of money and effort to be able to contribute to the advancement of the development project.

The open source development model, which underlies Linux, has attracted increasing attention in the last years. Today, Linux is considered to be a serious threat to Microsoft's market dominance in operating systems. More generally, open source development projects have in recent years had major impact in software and internet-based industries. For example, over 60 per cent of Internet connected Web servers were open source Apache servers in March 2000. As can be seen from Figure 1, the second most popular Microsoft servers were about one third as popular with 21 per cent. Some open source projects, such as Sendmail and Emacs, have achieved large market shares, making it difficult for commercial enterprises to enter the market. The Internet Engineering Task Force, which defines standards for the Internet, has also used an open source approach since its formation in 1986 (Bradner, 1999). Several commercial software firms have tried to adopt aspects of the open source model. For example, Netscape announced in 1998 that it would distribute the source code of Netscape communicator with open source

[1] http://www.uk.linux.org/LxReport.html

license. IBM decided to use the open source Apache server as the core of its Web server offers. Red Hat, in turn, thrives on packaging Linux distributions, and producing added value for Linux users. In all these cases, business firms are experimenting with ways to benefit from innovation that occurs in the open source communities. Instead of traditional economic competition, such initiatives rely on symbiotic relationships, and on the willingness of developer communities to collaborate.

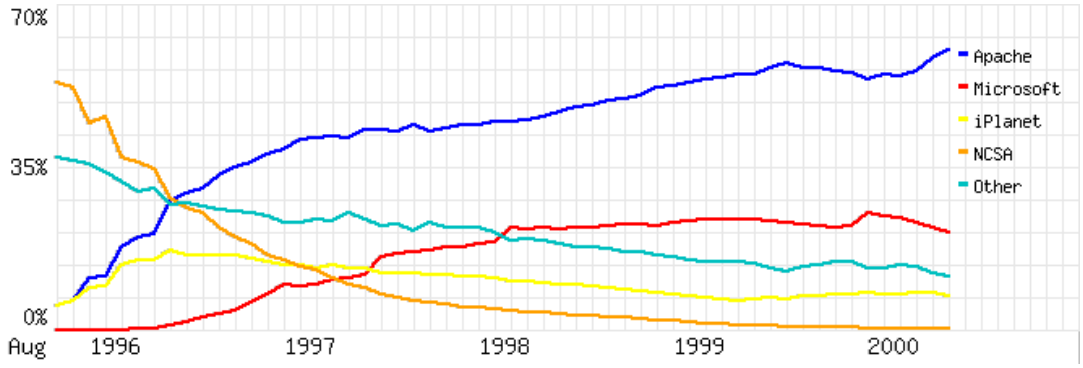

Figure 1. WWW servers connected to the Internet.[2]

This paper focuses on the Linux operating system kernel. I will describe its history and development model to discuss the organization and drivers of innovation in modern global economy. As the paper is heavily based on one specific software development project, I will also discuss the possible ways the learnings from this specific case may be generalized. A subsequent paper will further continue this work by linking the empirical results of this paper to the theory of innovation, research on product development, and technology policy.

In much of the innovation literature, innovation is defined as something that has economic impact. The case of Linux shows that this definition is problematic and possibly misleading in important practical cases. During its history, most Linux development has occurred independent of direct economic concerns. It would be tempting to argue that Linux development is different from “economic activity” and something that, strictly speaking,

---

[2] Source: Netcraft, http://www.netcraft. com/survey/.

should not be called innovation. Indeed, Linux development has not in any obvious way been associated with changes in production functions, market competition, or appropriation of economic investment and surplus. This, in itself, makes Linux an interesting test case for economic theories of innovation and technology development. For example, the case of Linux allows one to question to what extent existing economic models of innovation and technological development capture phenomena that underlie collective production of new technologies.

In very practical terms, Linux is an economically important phenomenon. Indirectly, the success of many new businesses, venture capitalists, investment funds, and individual investors critically depends on the productive activities of the Linux community. Yet, when we consider the entire history of Linux, the economic impact seems to appear almost as an afterthought and as a side effect of a long period of technology creation, in a way that seems to break commonly accepted rules of innovation and technology development. Linux, therefore, provides an interesting history of globally networked innovation, illustrating the substance that underlies the discussions on the “new economy.”

From a theoretic point of view, Linux is an interesting case as it enables us to discuss those social and cognitive phenomena that underlie technological change. In that sense, it enables us to penetrate some black boxes of innovation theory, including such widely used concepts as learning, capability, utility, and consumption. By observing the development of Linux, we can describe the microstructure of innovation, and transcend the boundary between invention and innovation. Linux development is collective productive activity, and the products of this activity are externalized as technological artifacts and discourses, which can be observed relatively easily. There exists sufficient documentation on the history and practices of Linux development so that we can—at least tentatively—describe some key principles that underlie Linux development.

From a practical point of view, the case of Linux also provides a test case for analyzing product development models and proposals for organizing for innovation. Specifically, the extensive use of modern communication and collaboration technologies in Linux

development highlights some aspects of technology development that were not easy to see in earlier studies on innovation.

This paper is organized as follows. First, I will briefly describe the Linux system and its developer community in an evolutionary context, highlighting some main characteristics of the socio-technical change that has led to the current Linux system. I will then discuss the organization of this technology creation process, focusing on control and coordination mechanisms. I will describe in some detail the ways the Linux community has managed the trade-offs between innovation and maintainability of the increasingly complex system, and discuss how the learnings of effective coordination and control mechanisms have been embedded in the system architecture.

Linux has attracted considerable attention because it has been argued that the open source quality control mechanisms are more effective than traditional methods used in software development. It has often been argued that Linux is more reliable than proprietary systems because it is developed using the open source principles. I will describe the quality control system, analyzing in some detail the Linux bug removal process and the complex socio-technical system that underlies it.

As was noted above, innovation literature sometimes leaves the process of invention into a black box where undefined psychological forces operate outside the domain of innovation research. The drivers for innovation are commonly understood to be economic. In this context, it is interesting to analyze the incentives and drivers of technology development, as they can be observed in the case of Linux. I will do this, describing the reputation and attention management processes that underlie Linux. Reputation and attention are shown to be closely related in the Linux community, and they are the key to resource allocation, which, in turn, directs technology development.

Open source development is a special form of technology development as it intentionally reverses some common intellectual property rights. Instead of copyright it uses “copyleft,” which guarantees the rights of users to modify the results of development, and derive new works from it. The fact that such a licensing model seems to work and promote

technology development has important consequences for discussions on intellectual property rights, patent system, and the theory of appropriation of the results of innovation. The open source licensing policy can be seen as an important social innovation that has major impact to the way Linux is developed. I will discuss regulations and standards that underlie Linux development, focusing on the various forms of licensing that have been used in the open source community.

Finally, I will conclude by drawing together some results from this descriptive and empirical analysis, putting them into the context of the global dynamics of modern economy and innovation.

The present case study of Linux development covers a broad set of issues. It is, of course, impossible to deal with all those issues in the depth they deserve within a single paper. The goal of this paper is to provide a rich enough description of the case for further discussions, as well as to open the area for more detailed theoretical and empirical study.

Evolution of Linux and its developer community

Linux development started in 1991 when Linus Torvalds got a new Intel 386 PC and wanted to learn it. In the beginning, Torvalds didn't expect that anyone would use Linux. It was, however, developed to be compatible with widely used Unix tools, and its source code was made available for anyone who was interested. As a result, people who wanted to have a Unix-like operating system on their Intel-based PC's quickly adopted Linux and started to add new functionality to it (Torvalds, 1999).

Linux was inspired by a small Unix-like operating system Minix, and many early adopters were familiar with Minix. Minix had been developed by professor Andrew Tanenbaum to teach operating systems for students who had only the first generation PC's available. Whereas Minix was intended to introduce the basic theoretical concepts of operating system design, Linux was a more pragmatic project. The goal was to develop an operating system that worked well on Intel 386, and which users were free to modify and play with (DiBona, Ockman, & Stone, 1999: 221-51). The first version of the system was release

0.01, September 1991. Although it is difficult to find accurate data on the usage of Linux, today there are probably over 12 million Linux users worldwide.[3] Indirectly, almost all people who are connected to the Internet use Linux, as many Web-servers rely on it.

Linux, and its open source development model, started to attract attention around 1994. Until that time the Berkeley BSD Unix had been the most visible open source development activity. It was generally believed that the era of Unix-based operating systems was over, and that Microsoft had secured its position as the dominant player in the operating system market. As an indication of this the Berkeley Unix development group was formally shut down (Raymond, 1999:22-3). The success of Linux became as a surprise to its developers, but also to people who had been closely observing the evolution of software and open source projects. In his influential article[4], Eric Raymond describes how Linux made him realize that there exists a new mode in software development:

> Linux overturned much of what I thought I knew. I had been preaching the Unix gospel of small tools, rapid prototyping and evolutionary programming for years. But I also believed there was a certain critical complexity above which a more centralized, a priori approach was required. I believed that the most important software (operating systems and really large tools like Emacs) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.
>
> Linus Torvalds's style of development—release early and often, delegate everything you can, be open to the point of promiscuity—came as a surprise. No quiet, reverent cathedral-building here—rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from *anyone*) out of which a coherent and stable system could seemingly emerge only by a succession of miracles.
>
> The fact that this bazaar style seemed to work, and work well, came as a distinct shock. As I learned my way around, I worked hard not just at individual projects, but also at trying to understand why the Linux world not only didn't fly apart in

---

[3] http://www.linux.org/info

[4] http://www.tuxedo.org/~esr/writings/cathedral-bazaar/ . The article is also included in Raymond (1999).

> confusion but seemed to go from strength to strength at a speed barely imaginable to cathedral-builders. (Raymond, 1999:29-30)

Linux is a fast growing system. The core Linux—the operating system kernel—consists of software that controls computer hardware and programs that run on it. When new interesting hardware becomes available, the operating system kernel is extended for it. Usually, Linux code for specific hardware components is developed as "drivers." Linux is available for several different processor architectures and therefore there also exists several "ports" of the system.

An operating system can be built based on several different architectures. The architecture of Linux has been strongly influenced by the Unix operating system. Unix architecture is implemented as layers, where each layer provides service to the layer above it (Tanenbaum & Woodhull, 1997). The bottom layer interfaces the software with hardware. A layered operating system can be represented as in Figure 2. The system kernel usually takes care of process, memory, file, security, network, and input and output device management. Utility or system programs are applications that provide key services that are needed for a functional operating system. In a Unix system, graphical user interfaces, user management, command shell, file backup, and, for example, directory listing programs are examples of such system utilities. These programs use the functionality provided by the kernel by calling system functions through the system call interface. The various end-user applications, such as word processors, database management systems, and web browsers, can use both utility programs and system calls to interface with the operating system. The operating system kernel, in turn, uses the underlying hardware through hardware-specific drivers that convert operating system calls into low-level hardware programs.

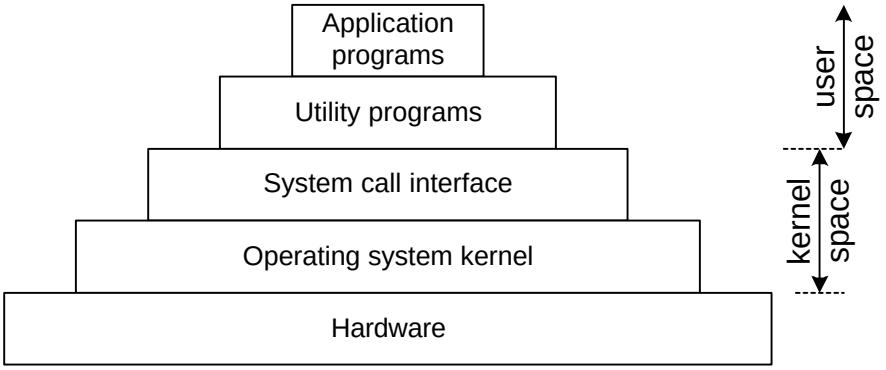

Figure 2. Layers of a Unix operating system.

An operating system kernel is not a very useful thing in itself. A fully functional operating system needs system utility programs and applications before it becomes a multipurpose platform that can support and run end-user applications. In practice, Linux relies on a large set of other open source programs to form a fully functional operating system. Most critical of these are the GNU c-language compiler and the GNU c-libraries, which are required for developing the system. A distinction is often made between the Linux operating system itself, and the set of open source applications, including the kernel, that together make a functional environment. The operating system kernel is usually called Linux and the complete system is called GNU/Linux.

The end-users of Linux mainly deal with large software distributions that comprise hundreds of applications in addition to the operating system. For example, the Debian distribution of GNU/Linux has over 1500 open source programs, including word processors, graphics programs, databases, and web-servers and clients.[5] The evolution of Linux-based systems is therefore only loosely coupled with the evolution of Linux itself. For example, the functionality of GNU/Linux has grown considerably when major software providers have recently started to port their systems for Linux.

