



## Table of Contents

<b>Introduction</b> .....	<b>1</b>
<b>I. The Problem and the Opportunity of Open Source</b> .....	<b>2</b>
<b>II. Traditional Economic Analyses of Open Source</b> .....	<b>3</b>
A. Macroeconomic approaches.....	3
B. Microeconomic approaches.....	5
<b>III. The Search for New Institutions: Case Studies in Open Source</b> .....	<b>8</b>
A. The Free Software Foundation.....	8
B. The Linux Operating System.....	9
C. Seven Key Features of Successful Open Source Projects.....	11
<b>IV. A New Political/Economic Logic</b> .....	<b>13</b>
A. Coordination and Leadership.....	14
B. Managing Complexity.....	16
C. Resolving Conflicts.....	18
<b>V. The Commercialization and Spread of Open Source?</b> .....	<b>21</b>
<b>Conclusion: Lessons from the Open Source Model</b> .....	<b>22</b>

# Introduction

---

Open source software, and the worldwide movement that gives rise to it, have recently attracted extraordinarily diverse hopes and fears about the social and economic consequences of the information revolution.

Libertarians see in open source a tool to emancipate individuals from governmental and corporate tyranny. Proponents of free markets see open source as the ultimate low-barriers-to-entry market, where only quality counts. Communitarians visualize a cross-national, cross-ethnic, and cross-just-about-every-other-traditional-boundary community, working together to advance the public good. Economists see a market in reputation, evolving naturally and almost automatically in a space with massively reduced transaction costs.

Why has the open source movement thus become the Internet era's Rorschach test? The answer is that open source is both a child of the network economy, and a potentially rich source of lessons for functioning within it. As such, the movement challenges much of what economists, lawyers, and businesspeople believe they know about modern capitalism and how intellectual property rights, production, and value-added are transformed into profit within it. The arguments and theories that explain why firms exist, why some knowledge is kept private and sold for a price, why some people earn higher salaries than others, and how groups of people can overcome the challenges of working together in pursuit of a common good, all may need to be reinterpreted in light of open source's success.

This GBN working paper takes a step in that direction. Building an explanation for open source requires a compound argument capable of reconciling the micro-foundations of traditional economic logic with the new social and political structures that replace "property rights" as the ordering constraints on production—structures characteristic of the open source movement.

The lessons we glean from this exercise may apply not only to non-commercial software production, but to production of other kinds of knowledge goods as well. In the long run, open source may be regarded as a harbinger of a networked, knowledge-based economy and the processes of production that work best within it.

# I. The Problem and the Opportunity of Open Source

---

The concept of “free software” is not new. In the 1960s and 1970s, making source code freely available was standard practice among researchers. The custom was largely taken for granted in leading computer science departments (such as those at MIT and UC Berkeley) and corporate research facilities (particularly Bell Labs and Xerox PARC).

Today, however, the vast majority of software production is organized under the economic logic imposed by an intellectual property rights system. Patents, copyrights, licensing schemes, and other means of “protecting” software code ensure that users cannot reverse-engineer, modify, or re-sell code developed by others. In this model, maintaining proprietary control over source code forms the cornerstone of profitability. Indeed, source code is probably the most valuable asset of a firm like Microsoft.

In parallel to the intellectual-property norm, a worldwide open source movement has been gathering steam since the early 1980s, premised on a fundamentally different economic model. Open source software is by definition “free”—public and non-proprietary. The Open Source Initiative, put forth in 1998 by a group of committed open source developers, specifies that software must share three essential characteristics if it is to be considered “open source”:

1. It must permit free redistribution of the software.
2. It must require that the full source code be distributed with any binaries.<sup>1</sup>
3. It must allow anyone to modify and redistribute their own versions under these same terms.<sup>2</sup>

Today, several thousand open source development projects are actively underway, ranging from small utilities and device drivers to more robust programs such as the e-mail transfer program Sendmail, the http server Apache, and even operating systems such as Linux. These projects are driven forward by contributions from hundreds, sometimes thousands, of developers who work around the world in a seemingly unorganized fashion, receiving neither direct pay nor other visible compensation for their contributions. Thwarting conventional economic logic, these collaborative open source projects demonstrate empirically that *large, complex systems of code* can be built, maintained, developed, and extended in *non-proprietary settings* in which *many developers* work in highly *parallel, relatively unstructured ways* and without *direct monetary compensation*.

Perhaps because the strength of this movement is so counterintuitive, tremendous uncertainty surrounds the question of what drives the open source model. Some observers have considered the phenomenon in broadly political or sociological terms, trying to understand the internal logic and external consequences of a geographically widespread community capable of producing

---

<sup>1</sup> Source code is in a common computer language that human programmers can read, write, understand, and modify. It is compiled into binary form (ones and zeros) that the computer can execute. Proprietary vendors distribute only this binary form of the program, so that no humans can understand or modify the code.

<sup>2</sup> Most open source licenses also require that the software itself be made available to others for no more than the cost of distribution. The terms of this definition originated in the Debian Social Contract, developed in the mid-1990s by Bruce Perens. [www.debian.org](http://www.debian.org)

excellent software without direct monetary compensation. In early writings about the movement by computer “hackers” who participate in open source projects (and, typically, who are true believers in the concept), open source has been characterized variously as:

- A methodology for research and development
- A new business model, requiring new mechanisms for compensation and profit
- The defining nexus of a community geared towards the development of common goods
- A new production structure unique to the knowledge economy
- And even a political philosophy

Does open source really hold as much promise as its proponents suggest? In the following pages we explore several interleaved dimensions of this question: open source’s role as a uniquely vigorous exemplar of production in the network economy, one in which conventional economic notions of profit and incentive do not apply and which has evolved distinct organizational and technical models in response to this idiosyncrasy.

Many of the innovations we will examine may be translatable back to the commercial sector, where they may help for-profit companies adapt to the peculiarities of the network economy. In addition, they may help developing economies, lacking high-tech infrastructures, to adapt to the global marketplace

## II. Traditional Economic Analyses of Open Source

---

### A. Macroeconomic approaches

1. *Collective action.* The starting point for most economic analyses of open source is a standard “collective action” approach.<sup>3</sup> In this context the economic puzzle is straightforward. For well-known reasons, nonexcludable public goods (ie something that cannot be kept out of the hands of anyone who wants to take it) tend to be underprovided in non-authoritative social settings. This is because the potential providers of such goods can access them as “free riders” in the public space; that is, without expending their own time or resources to create or distribute the goods. For similar reasons, such products also tend to be of uncertain or lower quality.

Open source products such as Linux ought to exist at the worst end of this spectrum, since they arise from collective voluntary action—yet, paradoxically, they occupy the high end. Mark Smith and Peter Kollock have gone so far as to call Linux “the impossible public good.”<sup>4</sup> While it requires contributions from a large number of developers, each developer has little incentive to contribute since he or she can partake of Linux unchecked as a free rider. Stark economic logic dictates that the system ought to unravel backwards, ensuring that no one makes any contributions, and there is no public good to begin with.

---

<sup>3</sup> See the intelligent if sometimes polemical critique by Eben Moglen, “Anarchism Triumphant: Free Software and the Death of Copyright,” *First Monday*, Issue 4

<sup>4</sup> Marc A. Smith and Peter Kollock, eds. *Communities in Cyberspace*. London: Routledge, 1999, p. 230.

2. *Non-rival goods*. Previous attempts to grapple with this paradox have focused on redefining the structural logic of economic exchange. Rishab Aiyer Ghosh, for example, introduces the notion of “non-rival” goods in order to circumvent the free-rider trap. Using the image of a cooking pot capable of cloning all food placed in it, Ghosh suggests that trade in non-rival goods is not plagued by the free-rider problem since the supply of these goods is inexhaustible. His cooking-pot analogy of course mirrors the Internet, where software, once uploaded, can be downloaded and copied an infinite number of times at essentially zero cost. The individual provider in this setting faces a different cost-benefit calculus than in a physical economy. As Ghosh explains, “You never lose from letting your product free in the cooking pot, as long as you are compensated for its creation.”<sup>5</sup> This compensation occurs as soon as even one other person contributes an item of some value to the pot, making participation utility-enhancing for all actors in the system. As Ghosh puts it, “if a sufficient number of people put in free goods, the cooking pot clones them for everyone, so that everyone gets far more value than was put in.”