---

[5] http://www.debian.org

In this paper, I will focus on the development of the Linux operating system kernel. Already in itself, it provides an interesting example of technology development. Since the first release of Linux, there has been about one new version of the system released every week. During this same time, the total size of the kernel code has grown from 236,669 characters in the distribution files to over 78 million characters. In other words, the code size has grown 333 times.[6] The different versions and their relative sizes are shown in Figure 3. The figure shows sizes for the compressed kernel source code packages. The actual code size is typically about four times larger.

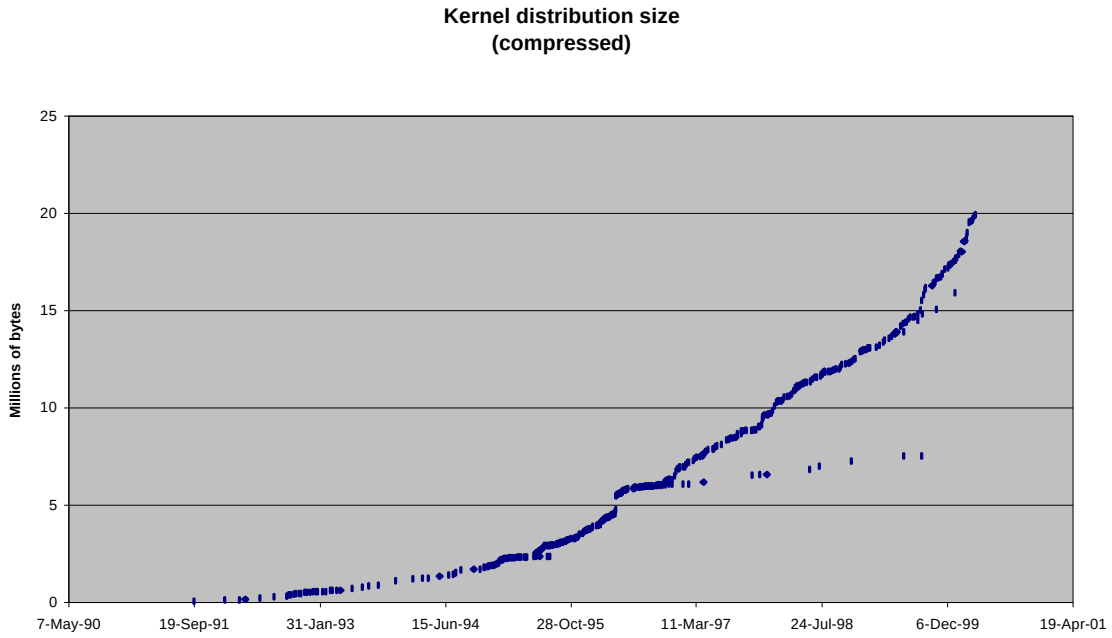

Figure 3. Linux kernel distributions.

From Figure 3 one can note one of the key characteristics of Linux development. The kernel releases are divided into “stable” and “developmental paths. In practice, the releases are numbered using a hierarchical numbering system where the first number denotes a major version, and the second number gives the version tree in question. In the recent

[6] For the purposes of this paper, I will measure code size in characters, counting comments and documentation as parts of the source.

years, the even numbered trees have been stable production releases, and the odd numbered trees have been "developmental" releases, where new features and functionality is introduced and tested. For example, in Figure 3 release paths for versions 2.0.x and 2.1.x create two distinctive paths. The paths fork when version 2.1.0 was introduced in September 1996, and when a new developmental path was started. New releases of the stable path are released simultaneously with developmental releases, but usually only with minor bug fixes. Indeed, the last version of the stable path, 2.0.38, was released about three years after the developmental path 2.1 started. The developmental path 2.1, in turn, consisted of 132 versions before it became the next stable version 2.2., at the beginning of 1999.

One of the characteristics of open source software projects is that the system design evolves based on ongoing innovation and learning. One way to illustrate this ongoing innovation is to analyze the increasing complexity of Linux during its history. The structural complexity of the system is reflected in the number of relatively independent subsystems. In practice, the code for each subsystem is organized into its own subdirectory. An estimate of the number of subsystems can therefore be found by counting the subdirectories in the kernel distribution. Figure 4 shows the number of new subdirectories created within two-week time windows, as well as the number of subdirectories in use across time.[7] Major peaks in the number of new directories indicate a major rewrite of the system. This happens when a new major version is released. On average, there were 1.6 new directories created each two weeks. In release 2.3.51, March 2000, there were 333 subdirectories in use.

---

[7] The graph was produced by analyzing the file creation dates for 11 kernel releases, including 0.01 and 2.3.51, and defining the directory creation date as the date the first file in the directory was created. A moving 14 -day time window was used, starting from the first file creation date found. The total number of files was about 20,000. The analysis was done using a set of Perl programs that processed the file lists of the various releases and counted the number of new directories within each time window. This rather labor-intensive process was used as some of the directories in the kernel archives had been recreated during the years, and therefore had lost their original creation dates.

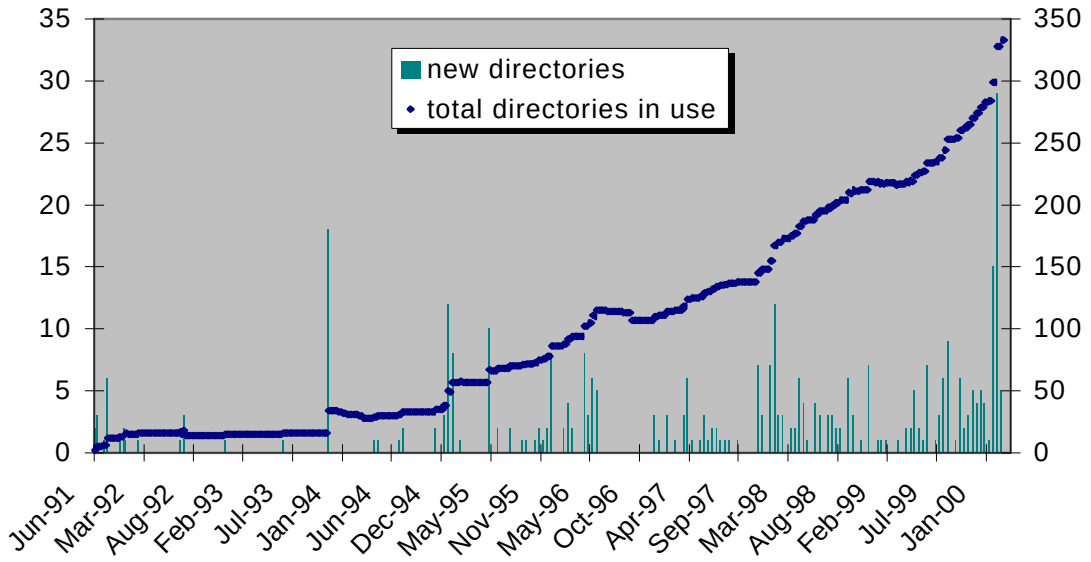

Figure 4. New source code directories.

Using a similar measure, it is also possible to estimate the intensity of "creative destruction" in Linux development. On a structural level, this can be viewed as the number of system components that become obsolete within a given time window. Using the directory structure as a proxy for this, we can count the number of directories that disappear within a given time window. This is shown in Figure 5.

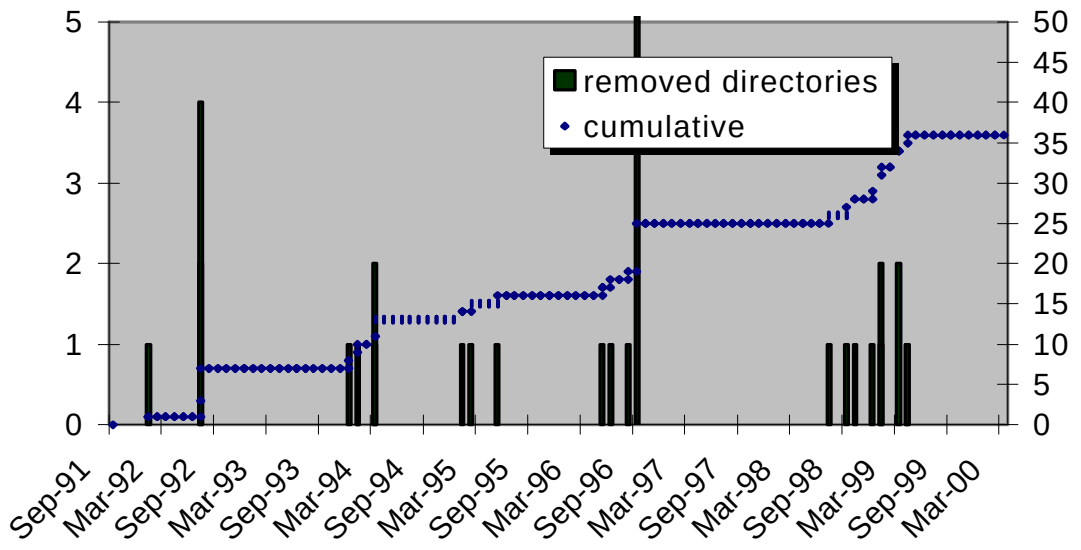

Figure 5. Removed directories as a function of time.

Already from the brief outline given above, it is easy to see that the Linux development model has led to continuing system development. Even within the kernel itself, the rate of technology creation seems to increase as the development proceeds. Although the system has gone through a large number of revisions, the rate of growth does not seem to slow down.

Linux developer community

Constant innovation creates major challenges for developing a coherent and maintainable system. When a number of people actively develop the same system, and thousands of end-users can freely report bugs they find and express their ideas for new functionality, there is an ongoing flow of suggestions for improvement. This easily leads to an increasingly complex system that becomes extremely difficult to understand and maintain. In the Linux development community, this phenomenon is known as “creeping featurism,” and it is one of the main concerns of the developers. Yet, it is also important that new innovations are incorporated into new releases without excessive delay. Without the possibility of new contributions to be integrated into the system, there would be little point in proposing and producing improvements. In practice, this inherent tension between the

need to incorporate new innovations and keep the system complexity manageable is a critical issue for open source development. A successful resolution of this issue requires effective social coordination and control. The resulting social structures and processes, therefore, reflect the requirements of successful system development. To the extent that Linux is a highly reliable and effective software system, one could then expect that its developer community implements effective social structures for technology development.

Since version 1.0, March 1994, Linux kernel files have included a "credits" file that lists important contributors to the project. The most recent credits file for Linux contains the names of 328 developers.[8] This is a good estimate of the number of people who have substantially and successfully contributed to the development of the core Linux system. The actual number of co-developers is, however, much higher. There are about 90,000 users who have registered themselves as Linux users[9], and a large proportion of these have programmed at least minor applications for Linux. These active developers are an important source of bug reports and bug fixes. Often the credit of such contributions is given only in the change logs and in source code comments. The "bazaar" described by Raymond, therefore, seems to consist of several hundreds of central members, and several thousands of more peripheral, but technically sophisticated users.

One important aspect of this "bazaar" is that it relies heavily on Internet to get its work done. The Linux development model emerged simultaneously with the explosion of Internet use. In early 1992, it was still argued that the development model relied too much on Internet, therefore excluding people without Internet access (Tanenbaum, quoted in DiBona, Ockman, & Stone, 1999:245). However, the rapid expansion of Internet use at the time when the Linux kernel was developed provided the developer community with new ways to distribute development work, a new distribution channel, and a global community of sophisticated users.

---

[8] ftp://ftp.funet.fi/pub/Linux/kernel; the CREDITS file can be found in the root directory of each release

[9] http://www.linux.org/info/index.html

The regional distribution of early Linux development work is depicted in Figure 6. The figure shows the number of people in different countries mentioned in the first credits file. To adjust for the different sizes of countries, the numbers in Figure 6 are given per million inhabitants. The figure shows that Linux development has been a geographically broadly distributed activity since the very beginning.

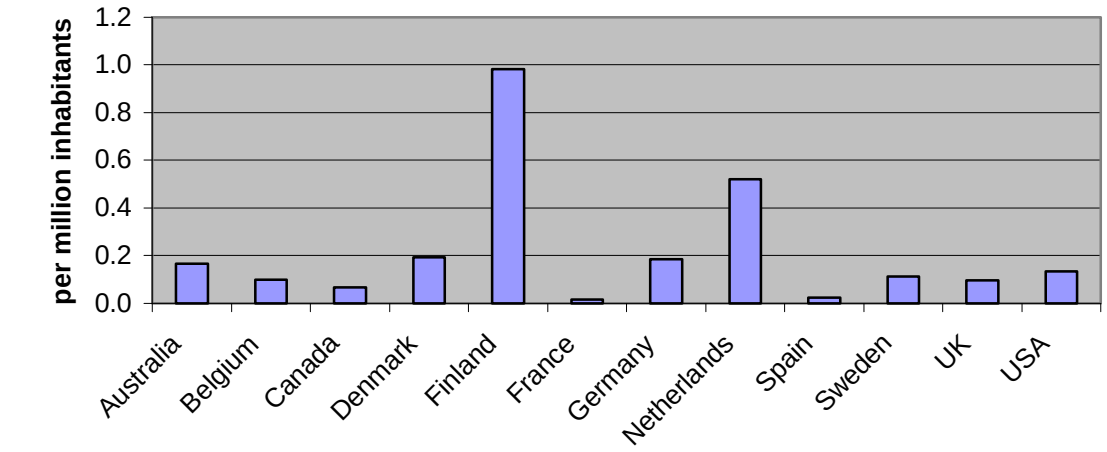

Figure 6. Location of active contributors, March 1994.

The first credits file acknowledged 78 contributors coming from 12 different countries. In addition there were two contributors whose location was not possible to identify using the information in the file. The credits file for 2.3.51 release, March 2000, had contributors from 31 different identifiable countries. In absolute numbers, the U.S.A. was the biggest home base for contributors with 114 people. Figure 7 shows the current geographic distribution of people in the credits file. Luxembourg had one developer in the most recent credits file, but as the country has less than half a million inhabitants, it is omitted from Figure 7.

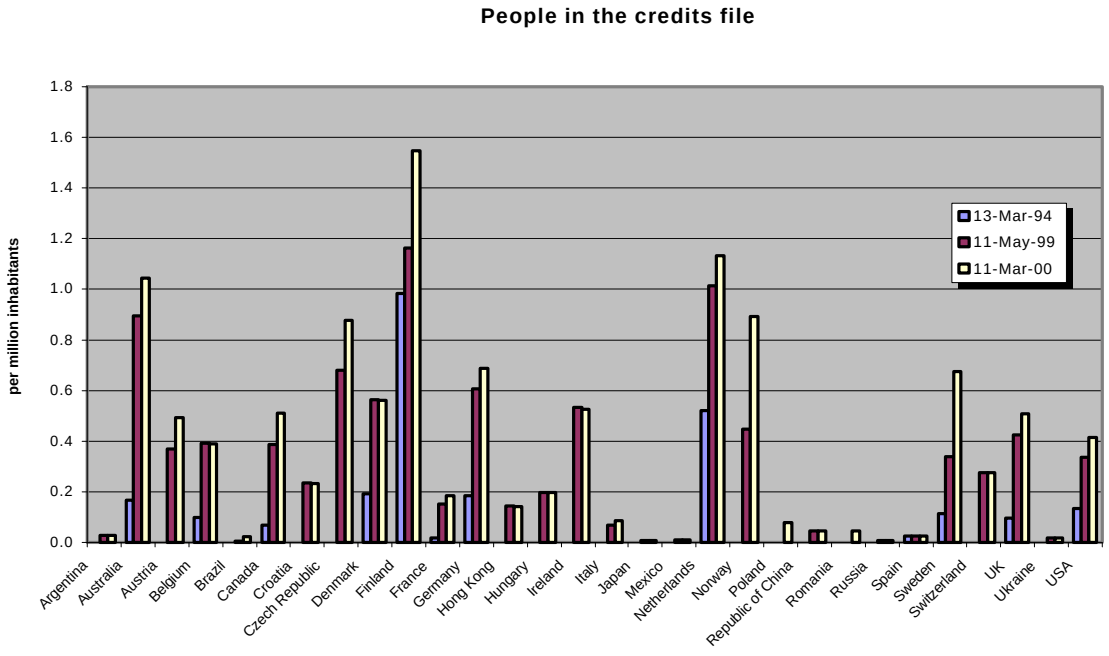

Figure 7. Geographical expansion of development activity.

Linux is in many ways a self-organizing effort. There is no formal organization, although several non-profit and business organizations have become important components of the Linux development effort during the recent years. The Linux development effort is in practice organized around projects, communication procedures, communication and collaboration tools, and software modules that are constantly evolving. In many ways, the Linux development community resembles a community of practice (Lave & Wenger, 1991; Brown & Duguid, 1991). Such a community is organized around central "gurus," "old-timers," and more peripheral novices who have been accepted as legitimate members of the community. In the case of Linux, the core community members consist of key contributors to the overall project. In contrast to the basic community of practice model, however, Linux development community has more than one center, as there are several important sub-projects. In recent years, the different main subsystems have been managed by a self-nominated group of maintainers. As a default, Linus Torvalds acts as the maintainer for those subsystems that have no explicitly defined maintainer.

Although Linux development is not formally organized, a key to the success of Linux is that the development is not random, however. The development of technology is based on a sophisticated system of social relations, values, expectations, and procedures. This system is in many ways quite different from conventional industrial product creation.

Control and coordination in the Linux community

Traditionally, organization theorists have argued that increasing complexity in division of labor leads to formal organizational structures (e.g., Mintzberg, 1979). In the case of Linux, this doesn't seem to be the case. Although the Linux community has some structural similarity with the cellular organizational form (Miles, Snow, et al., 1997) and the hypertext organization (Nonaka & Takeuchi, 1995), existing organizational models do not very well describe the structure of Linux community. Instead, the Linux developer community resembles a dynamic meritocracy, where authority and control are tightly bound with the produced technological artifacts. In this sense it also differs from most network-based organizational and innovation models, which typically have focused on firm and industry level networks (e.g., Powell, Koput, & Smith-Doerr, 1996; Van de Ven, 1993; Lynn, Aram, & Reddy, 1997). Indeed, the organizational structure could be characterized as a network of communities of practice, or as a fractal organization (Tuomi, 1999).

Peripheral additions to the GNU/Linux system are not controlled by anyone. For example, anyone can develop a new application that uses the Linux kernel, and distribute it. As a result, there exists a large set of potential sub-projects competing for community development resources. As will be discussed below in more detail, the allocation of these resources is to a large extent based on management of community attention, which in turn relies on accumulation of reputation. Sometimes it is possible to develop a simple program for one's own use, but in most cases interesting systems require that several developers become interested in developing and using them. Control, therefore, is indirectly based on capability to mobilize resources. Directly, it lies very much with the users and potential co-developers.

As there is no formal organization in the Linux community, its coordination and control mechanisms can only be analyzed by observing those explicit and implicit procedures that the community relies on. In the Linux development community, social issues are often described as technical issues. When Linux developers discuss the way the system should be developed and how it should evolve, discussions often focus on code portability, maintainability, possible forking of code-base, programming interfaces, and code size and performance. These technical discussions are critical for the success of the collaborative effort.

Implicitly, each technical choice implies specific procedures that need to be followed in the development work. Often the technical decisions are driven by the need to keep the collaborative development work going. Computer software is inherently flexible, and there is a very large set of possible ways to implement a specific system. Technical decisions, therefore, are to a large extent articulations of beliefs on the effective ways to organize development. In the course of the evolution of the system architecture, the learnings on problems and possibilities of collaborative development become implemented in the architecture of the technological artifact. Technical discussions on how things “work,” what a good design is, and how development should be done are therefore often reflections on social practices, externalized as specifications for technology. This is most obvious when developers discuss the maintainability of the code, “cleanness” of the interfaces, and the problem of “creeping featurism.”

A contingency theoretic view would imply that in a successful development project, such as Linux, the structure of software becomes a mirror image of important aspects of the social structure that is needed for a successful system to emerge. Although social institutions are not directly mapped to the development architecture, of course, the architecture and ways of doing things have to be complementary for a project to be successful. For example, the control of specific software modules, coordination mechanisms, incentives, and goals have to be aligned for the overall effort to be successful. These, in turn, have to be embedded within a larger social context, which limits the arrangements within the Linux community.

The lack of formal organizational structure in Linux development has enabled flexible experimentation with the procedures and values that support effective development. As Linux development occurs in “internet time,” the speed of evolution is fast. The resulting social innovations, therefore, crystallize some of the learnings in organizing collaborative and geographically distributed technology development.

Collaborative software development projects have inherent problems that create specific forms of division of labor, and related design traditions. A sociological description might view the emergent social structures in the Linux development community as solutions to underlying social tensions. Blumenberg 1985, for example, argued that social institutions grow around irreducible social contradictions and fundamental conflicts, somewhat as a pearl grows around an irritating grain of sand. A fundamental problem in the development of complex software is that small modifications in one part of code can have major implications for another part of the code. There is no natural decay in software, and therefore no universal dimension of distance or time. As Wiener (1975) noted long time ago, digital computers are unique among computational systems because digitalization makes computational machines noiseless, in the information processing sense. Modularity and “locality,” therefore, have to be created and maintained through social processes.

In practice, interdependencies in pieces of parallely developed code create a need to coordinate design decisions. Often there are conflicting interests. For example, a minor modification in some part of the program code may require major rework from people who maintain other parts of the program. A generic way to reduce this problem is source code modularization. A well-designed program has modules whose design limits interactions between modules. If modularization is successful, one programmer can modify the source code of his or her module without requiring changes in other modules. In other words, a programmer can control the evolution of a specific piece of code, without creating problems for other programmers.

In the case of Linux, modularization is based on social agreements, which are supported by commonly accepted development practices, and which are reflected in the overall

system design. Many of these social agreements are implicit, and community members have to learn them through socialization. Indeed, only after a novice programmer is able to display the mastery of the key rules, he or she is considered to be a full member of the community. To some extent these rules are also dynamic and they can change.

Sometimes conflicts arise about the implementation and functionality of a specific program module. If two programmers create different versions of the module, and the module is a key component of the system, this leads to forking of the code base. In effect, the system evolves into two different and incompatible variations. This means, in practice, that the synergy in development is lost, and the developers have to choose one of the versions as the basis for their future work. According to Torvalds, such code forking occurred in the first attempt to port Linux to a non-Intel processor architecture. As a result, the kernel design was modified to accommodate new processor architectures in a way that did not risk forks in the code base (Torvalds, 1999:102).

The design choices for modularization and interfaces are critical success factors in collaborative software development. It is possible to define modules so that development becomes extremely difficult. For example, if there is no simple mapping between the underlying hardware and the software, the implementation of new functionality may require changes in several modules. Similarly, a small modification in the user interface may require extensive reprogramming if the modularization is bad. The layered abstract architecture of Unix is one attempt to solve this problem. In practice, this leads to major challenges in finding the appropriate levels of system abstraction, which are then reflected in the structure of source code. The situation is made worse by the fact that programmers often want to by-pass some levels of the abstract system architecture, usually to improve performance. Often it means that abstract representations of the system only remotely resemble its concrete implementation.

For example, Linux modules that support different networks should in theory be independent modules. In practice, there have been many interdependencies between the modules for different networks. Armstrong (1998) has used automatic architecture

extraction tools to analyze Linux, and noted that these interdependencies create a potential maintenance problem for the kernel.

Although GNU/Linux development is open and has very little formal control, kernel development is socially very tightly controlled. It is in the very core of the Linux kernel, therefore, where the link between social and technical structures can most clearly be seen.

Controlling the kernel

The constant flow of improvements means that the Linux system is in a constant risk of losing its maintainability. In practice, balancing innovation and maintainability has led to tight control of some parts of the system. The control structures are very dynamic and continually reproduced in the ongoing communication within the developer community. As Linus Torvalds notes in one recent email:[10]

> If anybody thinks that being the maintainer equals being in 100% control, then I don't think they have understood the TRUE meaning of Open Source. Open source is about letting go of complete control. Accept the fact that other people are wonderful resources to fixing problems, and let them help you.

To study the interplay between control and technology design, we need to describe the architecture of the Linux kernel. As was noted before, a GNU/Linux distribution consists of a large set of application programs, the basic Unix utility programs, several versions of the kernel for the different supported processor architectures, and a large number of drivers for different types of hardware. In practice, the abstract high-level system architecture shown in Figure 2 is therefore relatively close to the actual Linux implementation. On a more detailed level, abstract descriptions, however, start to deviate from the concrete implementation. The kernel, for example, does not have a well-defined boundary between the system call interface and the core kernel. This is partly because

---

[10] Summarized from email traffic in linux-kernel mailing list in Kernel Traffic #60, 27 Mar, 2000, http://kt.linuxcare.com/

there are performance trade-offs, which sometimes make it practical to bypass some internal parts of the kernel. Partly it is simply because the evolution of Linux has led to interactions between the different parts of the system, and, as a consequence, the boundaries have become blurred. Also, in Linux the module called “kernel,” which architecturally most closely resembles the system call interface, implements some process management and memory management functions, as well as some error processing. The main components of the Linux kernel architecture can be represented as in Figure 8.