The problem with this argument is that it does not fully explain the benefits inherent in participation. Strictly speaking, it is still a narrowly rational act for any single individual to take from the pot without contributing—thus free-riding on the contributions of others. The collective action dilemma remains unsolved. In the traditional explanation, after all, the system unravels not because free riders use up the stock of the collective good or somehow devalue it, but because there is no real incentive to contribute to that stock in the first place. The cooking pot starts—and remains—empty.

3. *Anti-rival goods*. The solution to this paradox lies in pushing Ghosh’s concept of non-rivalness one step further. In certain aspects of a network economy like the Internet, software is more than simply non-rival—it is what might be termed *anti-rival*, an awkward but nicely descriptive term. Most software programs, and particularly complex interdependent programs such as operating systems, are subject to positive network externalities: their value increases as more people download and use them. As more computers in the world run Linux, for example, it becomes easier for all users of Linux to share applications and files. Standardization and network compatibility provide one explanation for why this is so. Whether this phenomenon is called a network good or an anti-rival good, it helps clarify why open source developers would rather participate than free-ride: by making the program better for everyone they encourage more usership, which in turn makes it better for themselves.

But open source software makes an additional and very important use of network externalities. The more individuals—including free riders—actively use a piece of software, the easier debugging becomes as errors are quickly found and reported. Software development, too, speeds up as the user base grows. Programmers have more incentive to expend time building plug-ins and coding new features, knowing that free riders will leverage their contributions and advance the product’s exponential improvement.

In fact, software development in the commercial world takes place in precisely this way—people inside organizations write code to perform applications and solve problems identified by end users within their own organizations. The open source process, however, leverages this huge

---

<sup>5</sup> Ghosh, “Cooking Pot Markets,” p. 16.

reservoir of talent and energy usually closeted *within* organizations, by allowing it to be shared in a coordinated fashion *across* organizational boundaries.

Thus, open source software is not merely able to accommodate free riders. It is actually *anti-rival* in the sense that *the system positively benefits from what are typically thought of as free riders on a collective good*. Free riders provide value to the system whether they are reporting a bug out of frustration or, by their very presence, encouraging greater commercial support for the platform in general. This is the key point recognized by a high-level Microsoft memorandum of summer 1998. Known as the “Halloween Memo,” this directive pointed to open source software as a direct threat to Microsoft’s revenues and to its quasi-monopolistic position in some markets. As the author observed, open source software represents a long-term strategic threat to Microsoft because “the intrinsic parallelism and free idea exchange in OSS [Open Source Software] has benefits that are not replicable with our current licensing model.”

## **B. Microeconomic approaches**

The logic outlined above constitutes a structural explanation for the success of open source projects. However, it provides no explanation for why core groups of programmers arise and why they undertake a program’s initial development costs. A closer look at microeconomic incentives helps to address these questions.

*1. The signaling incentive.* Lerner and Tirole in their article “The Simple Economics of Open Source” make perhaps the most forceful argument.<sup>6</sup> They portray individual programmers, regardless of whether they work in open source or as employees of a proprietary software firm, as rational actors engaged in a straightforward cost-benefit transaction. The immediate benefits to a programmer are private ones: fixing a specific coding problem, or working for direct monetary benefit. The primary cost is the opportunity cost of the time and effort that the programmer expends on the project.

Open source alters this standard cost-benefit calculus in two significant ways. First, the “alumni effect” should lower the cost of working on open source relative to working on proprietary code.<sup>7</sup> Since Unix syntax and open source tools are a standard element of most programmers’ training, the costs of simply extending the functionality of these widely used tools *should* be lower than building proprietary solutions from scratch.

Second, open source offers delayed benefits to an open source programmer’s reputation and career, which in turn create a strong “signaling incentive” to keep performing the programming.<sup>8</sup> Ultimately, these benefits may be translated into money. The logic is as follows: solving difficult programming problems generates ego gratification, because it creates peer recognition. Peer recognition is important because it creates a reputation. And a reputation as a

---

<sup>6</sup> Josh Lerner and Jean Tirole, “The Simple Economics of Open Source,” NBER, Feb 25, 2000. This is an important paper that draws usefully on others’ analyses, while recognizing its own limitations as a preliminary exploration inviting further research.

<sup>7</sup> Lerner and Tirole, p. 11.

<sup>8</sup> Lerner and Tirole, p. 15.

great programmer is monetizable—in the form of job offers, privileged access to venture capital, and the like.