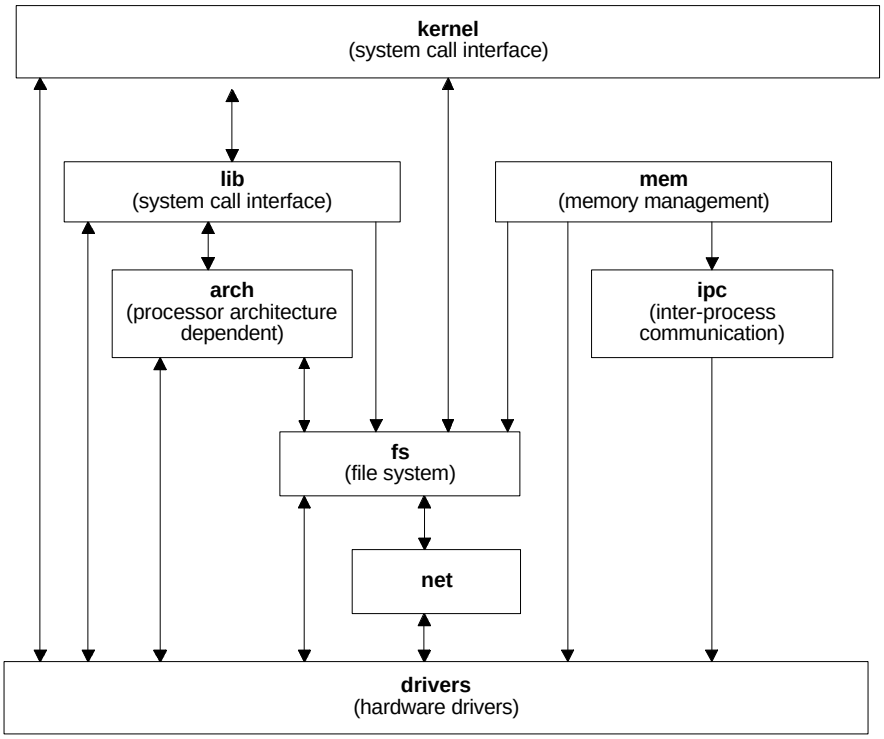

Figure 8. Linux architecture (modified from Armstrong (1998)).

The need to control the kernel was one of the topics in the famous debate between Andrew Tanenbaum and Linus Torvalds in 1992. Tanenbaum argued that it is critical for a successful operating system project that someone maintains tight control of the code, so that its complexity does not explode and that the core of the system does not fork:

> If Linus wants to keep control of the official version, and a group of eager beavers want to go off in a different direction, the same problem arises. I don’t think the copyright issue is really the problem. The problem is co-ordinating things. Projects

> like GNU, MINIX, or LINUX only hold together if one person is in charge. During the 1970s, when structured programming was introduced, Harlan Mills pointed out that the programming team should be organized like a surgical team—one surgeon and his or her assistants, not like a hog butchering team—give everybody an axe and let them chop away. Anyone who says you can have a lot of widely dispersed people hack away on a complicated piece of code and avoid total anarchy has never managed a software project. (quoted in DiBona, Ockman, & Stone, 1999:247)

At that time, Linus emphatically argued that he would not control the system:

> This is the second time I've seen this "accusation" from ast (Andrew Tanenbaum)…Just so that nobody takes his guess for the full truth, here's my standing on "keeping control", in 2 words (three?):
>
> I won't.
>
> The only control I've effectively been keeping on linux is that I know it better than anybody else, and I've made changes available to ftp-sites etc. Those have become effectively official releases, and I don't expect this to change for some time: not because I feel I have some moral right to do it, but because I haven't heard too many complaints, and it will be a couple of months before I expect to find people who have the same "feel" for what happens in the kernel. (quoted in DiBona, Ockman, & Stone, 1999:247)

Almost seven years later, at the end of 1998, Torvalds argued that the development had undergone major improvement when a new model for the kernel development was taken into use with release 2.0. In the new kernel architecture, the original monolithic kernel was extended by introducing loadable kernel modules. These are mainly used to dynamically load device drivers according to the needs of the specific computer configuration. According to Torvalds, this improved modularity by creating a well-defined structure for writing modules:

> Programmers could work on different modules without risk of interference. I could keep control over what was written into the kernel proper. So once again managing people and managing code led to the same design decision. (Torvalds, 1999:108)

An indirect implication of loadable kernel modules is that performance critical hardware-specific code can often be confined to a module. As a result, the core kernel becomes easily portable. For example, low-level interaction with hardware devices can be

programmed in separate device drivers that can be loaded if the specific device is present. Such device specific functionality, therefore, does not need to be implemented in the core kernel itself. The question, however, is not only about technical performance. As Torvalds notes:

> But Linux's approach to portability has been good for the developer community surrounding Linux as well. The decisions that motivate portability also enable a large group to work simultaneously on part of Linux without the kernel getting beyond my control. The architecture generalizations on which Linux is based give me a frame of reference to check kernel changes against, and provide enough abstraction that I don't have to keep completely separate forks of the code for separate architectures. So even though a large number of people work on Linux, the core kernel remains something I can keep track of. And the kernel modules provide an obvious way for programmers to work independently on parts of the system that really should be independent. (Torvalds, 1999:109-10)

Control and innovation

Comparing Torvalds' early and later statements on controlling the system it is clear that both the challenges and the possibilities of the system have considerably evolved in a few years. In 1992, when there were only few people developing the system, there was no obvious need to restrict derivative works. Although Tanenbaum warned Linux developers about the problems of uncontrolled forking, at this time the developers were more interested in the possibility to easily modify and improve the system. Indeed, the system was perceived as a huge technical opportunity and there were no visible constraints that restricted its future evolution. In other words, it was seen as a platform that could easily adjust to the best technical ideas anyone could come up with. Due to its simple structure, open source, and constant improvements, it was able to effectively grab attention among programmers who wanted to show their capability, and who enjoyed the possibility of creating code that contributed to the collective effort.

When the system has become more complex and there have been more active developers, the problem has increasingly been in balancing coordination, control, and local innovation. The key factors in this process seem to be modularization and implicit management of attention. Attention is allocated to a large extent based on centrality in the community, and

this, in turn, is based on reputation. Reputation within the community is, in turn, to a large extent based on producing working code that has relevance for the community, or solving important problems with existing code.

Under these circumstances, reputation is a good predictor for future achievements. If someone has successfully coded Linux, he or she most probably knows many things about programming. A distributed system of social control seems to work effectively in this case, and the technical artifact and its developer community evolve in compatible ways. Although the skill-base and tools change, the open and collaborative environment makes it relatively easy to learn new techniques and renew competences. As a result, meritocracy has definite merit in Linux development. Innovation, however, is also closely related to the modularity and extensibility of the underlying technical system. The market of resource mobilization is possible in the Linux community only because the kernel architecture provides a relatively stable focal point around which dynamic sub-communities can emerge.

Using the directory structure of the Linux kernel, it is possible to illustrate the evolution of the various kernel components. Such an analysis reveals that the growth in the system is quite heterogeneous. Innovation and development are strongly concentrated on some parts of the system, whereas other parts rarely change. The core kernel may be defined as those parts of the system that have stabilized. Figure 9 shows the evolution of some components of the core kernel. A typical pattern is that a period of relatively rapid change is followed by stabilization and lock-in.

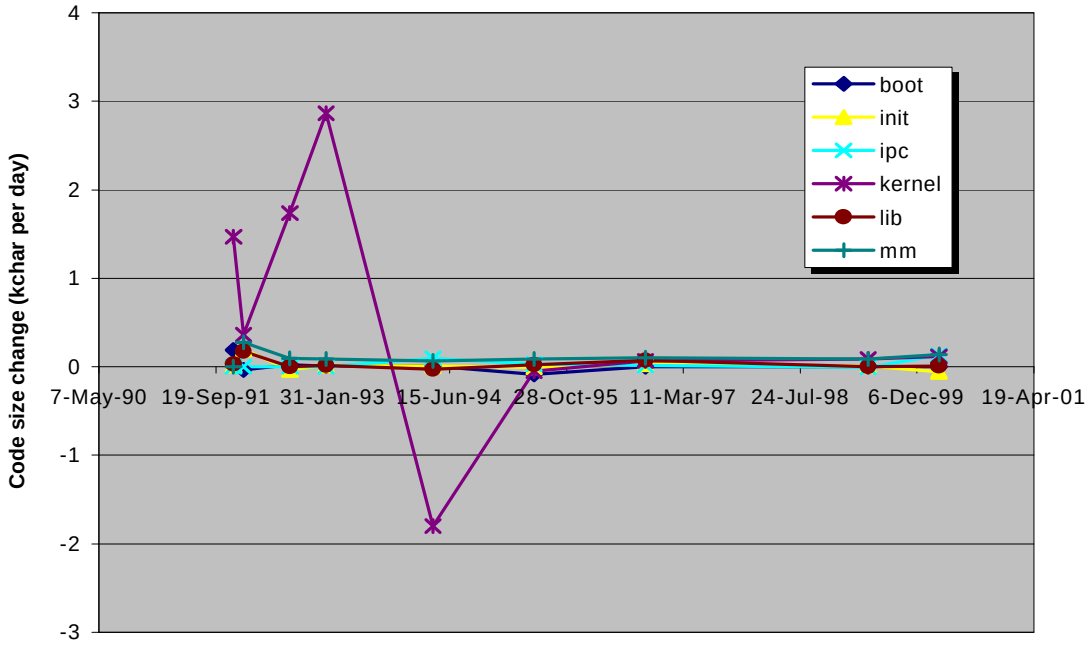

Figure 9. Growth of some core kernel components.

In most cases, changes in the core kernel would require extensive rewriting of other parts of the system that depend on given functionality in the core. Therefore such changes can happen only in limited ways. In practice, after there exists a substantial amount of code that depends on the core kernel, the core can only change its internal implementation or provide new functionality that is compatible with the old. Technically, it is not possible to constrain changes in such a way. In other words, the open source development model needs strong social controls to avoid the costs of lock-in. As Torvalds notes:

> The first very basic rule is to avoid interfaces. If someone wants to add something that involves a new system interface you need to be exceptionally careful. Once you give an interface to users they will start coding to it and once somebody starts coding to it you are stuck with it. (Torvalds, 1999:105)

The internal implementation changes in the core kernel when bugs are corrected or when inefficient code is rewritten. New functionality, in turn, is introduced only when there are

extremely important reasons for it. The conservative policy for the extensions originates from the fear that the core kernel becomes difficult to maintain, or that new bugs are introduced into the core. By keeping the core kernel simple and stable, and by providing support for extensions in the kernel architecture, technical change can be directed to areas where it can be managed. The fundamental trade-off is, of course, that radical innovation in the core kernel becomes difficult, or impossible in practice. The interactions between software modules make the system evolution extremely path-dependent, and tight control of some parts of the system are required to keep other parts of the system open and extensible.

As expected, in contrast to components shown in Figure 9, some other parts of the kernel grow very fast. Linux was originally developed for Intel 386 processor architecture, but since version 1.2.0 it has supported several alternative processor architectures. Linux can be extended to a new processor by porting its processor-dependent parts. The code for these different processor-dependent parts is organized into their own directories. Using such a modularization of source code, the code for different architectures becomes independent, and it is possible to add support for new processor architectures without interfering with other parts of the source code. In practice, this has led to a situation where several different teams of programmers have been able to develop the overall system in parallel.

Similarly, the hardware specific drivers can be developed as independent modules. Indeed, most of Linux development in recent years has been related to new hardware components. The open source policy makes it easy for anyone to develop hardware-specific additions to the system, as long as the developer knows the internals of the hardware in question.

Linux architecture is extensible also in areas that one would expect to be parts of the core kernel. For example, Linux supports a large selection of file systems. As long as existing file systems continue to work, it is possible to introduce a new file system without much risk of destroying system reliability. As Torvalds notes:

> Without modularity I would have to check every file that changed, which would be a lot, to make sure nothing was changed that would effect anything else. With modularity, when someone sends me patches to do a new filesystem and I don't necessarily trust the patches *per se*, I can still trust the fact that if nobody's using this filesystem, it's not going to impact anything else. (Torvalds, 1999:108)

As a result, the evolution of Linux is very much concentrated on those parts of the system that can be developed independently. This can be seen by analyzing the rate of change in the source code size. The main extensible components of the kernel distribution are shown in Figure 10. Comparing the rates of change for core and extensible components of the kernel, one can see that the extensible components grow typically about two orders of magnitude faster than the core components.

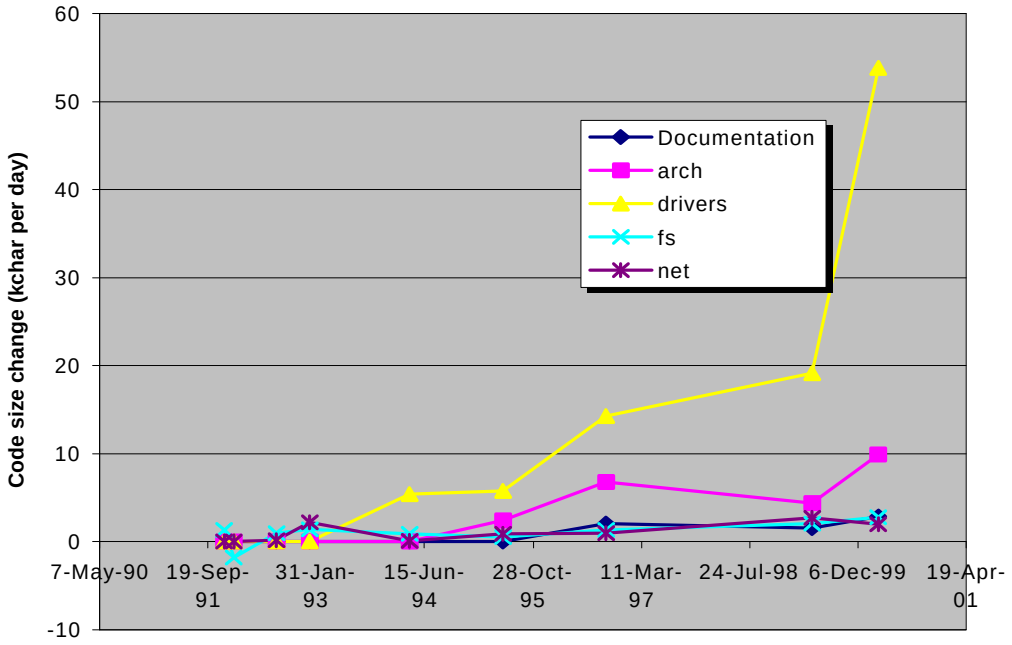

Figure 10. Growth of architecture dependent and extensible components.

Quality control and Linus's Law

An important characteristic of the Linux community is the way it controls the quality of its production. This, indeed, is the key when we try to understand how a distributed group of self-managing developers are able to create a useful product. Product quality is a critical success factor for an operating system project, and it is exactly here where the Linux development model shows its strength.

Almost all software has bugs, and in complex systems these bugs can emerge as a result of complex interactions between different program components. Large software systems are therefore difficult to develop. The more developers there are, the more difficult it becomes to control and understand all the possible interactions between software components. This is usually known as "Brooks' Law." In "The Mythical Man-Month", Fred Brooks (1995) noted that adding programmers to a late software project makes it later. He argued that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. Raymond, however, observes that if Brooks' Law were the whole picture, Linux would be impossible:

> Gerald Weinberg's classic "The Psychology Of Computer Programming" supplied what, in hindsight, we can see as a vital correction to Brooks. In his discussion of "egoless programming", Weinberg observed that in shops where developers are not territorial about their code, and encourage other people to look for bugs and potential improvements in it, improvement happens dramatically faster than elsewhere.
>
> Weinberg's choice of terminology has perhaps prevented his analysis from gaining the acceptance it deserved—one has to smile at the thought of describing Internet hackers as "egoless". But I think his argument looks more compelling today than ever. (Raymond, 1999:61-2)

Quality control is quite a different process in the Linux developer community than it is in traditional product development. Openness means that members of the developer community are able to review the work of others'. An interesting effect of this is that quality control of the results is good. Whereas traditional software development models mainly used end-users are a source of reclamations and bug-reports, in the Linux model users become problem solvers, providing an enlarged set of problem solving resources.

The user-developers bring many different perspectives, approaches, and experiences into play. From some of these perspectives, a specific problem is easier than from others. When the developer population is large enough, there probably is someone to whom the problem is easy. Raymond formulates this principle as “Linus’s Law”:

> *Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix is obvious to someone.*
>
> Or, less formally, “Given enough eyeballs, all bugs are shallow.” (Raymond, 1999:41)

Raymond argues that this is the main difference underlying the cathedral-building and bazaar styles. Whereas development problems are tricky, insidious, and deep phenomena in the cathedral model, in the bazaar model one can assume that “they turn shallow pretty quick when exposed to a thousand eager co-developers pounding on every single new release.”

> And that’s it. That's enough. If ”Linus’s Law” is false, then any system as complex as the Linux kernel, being hacked over by as many hands as the Linux kernel, should at some point have collapsed under the weight of unforeseen bad interactions and undiscovered “deep” bugs. If it’s true, on the other hand, it is sufficient to explain Linux’s relative lack of bugginess and its continuous uptimes spanning months or even years. (Raymond, 1999:42)

Raymond’s formulation of Linus’s Law therefore complements Von Hippel’s (1988) argument that users are important sources of product innovation. Indeed, Linus’s Law shows that there are two essentially different reasons why users are important for innovation. As Von Hippel noted, users can modify and adapt innovations, and thereby add value to them. As Raymond notes, however, users can also play an important role in quality control.

This second role is not a trivial one. Indeed, it is a very fundamental phenomenon which has major implications for the theory of innovations also more generally. Users appropriate innovations in idiosyncratic contexts. These contexts differ both cognitively and situationally. When source code is available, software bugs can be characterized and debugged using these multiple perspectives, each of which rests on large stocks of

unarticulated knowing. In open source environment, therefore, software bugs can become opportunities for innovative contribution, and not just sources of frustration.

In Linux development, the situation consists in part of the complex system of hardware and software that interacts with the operating system. To limit this complexity, developers, for example, use old and tested compilers to be able to separate compiler bugs from the kernel bugs. The Linux kernel mailing list FAQ answers the question whether different compilers can be used to compile the kernel in the following way:

> Sure, it's your kernel. But if it doesn't work, you get to fix it. Seriously now, there is really no point in compiling a production kernel with an experimental compiler. Production kernels should only be compiled with gcc 2.7.2.x, preferably 2.7.2.3. Newer kernels are known to break the 2.0 series kernels, known symptoms of this breakage are hwclock and the X server seg.faulting…Regarding 2.1 kernels, they usually compile fine with other compiler versions, but do NOT complain the list if you are not using 2.7.2. Linux developers have enough work tracking kernel bugs, to also be swamped with compiler related bugs.[11]

Often it is impossible to predict the interactions between different system components, and the only way to learn about them is to use the system in different concrete settings. Sometimes the bugs are in the hardware, and there is no way they can be corrected by studying the software. For example, Intel Pentium processors have bugs that need to be corrected by workarounds. In some cases, hardware bugs can be so unpredictable that there is no workaround. The Linux kernel mailing list FAQ lists some of the known processor bugs and, for example, tells that the AMD K6 processor has unpredictable hardware errors:

> **The AMD K6 "sig11" bug**, affects only a few K6 revisions. Was diagnosed by Benoit Poulot-Cazajous. There is no workaround, but you can get your processor

[11] http://www.tux.org/lkml/

> exchanged by contacting AMD. 2.2.x kernels will detect buggy K6 processors and report the problem in the kernel boot message. Recently, a new K6 bug has been reported on the linux-kernel list. Benoit is checking into it.

The importance of testing new software code is strongly emphasized, and "good ideas" rarely get support without working code that implements the idea. The MAINTAINERS file[12] that lists people responsible for the various kernel modules also gives guidelines for submitting changes to the kernel:

1. Always *test* your changes, however small, on at least 4 or 5 people, preferably many more.
2. Try to release a few ALPHA test versions to the net. Announce them onto the kernel channel and await results. This is especially important for new device drivers, because often that's the only way you will find things like the fact that version 3 firmware needs a magic fix you didn't know about, or some clown changed the chips on a board and not its name. (Don't laugh! Look at the SMC etherpower for that.)
3. Make sure your changes compile correctly in multiple configurations. In particular check that changes work both as a module and built into the kernel.
4. When you are happy with a change make it generally available for testing and await feedback.

The outline of the bug detection and removal process is straightforward. For a software bug to be removed from the system, first someone has to realize that there is a bug. After a bug has been detected, it has to be characterized, preferably by describing repeatable conditions under which the bug can be observed. This phase consists of diagnosing the exact nature of the bug. When the bug has been understood, it can be solved. This phase consists of writing new code that corrects the bug, and testing the new code to verify that

---

[12] MAINTAINERS file can be found from the root directory of new releases of the Linux kernel, for example, from http://www.kernel.org/pub/linux/kernel/

the bug has been removed, and that no new bugs have been introduced in the process. When a tested solution is available, it is distributed to other developers. Finally, if the bug is important enough and the new code does not seem to create excessive problems, the bug fix is eventually integrated into a new kernel release. This process is depicted in Figure 11.

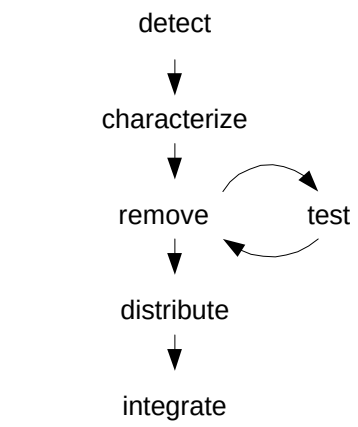

Figure 11. The basic bug removal cycle.

In actual practice, this rather straightforward process is more complicated. It relies on tools, social practices, and knowledge resources that implement the abstract bug removal procedure. Moreover, the developers apply the various resources in a creative way, improvising according to the needs of the situation. The appropriate way to improvise depends on the audience: if the community of developers understands that a specific way of breaking the standard procedure is justified, rules can be broken. The behavioral standards are usually given as expectations and suggestions, and there are only few explicit procedures for doing things. Usually such explicit procedures do not result from explicit specification of social processes; instead, they arise from the design of specific tools used in the process. In other words, some aspects of the process are hard-wired into the functionality of the tools.

Some widely used resources and tools for Linux kernel bug management are shown in Table 1. Some of the resources, such as the JitterBug bug reporting and patch distribution system, are both platforms for collaboration and informational resources. Some informational resources are meta-level resources that describe procedures used in bug processing. An important meta-level resource is, for example, the linux-kernel mailing list FAQ document that lists frequently asked questions and gives answers and links to further information on them. Some tools interface the object of development, i.e., source code, to the development community. An example of such a tool is the CVS version control system, and the CVS vger –server that maintains the different patches and versions of the kernel in hierarchical trees, and which provides a shared repository of source code to all developers. Many of the tools listed in the table are well-known generic and Unix-tools.[13] Although their existence is often taken for granted, in practice the bug removal procedures critically depend on the tools and their evolution.

Although Table 1 shows the main tools currently used in the bug removal process, one should note that many of these tools have emerged during the evolution of the kernel. Some of the tools and resources explicitly address problems that the success of the kernel development has created. For example, the Kernel Traffic list[14] produces an edited summary of the large volume of mailings in the linux-kernel mailing list. The linux-kernel mailing list FAQ, in turn, documents the common questions that novice developers have, as a way to keep such relatively low-priority questions crowding the linux-kernel list.

---

[13] *man* is a program for reading manual pages. *gcc* is the GNU c-compiler. make is a program that manages the compilation process. *gdb* is the GNU debugger. *diff* is a program that creates difference files from two source code files, and which updates modified files using differences. This is used to distribute patches that update files with modifications. *gzip* is used to compress files, and *tar* is used to package several files into one for easier distribution. Linux developers also use generic tools such as *IRC*, *ftp*, email, and mailing lists. Other tools and resources are systems that are more specific to Linux development. *ksymoops* maintains a list of symbols used in error messages. *Kernel Traffic* is an edited weekly summary of the mailings in the linux-kernel mailing list. *LDP* is Linux Documentation Project, which maintains a set of guidelines and documents for Linux developers. *JitterBug* is system that maintains information of known bugs and patches. *CVS* is a version control system that integrates with a shared CVS server called *vger*.

[14] http://kt.linuxcare.com/

Similarly, the mailings in the linux-kernel list are archived, so that they can be searched when someone needs to know whether something is known about a potential bug. In that way the mailing list archives provide a simple but effective form of community memory.

processing phase			**information resource**	**tool**	**community resource**
detect			compiled code documentation	man	LDP
debugging	characterize		source code linux-kernel list FAQ JitterBug oops-tracing.txt Kernel Traffic LDP project-specific sites linux-kernel archives README files log files bug reporting form	editor gcc make gdb ksymoops IRC computer configuration	linux-kernel list JitterBug personal email IRC channels kernel-newsflash LDP project-specific lists
	remove		source code	editor gcc make	
		test	patch MAINTAINERS file	diff gcc make editor ftp	personal email linux-kernel list
distribute			patch MAINTAINERS file	gzip tar email ftp	linux-kernel list JitterBug
integrate			patch release	CVS vger package managers	Maintainers vger

Table 1. Kernel bug management resources.

Already a superficial analysis of the tools and resources used in the bug removal process reveals that a complex socio-technical system underlies this apparently simple process. Quality control in the Linux kernel, therefore, is not only about finding bugs and correcting them. It is also very much about the complex and continuously evolving system that makes the detection, characterization, and removing bugs possible in the first place.

Moreover, the bug removal process is operating in a context where source code patches can be distributed and integrated into new kernel releases.