The key point is that open source programming is a uniquely powerful “signaling” mechanism for individual developers. As is true in many technical and artistic disciplines, the quality of a programmer’s mind and work is not easy to judge in standardized and easily comparable metrics. But an individual’s reputation within a well-informed, committed, and self-critical community is one proxy measure for this quality. Thus, the best programmers have a clear incentive to facilitate others’ access to their work, so others can see and understand just how good they are. Hence the importance of signaling: the programmer participates in an open source project as a strategic act of credentialing.

Lerner and Tirole argue that the signaling incentive is stronger when a performance is visible to the audience; when effort expended has a high impact on performance; and when performance yields good information about talent. Open source projects maximize the incentive along these dimensions, in several ways.

With open source, a software user can see not only how well a program performs; she can also look to see how clever and elegant is the underlying code—a much more fine-grained measure of the quality of the programmer. And since no one is forcing anyone to work on any particular problem in open source, the performance can be assumed to represent a voluntary act on the part of the programmer, which makes it that much more informative about that programmer. The signaling incentive should be strongest in settings with sophisticated users, tough bugs, and an audience that can appreciate effort and artistry and thus can distinguish between merely good solutions and excellent ones. As Lerner and Tirole note, this argument is bolstered by the fact that open source has flourished (at least initially) in more technical settings like operating systems rather than in end-user applications.

2. *The gift economy.* Alternate microeconomic explanations of open source paint individual incentives as a product of conventional social institutions. One of the more interesting is proposed by Ko Kuwabara.<sup>9</sup> Kuwabara uses a metaphor of complex adaptive systems and evolutionary change to describe the software development process. His account boils down to a series of causal steps. Programmers are motivated by a “reputation game,” similar to the one Lerner and Tirole depict. But he argues that the social structure shapes the individual incentives, not vice versa. Because online communities live in a situation of abundance, not scarcity, Kuwabara suggests that they develop “gift cultures” where social status depends on what you give away rather than on what you control.<sup>10</sup> Expand this into an evolutionary setting over time, and the community will self-organize a set of ownership customs along lines that resemble a Lockean regime of property rights. These ownership customs constitute a sufficient framework for successful and productive collaboration, even if they do not involve explicit legal control over property.

---

<sup>9</sup> Ko Kuwabara, “Linux: A Bazaar at the Edge of Chaos,” *First Monday*, March 2000.

<sup>10</sup> The “gift culture” argument is taken principally from Eric Raymond, “Homesteading the Noosphere.” See also David Baird, “Scientific Instrument Making, Epistemology, and the Conflict Between Gift and Commodity Economies,” *Philosophy and Technology* 2, Spring-Summer 1997.

The gift culture hypothesis is an important one. Gift economies—where social status depends more on what you give away than what you keep—are reasonable adaptations to conditions of abundance. They are often seen among aboriginal cultures living in mild climates and ecosystems with abundant food, as well as among the extremely wealthy in modern industrial societies.<sup>11</sup> And the culture of gift economies shares some notable characteristics with that of open source communities: gifts bind people together, encourage diffuse reciprocity, and support a concept of property that resembles “stewardship” more than ownership per se.

This cultural argument is likewise strongly evident in the writing of Eric S. Raymond, the unofficial ethnographer of the open source movement. In his piece “Homesteading the Noosphere,” Raymond suggests that gift culture logic works particularly well in software development, since the value of the gift (in this case a complex technical artifact) cannot be easily measured except by other members of the community, who have the expertise to evaluate its technological sophistication. Naturally, therefore, “the success of a giver’s bid for status is delicately dependent on the critical judgement of peers.”<sup>12</sup>

While the culture of open source shares some of the characteristics of a gift economy, its incentives cannot be defined exclusively in terms of the social framework. “Abundance” in a knowledge economy like open source software development is essentially different from abundance in a physical economy. Of course the physical tools of programming—bandwidth, disk space, and processing power—are plentiful and cheap, and doubtless each will grow more abundant and cheaper over time. Yet when anyone can have a supercomputer on his or her desk, there is little status associated with that “property.” The very abundance of computing power should, in fact, devalue it. Rather, the things that add value in this setting depend on *human mindspace and the commitment of time and intellectual energy by very smart people to a creative enterprise*. It is the time and brainspace of smart, creative people that is scarce, and probably becoming more scarce as demand for their talents increases in proportion to the computing power available. Great programming skills are extremely rare. Nor is a *reputation* for greatness typically abundant, because only a certain number of people can really maintain a reputation for being “great coders” at any given point in time.<sup>13</sup>

---

<sup>11</sup> Raymond, “Homesteading the Noosphere,” p. 99.