Tools and resources therefore mediate the relations between developers, developer community, and the technical object that is developed. The overall bug removal system can then be represented in a simplified conceptual way as in Figure 12. Whereas Raymond emphasizes the cognitive capabilities of co-developers, he forgets the role of mediating technologies. Sociocultural analysis (e.g., Wertsch, 1998; Leont'ev, 1978; Engeström, 1987; Cole, 1996) would highlight the fact that cognition is also very much dependent on the tools and resources that are available for the developers. Implicitly, the guidelines for kernel bug removal note this when they insist that new patches need to be tested in different hardware configurations. Linus's law could then be augmented by noting that it is the combination of eyeballs and other resources that makes even the most insidious bugs shallow.

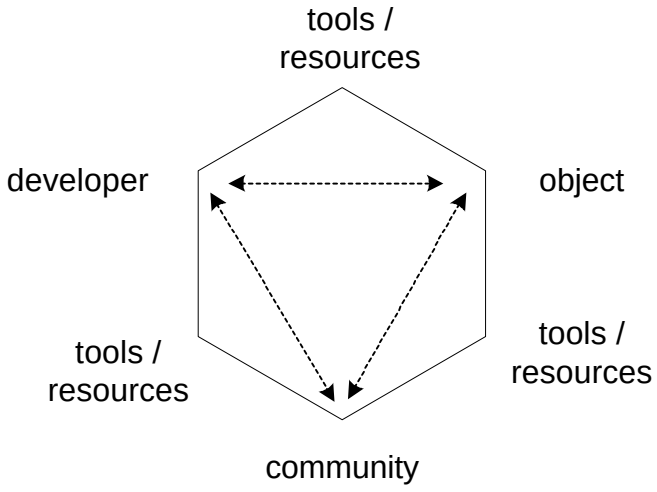

Figure 12. Mediated interactions in the bug removal process.

Quality control in innovative and continuously evolving projects is essentially about learning. Whereas the traditional models of learning in product development focused on decreasing errors in a given product design, in the case of Linux learning is also creative.

Theoretical models of innovative learning generally claim that learning starts when a problem arises, and innovative solutions are generated in the process of defining ways to overcome the practical problem at hand (e.g., Dewey, 1991; Schon, 1963; Engeström, 1987). The Linux development model is compatible with such theories of innovative learning. In this sense, it is also different from the conventionally used product development practices (c.f. Griffin, 1997; Mahajan & Wind, 1992), which rarely consider the microstructure of learning. The Linux model, however, does highlight some characteristics of successful product development that have been discussed within the disciplined problem solving literature on product innovation (Brown & Eisenhardt, 1995). Within this literature, the importance of exploratory learning, non-financial goals, continuous problem solving, and diversity of problem solving resources have been often noted.

Developer incentives and resource allocation

Linus's Law and the compatibility of social and technical structures may explain why community-based technology development can lead to high quality results. Theoretically, Linus's Law means that each contributor can contribute where his or her impact is greatest. In this sense, the Linux development community implements a market where cognitive resources are effectively allocated.

The existence of such allocation mechanisms do not, however, explain why the product emerges. In the case of a software project, some development may occur simply because debugging, coding, and solving technical problems can be rewarding as such. This, indeed, is an important driver for development. Developers often describe the joy of hacking as their primary motive (e.g., Raymond, 1999).

To be able to make relevant contributions to the project, one has to skillfully use tools and concepts, and do something that no one has ever done before. If one succeeds in creating a new piece of software that is taken into use in the community, there is clear evidence of success and a socially validated proof of mastery. Indeed, software projects provide

unlimited opportunities for testing one's skills and creating new grounds for mastery. In this sense, one may regard "the joy of hacking" as something highly non-trivial. Instead, it can be seen as a prototypical driver for technological progress. As Csikszentmihalyi (1990) has shown, people are most happy when they are performing on the edge of their competences. Software is special simply because the developers live in a world of their own creation (Weizenbaum, 1984). In such a world, each new advance and border-crossing moves the boundaries further, expanding the domain where new achievements can be realized. In this context, Linux, therefore, is not just a operating system kernel, but an interesting metaphor of modern technological culture.

Such socio-cognitive explanations are important parts of the whole picture when we try to understand the drivers of technological change. As such, however, they cannot explain the fact that a technological system evolves as a coherent system.

Raymond has proposed that the dynamic of Linux development can be understood by noting that the ownership rights that underlie the development are essentially similar to those that underlie the Anglo-American land tenure (Raymond, 1999).[15] In this Lockean theory of property rights, ownership can be gained in three different ways. First, in frontier areas that have never had an owner, one can acquire ownership by homesteading: by mixing one's labor with the unowned land, fencing it, and defending one's title. Second, in an area where ownership already exists, one can acquire ownership through the transfer of the title. In theory, at least, such chain of title goes back to the original homesteading. Third, property that has been abandoned can be claimed by adverse possession. This happens in a similar way as the original homesteading: one moves in, improves the property, and defends the title as if homesteading (Raymond, 1999:93). Similarly, in the space of potential technological developments of the Linux system, developers can gain ownership rights for specific sub-projects.

---

[15] Raymond discusses the property rights in his article Homesteading the noosphere. This is included in (Raymond, 1999), and also available at http://www.tuxedo.org/~esr/writings/

These informal ownership rights are important because they make exchange possible. According to Raymond, the exchanges that underlie the success of Linux, however, are not conventional economic transactions. Instead, he suggests that the system of social exchanges can be understood as a gift culture (Raymond, 1999:97). The developers give the results of their work as gifts to the community, and the mutual exchange of gifts leads to a technically highly advanced system with a very high quality. By giving gifts, the developers are also able to build reputation. Good reputation among one's peers is a source of reward in itself. Reputation, in turn, makes it easier to mobilize community resources. In some cases, good reputation within the community may spill over to another areas the society, and earn a higher status there.

In this sense, the Linux development community is similar to academic disciplines. As Raymond notes, one peculiarity of such communities is that only the members of the community can appreciate the quality of gifts. Indeed, the value of a gift is what others can make out of it.

There is, however, more than one way to run a gift culture. According to Raymond, two sides of gift culture are represented within the software development community by crackers, who try to gain reputation by breaking computer security, and by benevolent hackers, who gain reputation by sharing useful software in source code (Raymond, 1999:100). The cracker culture is a tightly closed one, and protects its secrets, whereas the hacker culture is based on transparency and openness. This has obvious implications for the way competence, knowledge, and technological artifacts develop. Openness means that results and techniques can accumulate, as it is relatively easy to learn from others' work and add on it. There is a very strong expectation within the community that developers should develop their systems in ways that make is possible and easy for others to improve on them (e.g., DiBona, Ockman, & Stone, 1999:221-51). This expectation is reflected, for example, in the Open Source Definition[16], which forbids deliberately

[16] http://www.opensource.org/osd.html

obfuscating source code, and which requires that source code be distributed in a format that a typical programmer would use to modify the program.

Raymond argues that Linux development works well because reputation is mainly associated with software modules. Although, according to Raymond, developers are driven by ego-satisfaction there are strong taboos on claiming personal credit. Reputation is made objective by associating it with the produced technical artifacts. Although hackers relatively freely flame each other over ideological and personal differences, it is rare that they would publicly attack someone else's technical competences. Instead of criticizing each other, they criticize the software.

> Bug-hunting and criticism is always project-labeled, not person-labeled. Furthermore, past bugs are not automatically held against a developer; the fact that a bug has been fixed is generally considered more important that the fact that one used to be there. (Raymond, 1999:110)

Raymond also notes that the hacker culture consciously distrusts and despises egotism:

> …self-promotion tends to be mercilessly criticized, even when the community might appear to have something to gain from it. So much so, in fact, that the culture's 'big men' and tribal elders are required to talk softly and humorously depreciate themselves at every turn in order to maintain their status. (Raymond, 1999:107)

In Raymond's terms, reputation is very much "project-based." His interpretation is that most hackers, as members of the cultural matrix, learn that desiring ego satisfaction is bad. However, he also notes that the rejection of self-interest in the hacker community is so intense that it probably plays some other valuable function.

Raymond proposes two explanations for the taboos on posturing and personal attacks on technical competences. First, when results are judged by their merit, the community competence base increases rapidly. The taboo against ego-driven posturing therefore increases productivity. More importantly, however, when personal status is discounted, the community information on system quality closely reflects the quality of the system, and does not become polluted by personal reputations of the developers.

Implicitly Raymond's account on the reputation mechanisms, however, assumes that there are two systems of reputation operating at the same time. The other one drives the developers as seekers of ego-satisfaction, whereas the other describes the quality of the collectively produced artifact.

As in any social system, reputation, authority, and legitimation are products of history, and all abstract definitions of them fail in a closer study. Reputation is defined within the community in question, and the criteria it uses in managing reputation change as the community evolves. The only way to learn the rules of reputation building is to become engaged in the community discourse. Breaking the rules, in turn, can lead to excommunication.

A recent email exchange in the linux-kernel mailing list gives an example of this process. The weekly Kernel Traffic linux-kernel mailing list summary called this episode "Tulip driver developer flame war." There were 71 mailings around the topic between 13-20 March, 2000. The main issue was the style of development by Donald Becker, one of the people mentioned in the credits file. The Kernel Traffic editors summarized some of the discussion:[17]

> In the course of argument, Donald Becker said to Jeff Garzik, "you didn't understand the task you were taking on when you decided to take over maintaining the Ethernet drivers. It took years to write the driver set -- it's something you can just pick up in a few months. And expecting me to now fix or maintain your hacked up code branch is just completely unreasonable." Jeff replied with venom:
>
> > No one expects anything from you and has not for a long time. If you wanted to actually WORK on the drivers, rather than just complain, then I'm sure many people including myself would find that work very valuable. …

[17] This is an abridged version of the summary in Kernel Traffic #60, 27 Mar, 2000, http://kt.linuxcare.com/

Elsewhere, Jeff went on, "Donald, I, and others all seem to agree that having his drivers and the kernel drivers diverge is a poor situation. However, while Donald continues closed source development with periodic code drops, and does not work with other kernel developers when creating infrastructure, I do not see a resolution to the situation any time soon." David Ford replied angrily, "Please explain how his code development is closed source? This is totally BS and you know it. All the code is available, all the list discussion is available, and patches and requests are accepted all the time. Quit it. His development is quite open …" Linus Torvalds replied:

> David, pipe down.
>
> You seem to like the approach Donald has taken. But take it from me, it DOES NOT WORK.
>
> The problem is that maintaining the drivers in their own small universe means that only those people who follow the driver development will ever even test them. …
>
> I fixed the tulip driver at least twice to work with the media detection, and sent Donald email about what I had done and why … I don't know if my fixes ever actually made it into Donald's version, because after the second time I just stopped bothering trying to re-fix the same thing, and I never updated his driver again.
>
> In contrast, what Jeff and others have done have been of the type where immediately when a fix is made, it is released. Which means that if there are problems with it, people who follow new kernel releases will know. Immediately. Not in a few months time when the next "driver release" happens.
>
> This is what Jeff means with "closed source". Yes, the sources are there. Yes, they get released every once in a while. But Donald doesn't let people _participate_. He thinks he is the only one who should actually touch the driver, and then he gets very upset when things change and others fix up "his" drivers to take into account the fact that the interfaces changed. …

Jeff also replied to David:

> Donald's development is not open AT ALL. … He disappears for many months, creates a design without interfacing with kernel developers, and then appears again with a code drop.
>
> It is classic cathedral style of development. Read Eric Raymond's paper on why the bazaar method is far, far superior. …

Donald replied to Jeff:

> A quick search of the two very active Tulip mailing lists reveals that you have contributed nothing until this year. Apparently you were not even a subscriber until then, and know nothing about the very open way development has been done. Yet you willing throw around pejorative phrases like "cathedral style" -- a hot button in this community.
>
> For those not interested what superficially appears to be a kernel power grab, there are issues underlying all of what appears to be a personal conflict.

The Kernel Traffic summarized in more detail Donald's arguement that the underlying questions are about the stability of kernel source code interfaces, testing the drivers in the context of continuously changing kernel releases, and the large and frequent kernel patches that make life difficult for driver developers. Donald further stated that the earlier interfaces were better than the more recent ones, and questioned the viability of the monolithic, single-point kernel source tree. Linus Torvalds replied:

> You're basically the only one thinking so.
>
> The fairly recent changes in 2.3.x (the so-called "softnet" changes) are just incredibly more readable and robust than the old crap was that I don't see your point at ALL.
>
> Just about every single network driver out there was SERIOUSLY broken … I know, I had fixed many of them. The games the drives played …were just incredibly baroque, and had absolutely NOTHING to do with "clean".
>
> All of that crap is gone, and it was much overdue. …

The Kernel Traffic summary further recorded that Donald was not any more considered to be the owner of the network drivers that he had earlier developed. First Donald lamented on the difficult situation he is because he doesn't have sufficient control over the development. Then Linus gave his assessment of the situation. The Kernel Traffic summarized:

Elsewhere in an entirely different subthread, Donald argued:

> I'm in the increasingly untenable position of being expected to maintain drivers for the current and older kernels, but not having any influence over the new development exactly because of that backwards compatibility. It's no fun being responsible for just the old versions, especially after I did years of unpaid development work.
>
> There were many interface changes added incrementally in the 2.3 kernels. Some were added without consideration of, or even in opposition to, cross-version compatibility. And few of those interface changes were designed, as opposed to just hacked in. When I proposed an new PCI detection interface I wrote a skeleton driver, converted several of my drivers, demonstrated that it worked with several hardware classes and wrote a usage guide. But the few day hack was added because the patches were incremental (even if misdesigned and broken).

Linus replied:

> Donald, that's not true, and you know it.
>
> Neither I nor anybody else has expected you to maintain the drivers for quite a long time now - you just didn't seem to have the interest, and a lot of people have acknowledged that. That is why there ARE new maintainers for things like tulip and eepro100, whether you like it or not.
>
> You did not lose influence of the drivers because you want to maintain backwards compatibility. You lost influence over the drivers simply because you never bothered to send in your changes. Don't start blaming anybody else.

As the outline of the driver developer flame war shows, the open source model has conflicts, and reputation and authority can be gained and lost. As the comments of Linus Torvalds reveal, the breaking of expectations can lead to neglect of contributions, thus effectively destroying the possibilities to gain reputation within the community. When the reputation has decreased enough, it becomes easy for someone to start parallel development. Eventually this can lead to explicit transfer of “ownership” rights.

The question is, however, also about the locus of control. Donald, as a driver developer, prefers that the kernel stays stable so that he can more easily develop his software. Linus, however, indicates that in the Linux community, the kernel is the central artifact, and

driver developers should adjust to the requirements of kernel development. Donald's position is therefore rather similar in relation to the kernel as kernel developers' position to the GNU c-compiler. As was noted above, the kernel developers argue that the compiler version needs to be held constant to effectively debug problems in the new kernel releases. By following such discussions, novice developers can learn how the open source development is interpreted in practice, and what are the taboos that should not be broken. As can be seen from the example above, the open source model is not restricted only to the software code; instead, it implies a code of conduct, which is supported by a socialization process that also occurs in the open source mode. The negotiation of social practices and the development of reputation can be observed by the global community in real-time.

Rules and regulations

As was noted above, procedures that underlie Linux development, are often learned when novice developers become socialized into the community. Many of the procedures and practices are also embedded into the functionality of the tools that support the development. There exists, however, also important explicit standards and agreements that are key components in the development system. On a technical level, one such standard is the ISO Posix interface standard, which defines the way application programs can use the kernel functions. The licensing policy that defines the open source model is a central social innovation that underlies Linux. Indeed, a lot of social order is encoded and embedded into the licenses and documents that describe different licensing alternatives.

There exist several variations of commonly used open source license policies, some of which are more restrictive than others. In a clear contrast to the typical use of copyright licenses, which restrict the ways the copyrighted work can be used, the main goal of the free software licenses is to guarantee the ongoing re-use and development of software.

> In commercial software, the license terms are designed to protect the copyright. They're a way of granting a few rights to users while reserving as much legal territory is possible for the owner (the copyright holder). The copyright holder is

> very important, and the license logic so restrictive that the exact technicalities of the license terms are usually unimportant…In free software, the situation is usually the exact opposite; the copyright exists to protect the license. The only rights the copyright holder always keeps are to enforce the license and to change the license terms of future versions. Otherwise, only a few rights are reserved and most choices pass to the user. In particular, the copyright holder cannot change the terms on a copy you already have. Therefore, in free software the copyright holder is almost irrelevant—but the license terms are very important.[18]

Free software licenses guarantee various rights to use, modify, distribute, and distribute modified code. According to the Debian Free Software Guidelines, and the Open Source Definition that has been derived from it, there are several requirements that a software component must meet.[19] First, the license must guarantee that the code may be freely distributed without royalties. Second, the source code must be easily available, and the license must not restrict the distribution of the source code. Third, the license must allow distribution of modifications and derived works under the same terms as the original code. These are the main characteristics of open source software. In addition, to comply with the Debian Guidelines and Open Source Definition, the license may restrict distribution of modified source code only if it allows distribution of "patch files" that can modify the original code at the compile time. This is to simultaneously guarantee that the original programmer can maintain the integrity of his or her code, and that subsequent modifications are still possible by adding new "patches." In addition, the license must not discriminate against any persons or groups, or against any uses, including commercial use. The license must also apply to all to whom the program is distributed, without the need to write separate license agreements. Further, the license must not require that the program be used as a part of a specified software distribution. To avoid contamination of licenses, for example by requiring that the program be distributed only together with other programs that have similar licenses, the license must not place restrictions on other programs.

---

[18] "Free software licensing alternatives." http://metalab.unc.edu/pub/Linux/LICENSES/theory.html

[19] "A social contract." http://www.debian.org/social_contract.html

The least restrictive form of license is public domain, which puts no restrictions on the use or distribution of the original code. It can be freely copied, used, and modified for any purpose. If a public domain program is available as source code, it adheres to the Open Source Definition. A rough estimate of the use of public domain licenses is that in mid-1997 about 3 per cent of about 2600 software packages and documents on the Sunsite server were defined as public domain sources.[20] Public domain licenses are therefore not very common within the open source community.

The least restrictive commonly used license is the MIT or X consortium license, which requires only that the original copyright and license terms are included in the distribution. Shareware programs often use this type of license, although they may also request a donation from users who find the program useful. A slightly more restrictive license is the BSD-license, which requires that all documentation and advertisements acknowledge the original copyright holder. Freely Redistributable Software, in turn, has a FRS license, which requires that software can be freely copied, used, and locally modified. It must also grant the right to distribute modified binaries, although it can put some restrictions on the ways the modified source code can be distributed. To be "open source," FRS restrictions have to adhere to the Open Source Definition, however.

The most widely used free-software license is the GNU General Public License, or GPL. This is the license under which the core Linux system is distributed. It allows free copy, use, and modification. Modified source code can be redistributed if the modified source code shows a "prominent notice" of the modification. There is also a requirement that an interactive GPL program displays a start-up notice that shows it is a GPL program. More interestingly, however, the GPL license also requires that if a program contains components that are licensed under GPL, all the components must have a GPL.[21] This last

---

[20] http://metalab.unc.edu/pub/Linux/LICENSES/theory.html

[21] GPL was originally defined by the Free Software Foundation, with the explicit aim to promote non-proprietary software. GPL proponents argue that proprietary software limits innovation, and that fair use of software should be allowed in the same way as fair use of scientific results. Richard Stallman (1999), the founder of Free Software Foundation, has noted that the recent open source movement has to a large

requirement of GPL has no simple interpretation in practice (Perens, 1999). To enable commercial programs to be developed for the Linux platform, the license in Linux explicitly declares that the use of the system is not considered to generate a derivative work. This means that commercial and proprietary programs can use Linux even when they don't want to use GPL. The original idea in GPL was that it shouldn't be possible to change open source software proprietary by adding to it some proprietary components (Stallman, 1999).

The license, although important, is only part of the story, however. There are several explicit and implicit expectations that define appropriate behavior within the open source community. For example, the Debian GNU/Linux community has defined a "social contract" that declares its commitments to keeping the programs free software, transparency in handling software bugs, and support for users who develop commercial and restricted software based on the free software developed by the community. Moreover, the rights to distribute key components of programs are tightly controlled by informal social mechanisms. A key factor in open source development is, however, the fact that formal contracts are intended to promote development, not to restrict it.[22] Property rights are used here to enable symbiotic development, instead of competition.

In the course of time, commercial interests have become increasingly important in the Linux community. In the beginning, Linux development was closely aligned with the free software movement. Linux development was explicitly defined as a non-commercial

---

extent neglected the ethical implications of software licensing, and focused on a short-sighted way to the productivity aspects of the open source model.

[22] Open source projects therefore also remind us that there are intellectual properties for which appropriation of returns on investment is not a major issue. It is also interesting to note that in the historic controversies on patent rights (c.f., Machlup & Penrose, 1950) the proponents of free market argued that patent rights may slow down development as they distort markets and do not necessarily allocate returns to those who contributed to the invention. Both free market advocates and proponents of patent monopolies, however, missed the possibility that technological development can result from giving away monopolies.

project. In 1992 Torvalds noted that the only exception for the free use of the code was the restriction that someone creates a commercial product out of it:

> The only thing the copyright forbids (and I feel this is eminently reasonable) is that other people start making money off it, and don't make source available etc…This may not be a question of logic, but I'd feel very bad if someone could just sell my work for money, when I made it available expressly so that people could play around with a personal project. I think most people see my point. (quoted in DiBona, Ockman, & Stone, 1999:248)

More recently, commercial organizations have become important actors in the Linux development system. This has created tensions and continuing discussions on the way open source licensing can be applied in practice (e.g. Perens, 1999). Raymond argues that the open source definition is a major improvement to the original GNU license policy, as it explicitly allows commercial software developers to join the Linux development community. This evolution of licensing policy can be viewed as one example of the ways in which the socio-technical system changes its social expectations in a response to the increasing variety of actors in the developer community.

When the institutions of licensing are viewed as social innovations, it is possible to see that also social innovations can be a source of path dependence in socio-technical evolution. When the Open Source Definition is used as a guideline for licensing, it becomes very difficult to return to the closed source mode. Indeed, this was exactly the intention of the Free Software Foundation when it designed the GNU license, with the aim of guaranteeing that the results of technical work can accumulate. The Open Source Definition, with its less contagious licensing policy, however, makes it possible to incrementally develop closed extensions to the Linux system. In practice, this may be difficult as most developers rely on the collective resources of the community, and unfair free-riding easily leads to social exclusion. The transparency of the open source development model also means that it is difficult to hide such attempts of free-riding.

Why Linux works: Linux as modern economy

Torvalds has frequently noted that his approach to design is pragmatic. The pragmatic approach has, for example, meant that he has mainly been interested in the portability of the system across existing computer architectures, and theoretical designs that were supposed to support portability have been of secondary importance. A somewhat surprising result is that this pragmatic approach to portability actually has led to easily portable systems. According to Torvalds the main idea was to design Linux so that it operates on “sane” computer architectures (Torvalds, 1999:104-5). Instead of abstracting operating system principles from theoretical research on operating systems, Linux has implicitly abstracted existing operating system architectures, and best practices that had evolved in previous implementations. Whereas operating system research in most cases starts from logical and computational considerations—and only afterwards tries to overcome practical implementation constraints—the development of Linux has followed the opposite route. Linux can therefore be seen as an example of grounded theory, where theoretical constructs emerge through observation and conceptualization of practice (c.f.Glaser & Strauss, 1967).

Although there are similarities in the social systems that underlie scientific disciplines and Linux development, there are also important differences. The main difference is that the Linux community constructs a shared technological artifact. This artifact, i.e., software source code, enables social processes that seem to be in some ways more effective than those of traditional sciences. In the case of Linux, accumulation is an objective fact. Whereas the traditional view saw scientific progress as accumulation of increasingly accurate representations of reality, Linux development constructs its reality in an ongoing process. Linux, as a shared technological artifact, acts as a common reference point to the community.

This shared artifact makes the dynamics of reputation different from those that organize scientific research. In contrast to traditional academic disciplines, where reputation is tightly personalized, in the Linux world reputation can also be attached to parts of the technological artifact.

There are also some potential explanations why the taboos of Linux community work. When newcomers have the possibility to make important contributions to the community project, they have strong incentives to do just that. In a way, the Linux community implements the idea that anyone can be a star, and it only depends on your effort. Here it perhaps reproduces some of the values and rhetoric of Silicon Valley. In practice, Linux development, however, is a collective effort, and achievement is possible only by using collectively created resources. As the developers build on existing contributions, the gift givers eventually get their gifts back in an improved format. There is no “tragedy of commons” or “winner takes all” in this space of technical artifacts, but only positive returns. As long as the quality system works, the more you give, the more you get.

Under such conditions, a critical success factor is the capability to mobilize collective resources for promising new directions. This, in turn, requires that there are effective ways to manage attention. In addition to its important role in the allocation of control, reputation is also a key in the management of attention. However, as the case of Linux shows, a successful project may attract so much attention that strong social filters are needed to avoid overload at the centers of power. In addition to expectations that newcomers learn in the socialization to the community practices, also technology is used to maintain social filters. For example, the linux-kernel mailing list has so much traffic that it is impossible for most members to read all messages in detail. Many readers, therefore, use mail scripts that automatically delete mail. Sometimes deletion is based on the topic, but sometimes also based on the sender. Although Linux development has been characterized above as an open social process that occurs in a collaborating community, this, of course, does not mean that the community would not exclude some potential members. In practice, social exclusion is often implemented using the same computer programs that the community develops.