<sup>12</sup> Raymond, “Homesteading the Noosphere,” p. 103.

<sup>13</sup> I say this because standards of “greatness” are themselves endogenous to the quality of work produced in a particular population. If there is a normal distribution of quality, and the bell curve shifts to the right, what would have been “excellent” in the past is now merely good.

### III. The Search for New Institutions: Case Studies in Open Source

---

Macroeconomic approaches do not explain the motivations of individual programmers. Micro-level arguments about utility functions do not follow directly from exogenous social structures. A static conception of property rights has traditionally allowed analysts to bridge this gap. Under its logic, the level-of-analysis problem can be sidestepped because pressures on both levels are expressed in terms of a common independent variable: money.

But because open source development is not conditioned by a traditional logic of property rights, bridging the gap between macro- and micro-level approaches is no longer automatic. Any successful explanation must therefore identify the logic of the particular software licenses and other social constraints that effectively replace standard systems of property rights as the fundamental ordering principles. It is this task to which this paper now turns, by examining in greater depth how two open source communities actually work: the Free Software Foundation, and the Linux development community.<sup>14</sup>

From these analyses we will then draw general conclusions about successful open source development and how it might be applicable to other knowledge-based production processes.

#### A. The Free Software Foundation

Steven Levy's book *Hackers* gives a compelling account of the impact on the software programming community, particularly at MIT, of intellectual property rights. With the unbundling of software from hardware in the mid-1970s, many of the best programmers at MIT were hired away into lucrative positions in spin-off software firms. MIT began to demand that its employees sign nondisclosure agreements as a condition of using the university's computing facilities. The newest mainframes, such as the VAX or the 68020, came with operating systems that did not distribute source code—in fact researchers had to sign nondisclosure agreements simply to get an executable copy.

MIT researcher Richard Stallman led the backlash. Driven by moral fervor as well as simple frustration at not being able to easily modify software for his particular needs (such as fixing a printer driver), Stallman in 1984 founded a project to revive the “hacker ethic” by creating a complete set of “free software” utilities and programming tools.<sup>15</sup> Called the Free Software Foundation (FSF), this project aimed to develop and distribute software under what he called the General Public License (GPL), also known in a clever word-play as “copyleft.”

---

<sup>14</sup> These two examples attract a great deal of public attention but are by no means the only important examples. My forthcoming book, *The Success of Open Source*,<sup>2</sup> examines these and others in much more detail.

<sup>15</sup> In Stallman's view, “the sharing of recipes is as old as cooking” but proprietary software meant “that the first step in using a computer was a promise not to help your neighbor.” He saw this as “dividing the public and keeping users helpless.” (“The GNU Operating System and the Free Software Movement,” in Chris DiBona, Sam Ockman, and Mark Stone, eds. *Open Sources: Voices from the Open Source Revolution*. Sebastopol: O'Reilly, 1999, p. 54. For a fuller statement see [www.gnu.org/philosophy/why-free.html](http://www.gnu.org/philosophy/why-free.html).)

In a reverse of conventional copyrighting, GPL aims to prevent cooperatively developed software or any part of that software from being turned into proprietary software. Users are permitted to run the program, copy the program, modify the program through its source code, and distribute modified versions to others. What they may *not* do is add restrictions of their own. This is the “viral clause” of GPL: it compels anyone releasing software that incorporates copylefted code to use the GPL in their new release. The FSF’s version of GPL stipulates: “You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program [any program covered by this license] or any part thereof, to be licensed as a whole at no charge to all third parties *under the terms of this license.*”<sup>16</sup>

Stallman and the FSF have created some of the most widely used pieces of Unix software, including the text editor EMACS, the GCC compiler, and the GDB debugger. As these popular programs were adapted to run on almost every version of Unix, their availability and efficiency helped cement Unix as the operating system of choice for free-software advocates. But the FSF’s success was in some sense self-limiting. Partly this is because of the moral fervor underlying Stallman’s approach—not all programmers found his strident libertarian attitude practical or helpful. Partly it was a marketing problem. “Free software” turned out to be an unfortunate label, despite FSF’s vehement attempts to convey the message that free was about freedom, not price—as in the slogan “think free speech, not free beer.”