In a system where every sub-project is potentially important to the overall success, and the only way to evaluate the value of contributions is by evaluating them after the results are generated, it is useful to celebrate the results, and downplay the status of individuals. Such a policy guarantees that authority is tightly linked to competence and responsibility, at the

same time making continuous revolution possible. This, however, also requires that the community understands that the celebration of results implicitly celebrates their producers. Therefore it is critical that the source code files keep an accurate and open record of the contributors to the project. In knowledge intensive work, commitment is critical, and any perceived unfair distribution of rewards very rapidly deteriorates commitment.

The visible humility of tribal elders in the Linux community has, however, also another potential explanation. In technical terms, humility is useful for flexibility. When there is no rigid social hierarchy, the system can rapidly utilize emerging opportunities and develop into new directions. The growth of the system is not limited by its starting points; instead, it grows where the growth is fastest. This is of fundamental importance. Linux is not a pre-defined product; instead, its developers can plug in their own interests and interpret the possibilities of the accumulated system from their own perspectives.

Linux, therefore, seems to represent a socio-technical system where resources consist of technological artifacts, tools and resources, social innovations embedded in institutions, and collective competences. These different components of the system develop simultaneously. A distinctive character of the open source model is that the boundary between ideas and implementation is not clear, and innovative ideas are often inseparable from their implementation. Source code is used as an external knowledge base and a cognitive tool, and cognitive resources are distributed among technological artifacts and humans. The development directions are to a large extent based on management of attention and accumulation of reputation.

Linux development is based on a complex interplay between social practices and a focal technological artifact. Any single driving force, for example, financial rewards, cannot explain Linux development. We do things that allow us to use our competences and develop them, which our peers appreciate, and which are meaningful in the social context we are in. Linux development, therefore, is not a result of any specific economy based on transactions, bartering, or exchange of gifts. Instead, it is better characterized as a form of social life. The artifact that organizes this form of life emerges as people go on with their

lives in ways that are meaningful to them. In this sense, Linux development is totally endogenous: the technological artifact can be seen as a side-effect of the fact that people live in a social world.

Since Schumpeter economists have assumed that innovation is about economic change. More exactly, innovation has been *defined* as something that has direct economic implications. According to this view, technological change becomes innovation only when it changes production functions that relate economic inputs to economic outputs. The Linux case shows, however, that there can be technological change that is not captured by this definition of innovation. Yet, Linux obviously has potentially very important implications for software industry and for the rest of the economy. In other words, although there exist, of course, many links between technological and economic development in the modern society, they are not causally related. Therefore it is not obvious that we can explain economic growth by technological development. Nor it is obvious that we can explain technological development using economics. To put technology and economy under the same explanatory framework probably requires that we turn back to the sociology of economic processes and, indeed, explain economy itself as one of the sophisticated technologies of modern society.[23]

Berman (1982) characterized the modern mentality as a set of culturally shared beliefs. According to Berman, modern mentality is composed of strong individuality, belief in self-directed reason, assumption of the individual as the locus of control, commitment to progressive improvement, generally optimistic outlook, and a strong belief in meritocracy and social mobility. In the Linux community, these beliefs are easy to detect. For example,

---

[23] Simmel's (1990) observation was that money is the most perfect tool. In this sense, we can say that economics never really neglected technological innovation, but was, indeed, a discipline that studied the implications of one technology, money. A sociological and cultural starting point, however, leads us to discuss economic systems as forms of socio-technical systems. I am not aware that such economic theories would currently exist. It seems, however, that they could be developed from the sociocultural activity theory (Leont'ev, 1978). Indeed, Engeström (1987) tried to develop a version of activity theory where exchange, division of labor, productive action, and technology are parts of the same structure of human activity. Engeström's focus, however, was on theory of learning.

Raymond (1999) notes that in the hacker community the best craftsmanship wins. There is a very strict rhetoric of meritocracy. In theory, everyone is judged based on the quality of the results, and seniority comes into play only in those exceptional cases where peers cannot judge quality, or when ownership rules do not work. The great commitment and enthusiasm of Linux developers indicate that the developers believe that their efforts and contributions matter, and that the system, as a whole, is improved as a result of these efforts. The joy of hacking, in turn, is very much about getting control over a constructed world, and becoming a wizard in such a technological world.

The way Linux developers live, indeed, reflects major currents in the modern world. This is also probably one of the reasons why the Linux community has been so successful in technological development. The social system of Linux community is not only aligned with the needs of the community itself. Instead, it is also aligned with important components of the broader social system where the development transpires.

Although it is important to note that in other cultural settings such modern beliefs do not necessarily organize social systems, in this case technology development indeed is closely linked to the modern worldview. The culture of hacking is probably the most perfect implementation of modernity, and therefore it also produces technological products effectively. There are no deep internal conflicts within the culture of hacking that would compromise its efficiency. Indeed, as long as it builds itself around those technological artifacts that it produces, it is able to avoid many of those conflicts that make similar efficiency difficult in broader social contexts.

The values of modern technological society are closely aligned with the values of the Linux development community. It would, indeed, be surprising to see successful technology development projects where the values of modernity were strongly contested. It is difficult to imagine successful collaborative development of technological artifacts in a cultural setting where the developers would believe in unpredictable accidents, irrelevance of one’s own interests and decisions, belief in the inevitable deterioration of the developed system, and questionability of the meaning of the whole effort. In this sense, successful

technology development requires modern values. But even within modern high-tech organizations there are important cultural differences that may constrain or facilitate open source development. For example, the Linux development model seems to require a culture with low power distance and low uncertainty avoidance. This has interesting consequences, as it is well known that the regions of the world differ greatly in this respect (Hofstede, 1991). Indeed, it may have some relevance that Finland and the U.S.A. happen to be countries with the least power distance and uncertainty avoidance.

The Linux community consists of members who live in many different local cultures. In this sense, the Linux community is an interesting case of a "global" culture, and it might provide important insights of the mechanism that link regional cultural resources to global technological development. For example, Freeman, Clark, and Soete (1982), and Perez (1985) argued that the long cycles in economy require that forms of production, organizational structures, banking and credit system, and other social institutions change before productivity of a new key technology can be realized. The rigidity of social institutions, essentially, was the reason why long cycles are long. In the case of Linux we can see a process of socio-technical change where these assumptions are not necessarily appropriate. The social system is continuously negotiated, on-line, according to the problems and opportunities generated in the process. In this sense, Linux could also be analyzed as an example of socio-technical development that escapes the logic of long cycles. The innovation process that underlies Linux development, therefore, could also give a concrete example of what the discourse on "new economy" is about.

Although social and scientific progress has often been associated with meritocracy, social mobility, and individualistic appropriation of opportunities, in the case of Linux this ideal world is in fact implemented. Here Linux development also differs from any system of economy. People construct the same collective artifact, interpret it from their own perspectives, and adapt it for their own practices. The dynamics of the technological artifact and the social system that produces it are well aligned. The speed of development is fast because there are no fundamental contradictions in the co-evolution of the community and its artifacts.

Linux development proceeds in such an expanding galaxy of technical artifacts and social relations. Fired by the engines of modernity, its boundaries explode into the space of technological possibilities. Indeed, one cannot but wonder whether it is just because of the abstract nature of this space that it has been so successful in its emergent goals. In the world of Linux, modern technological economy is perfectly realized. And—paradoxically—we find ourselves once again in the beginning of time, in the economy of gifts.

References

Armstrong, M. (1998). Kernel architecture. http://se.math.uwaterloo.ca/~mnarmstr/report2 .

Berman, M. (1982). *All That Is Solid Melts into Air.* New York: Simon & Schuster.

Blumenberg, H. (1985). *Work on Myth.* Cambridge, MA: The MIT Press.

Bradner, S. (1999). The Internet Engineering Task Force. In C. DiBona, S. Ockman, & M. Stone (Eds.), *Open Sources: Voices from the Open Source Revolution.* (pp. 47-52). Sebastopol, CA: O'Reilly & Associates, Inc.

Brooks, F.P. (1995). *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley.

Brown, J.S., & Duguid, P. (1991). Organizational learning and communities of practice: toward a unified view of working, learning, and innovation. *Organization Science,* **2**, pp.40-57.

Brown, S.L., & Eisenhardt, K.M. (1995). Product development: past research, present findings, and future directions. *Academy of Management Review,* **20** *(2)*, pp.343-78.

Cole, M. (1996). *Cultural Psychology: A Once and Future Discipline.* Cambridge, MA: The Belknap Press of Harvard University Press.

Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience.* New York: HarperCollins Publishers.

Dewey, J. (1991). *How We Think.* Buffalo, NY: Prometheus Books.

DiBona, C., S. Ockman, & M. Stone. (1999). *Open Sources: Voices from the Open Source Revolution.* Sebastopol, CA: O'Reilly & Associates, Inc.

Engeström, Y. (1987). *Learning by Expanding: An Activity Theoretical Approach to Developmental Work Research.* Helsinki: Orienta Konsultit.

Freeman, C., J. Clark, & L. Soete. (1982). *Unemployment and Technical Innovation: A Study of Long Waves and Economic Development.* Westport, CT: Greenwood Press.

Glaser, B.G., & A.L. Strauss. (1967). *The Discovery of Grounded Theory: Strategies for Qualitative Research.* Hawthorne, NY: Aldine Publishing Company.

Griffin, A. (1997). PDMA research on new product development practices: updating trends and benchmarking best practices. *Journal of Product Innovation Management,* **14**, pp.429-58.

Hofstede, G. (1991). *Cultures and Organizations: Software of the Mind.* London: McGraw-Hill Book Company.

Lave, J., & E. Wenger. (1991). *Situated Learning: Legitimate Peripheral Participation.* Cambridge: Cambridge University Press.

Leont'ev, A.N. (1978). *Activity, Consciousness, and Personality*. Englewood Cliffs, NJ: Prentice-Hall.

Lynn, L.H., Aram, J.D., & Reddy, N.M. (1997). Technology communities and innovation communities. *Journal of Engineering and Technology Management,* **14**, pp.129-45.

Machlup, F., & Penrose, E. (1950). The patent controversy in the nineteenth century. *The Journal of Economic History,* **X** *(1),* pp.1-29.

Mahajan, V., & Wind, J. (1992). New product models: practice, shortcomings and desired improvements. *Journal of Product Innovation Management,* **9**, pp.128-39.

Miles, R.E., Snow, C.C., Mathews, J.A., Miles, G., & Coleman, H.J., Jr. (1997). Organizing in the knowledge age: Anticipating the cellular form. *Academy of Management Executive,* **11** *(4),* pp.9-24.

Mintzberg, H. (1979). *The Structuring of Organizations.* Englewood Cliffs, NJ: Prentice-Hall.

Nonaka, I., & H. Takeuchi. (1995). *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation.* Oxford: Oxford University Press.

Perens, B. (1999). The Open Source Definition. In C. DiBona, S. Ockman, & M. Stone (Eds.), *Open Sources: Voices from the Open Source Revolution.* (pp. 171-188). Sebastopol, CA: O'Reilly & Associates, Inc.

Perez, C. (1985). Microelectronics, long waves and world structural change: new perspectives for developing countries. *World Development,* **13** *(3),* pp.441-63.

Powell, W.W., Koput, K.W., & Smith-Doerr, L. (1996). Interorganizational collaboration and the locus of innovation: networks of learning in biotechnology. *Administrative Science Quarterly,* **41** *(1),* pp.116-45.

Raymond, E.R. (1999). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* Sebastopol, CA: O'Reilly & Associates, Inc.

Schon, D.A. (1963). *Invention and the Evolution of Ideas.* London: Social Science Paperbacks.

Simmel, G. (1990). *The Philosophy of Money. Second Enlarged Edition (first edition 1900).* London: Routledge.

Stallman, R. (1999). The GNU operating system and the free software movement. In C. DiBona, S. Ockman, & M. Stone (Eds.), *Open Sources: Voices from the Open Source Revolution.* (pp. 53-70). Sebastopol, CA: O'Reilly & Associates, Inc.

Tanenbaum, A.S., & A. Woodhull. (1997). *Operating Systems: Design and Implementation (2nd ed.).* Upper Saddle River, NJ: Prentice Hall.

Torvalds, L. (1999). The Linux edge. In C. DiBona, S. Ockman, & M. Stone (Eds.), *Open Sources: Voices from the Open Source Revolution.* (pp. 101-111). Sebastopol, CA: O'Reilly & Associates, Inc.

Tuomi, I. (1999). *Corporate Knowledge: Theory and Practice of Intelligent Organizations.* Helsinki: Metaxis.

Van de Ven, A.H. (1993). A community perspective on the emergence of innovations. *Journal of Engineering and Technology Management*, **10**, pp.23-51.

Von Hippel, E. (1988). *The Sources of Innovation.* New York: Oxford University Press.

Weizenbaum, J. (1984). *Computer Power and Human Reason: From Judgement to Calculation.* Harmondsworth: Penguin Books.

Wertsch, J.V. (1998). *Mind as Action.* Oxford: Oxford University Press.

Wiener, N. (1975). *Cybernetics: or Control and Communication in the Animal and the Machine.* Cambridge, MA: The MIT Press.