But there was a deeper problem in the all-encompassing nature of the GPL and its “viral” clause. Stallman’s moral stance against proprietary software clashed with the utilitarian view of many programmers, who wanted to use pieces of proprietary code along with free code when it made sense to do that, simply because the proprietary code was technically good. The GPL did not permit this kind of flexibility and thus posed difficult constraints to developers looking for pragmatic solutions.

## **B. The Linux Operating System**

The history of Linux provides more insight into this phenomenon. Linus Torvalds, a computer science student at the University of Helsinki, strongly preferred the technical approach of Unix to the DOS operating system commercialized by Microsoft.<sup>17</sup> But Torvalds did not like waiting in long lines for access to the limited number of university machines that ran Unix for student use. And it simply wasn’t practical to run a commercial version of Unix on a personal computer—the software was too expensive, and also much too demanding for the limited PCs of the time.

In late 1990 Torvalds came across Minix, a simplified Unix clone that was being used for teaching purposes at Vrije University in Amsterdam. Minix ran on PCs, and the source code was available. Torvalds installed this system on his IBM-AT, a machine with a 386 processor and 4MB of memory, and went to work building the kernel of a Unix-like operating system with Minix as the scaffolding. In autumn 1991, Torvalds let go of the Minix scaffold and released the source

---

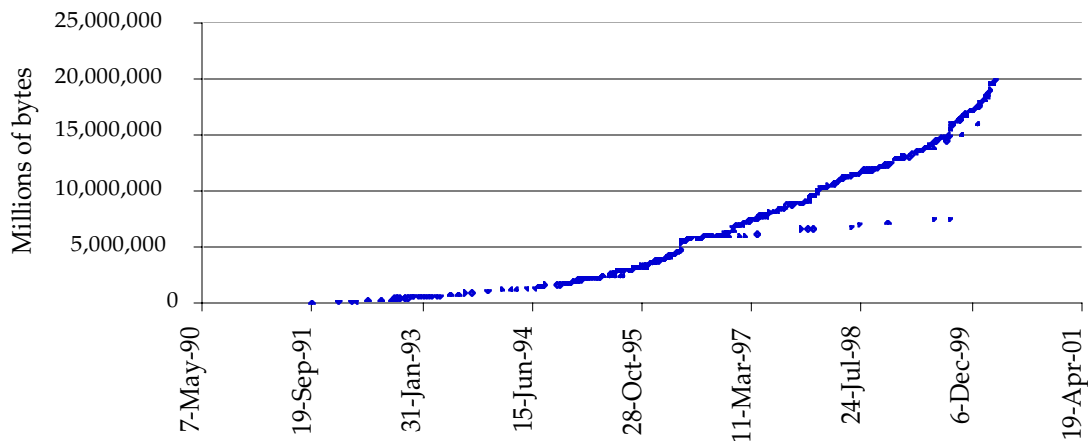
<sup>16</sup> Free Software Foundation, “GNU General Public License, v. 2.0,” 1991. [www.gnu.org/copyleft/gpl.html](http://www.gnu.org/copyleft/gpl.html). Emphasis added. Several modifications to these provisions now exist, but the general principle remains.

<sup>17</sup> Torvalds was particularly interested in “task-switching,” a major feature of Unix which is not available in DOS. Unix allows the computer to switch between multiple processes running simultaneously.

code for the kernel of his new operating system, which he called Linux, onto an Internet newsgroup. He appended the following note:

I'm working on a free version of a Minix look-alike for AT-386 computers. It has finally reached the stage where it's even usable (though it may not be, depending on what you want), and I am willing to put out the sources for wider distribution.... This is a program for hackers by a hacker. I've enjoyed doing it, and somebody might enjoy looking at it and even modifying it for their own needs. It is still small enough to understand, use and modify, and I'm looking forward to any comments you might have. I'm also interested in hearing from anybody who has written any of the utilities/library functions for Minix. If your efforts are freely distributable (under copyright or even public domain) I'd like to hear from you so I can add them to the system.<sup>18</sup>

**Figure 1**



The response to Linux was extraordinary (and according to Torvalds, mostly unexpected). By the end of the year nearly 100 people worldwide had joined the Linux newsgroup. Through 1992 and 1993 the community grew steadily. New users downloaded it, played with it, tested it in various settings, and attempted to extend and refine it. Flaws surfaced in the form of bugs and security holes, while new features were continually added. Users submitted reports of problems they found, or proposed a fix and sent a patch on to Torvalds. Gradually, the process iterated and scaled up to a degree that just about everyone, including its most ardent proponents, found startling. In 1994 Torvalds finally released the first official version of Linux (version 1.0). Thereafter, the pace of development accelerated, with updates to the system being released on a weekly, or sometimes even a daily basis. Figure 1 illustrates the dramatic growth of the operating system, which today consists of more than three million lines of code.

This rapid growth is attributable to an extremely large and geographically far-flung community. Indeed, the credits file for the original release names contributors from at least 31 different countries. In both FSF and Linux circles, as in most open source communities, a large number of

<sup>18</sup> Linus Torvalds, "Linux History," 1999. [www.li.org/li/linuxhistory.html](http://www.li.org/li/linuxhistory.html)

moderately committed individuals who contribute irregularly complement a smaller but much more committed core group.

A July 2000 survey of the open source community identified approximately 12,000 developers working on Linux. Although the survey acknowledges difficulties with measurement, it reports that the top 10 percent of the developers are credited with about 72 percent of the code—loosely parallel to the apocryphal “80/20 rule” (where 80 percent of the work is done by 20 percent of the people).<sup>19</sup> Linux user/developers come from all walks of life: hobbyists; people who use Linux or related software tools in their work; committed “hackers.” Some have full-time jobs, others don’t.

### **C. Seven Key Features of Successful Open Source Projects**

In both the FSF and Linux cases, the logic behind the process is both functional and behavioral. Development occurred largely through a game of trial-and-error by people embedded in an increasingly self-aware community. Observers and participants (particularly Eric Raymond) have analyzed this evolving process and tried to characterize (inductively for the most part) the key features that made the process work. Drawing largely on Raymond’s analysis as well as on my own set of interviews, I propose seven key features common to development in successful open source projects.

#### **1) People pick important problems and make them interesting.**

Open source user/developers tend to work on projects that they judge to be significant. There is also a premium for what in the computer science vernacular is called “cool,” which roughly means creating a new and exciting function, or doing something in a newly elegant way. There seems to be an important and somewhat delicate balance around how much and what kind of work is done upfront by the project leader(s). User/developers look for signals that a project will actually generate a significant product, not an evolutionary dead end, and also will contain interesting challenges along the way.

#### **2) Developers look for solutions to their own most pressing problems.**

Raymond emphasizes that since there is no central authority or formal division of labor, open source developers are free to pick and choose exactly what they want to work on. This means they tend to focus on an immediate and tangible problem that they themselves need to solve—an “itch that needs to be scratched.” The Cisco enterprise printing system (an older open source-style project) evolved directly out of an immediate problem—system administrators at Cisco were spending an inordinate amount of time (in some cases half their time) working on printing problems.<sup>20</sup> Developers sometimes put out requests for cooperation, as in, “Isn’t there somebody out there who wants to work on ‘X’ or try to fix ‘Y’?” The odds are that in a community as large as the open source world, someone will want to scratch most any itch that arises.

---

<sup>19</sup> See Rishab Ghosh and Vipul Ved Prakash, “The Orbiten Free Software Survey,” *First Monday*, July 2000. Specifically regarding Linux, as of Spring 2000 there were approximately 90,000 registered Linux users, a large proportion of whom have programmed at least some minor applications or bug fixes; as well as a core of over 300 central developers who have made major and substantial contributions to the kernel.  
[www.linux.org/info/index.html](http://www.linux.org/info/index.html).

<sup>20</sup> <http://ceps.sourceforge.net/index.shtml>

### **3) Reuse whatever you can.**

Open source user/developers are always searching for efficiencies. Put simply, because they are not paid directly for contributions, they have a strong incentive never to reinvent the wheel. An important related point is that there is relatively little pressure on them to do so, because under the open source rubric they will always have access to the source code and thus will not need to recreate any tools or modules that are already available in open source.

### **4) Use a parallel process to solve problems.**

If a project has an important problem, a significant bug, or a feature that has become widely desired, many different people or perhaps teams of people will be working on it—in many different places, at the same time. They will likely produce a number of different potential solutions. The best solution can then be incorporated into the software, where it can be refined further.

Is this inefficient and wasteful? That depends. The relative efficiency of massively parallel problem solving depends on lots of parameters, most of which cannot feasibly be measured. As in nature, evolution in the open source world is messy. Given the right parameters, the open source community's size, diversity, and decentralization may make it an exceptionally efficient breeding ground for the messiness of software testing. As Paul Vixie puts it, “the essence of field testing is *lack of rigor*.”<sup>21</sup>

What is clear is that the stark alternative—some imaginary near-omniscient authority, capable of predicting which route will deliver the best payoff before any route is taken—is not a realistic option in complex systems.

### **5) Leverage numbers.**

The Linux process relies on a kind of law of large numbers to generate and identify software bugs, and then to fix them. Even a moderately complex program has a functionally infinite number of paths through the code, and only some tiny proportion of these paths will be generated by any particular user or testing program. The key to success with the vast user base is to generate patterns of use—the real world experiences of real users—that are inherently unpredictable by developers. In the Linux process, the huge group of users constitutes essentially an ongoing huge group of beta testers.

Eric Raymond says, “Given enough eyeballs, all bugs are shallow.”<sup>22</sup> Contained in this oft-quoted aphorism is a related point: given enough eyeballs and hands doing different things with a piece of software, more bugs will appear—and that is a good thing, because a bug must appear and be characterized before it can be fixed. Torvalds reflects on his experience over time that the person who runs into and characterizes a particular bug, and the person who later fixes it, are usually not the same person—an observational piece of evidence for the efficacy of parallel debugging.

---

<sup>21</sup> Paul Vixie, “Software Engineering,” in *Open Sources*, p. 98. Emphasis added.

<sup>22</sup> Raymond, “The Cathedral and the Bazaar,” p. 41.

#### **6) Write code that others can understand and document it well.**

In a sufficiently complex program, even open source code may not necessarily be transparent in terms of precisely what the writer was trying to achieve and why. The Linux process depends upon making these intentions and meanings clear, so that future user/developers understand (without having to reverse-engineer) what functions a particular piece of code plays in the larger scheme of things. Documentation is time-consuming and sometimes boring. But it is considered essential in any scientific research enterprise, in part because replicability is a key criterion.

#### **7) “Release early, release often.”**

User/developers need to see and work with iterations of software code in order to leverage their debugging potential. Commercial software developers understand just as well as open source developers do that users are often the best testers, so the principle of “release early, release often” can make as much sense in the commercial environment. The countervailing force in the commercial setting, of course, is customer expectations: customers are paying for reliability and ease of use, i.e., specifically *not* to expend their time and learning capacity on fixing bugs or updating frequently.

Open source user/developers have a very different set of expectations. For them, bugs are an opportunity rather than a nuisance.<sup>23</sup> In this setting, without the consumer middleman, software projects typically have a feedback and update cycle an order of magnitude faster than those of commercial products. In the early days of Linux, new releases of the kernel came out weekly—sometimes even daily!

## IV. A New Political/Economic Logic

The open source developer community is neither leaderless nor uniformly hierarchical. Rather, it blends clear lines of authority at the core with a high degree of creativity, self-government, and voluntary participation both inside and outside that core.

In larger projects, the formal organization of authority is quite structured. Libertarian and anarchist accounts of open source software development are frankly naïve. In the Linux community, Torvalds sits atop a decision pyramid as a de facto benevolent dictator. Apache is governed by a committee.

To understand the new logic of production exemplified by open source, this section examines how open source communities grapple with three fundamental problems:

- How to coordinate their efforts
- How to manage complexity
- How to resolve conflicts

---

<sup>23</sup> Linux kernel releases are typically divided into “stable” and “developmental” paths. This gives users a clear choice: download a stable release that is more reliable or a developmental release where new features and functionality are being introduced and tested.

Embedded in these overarching issues is a series of practical questions, from who has the right to make decisions about the development of code, to who gets credited for what work, and how conflicts are resolved when they arise.

## A. Coordination and Leadership

1. *Self-organization.* In a conventional system of property rights, authority within a firm and the price mechanism across firms are standard ways to efficiently coordinate specialized knowledge. Neither of these mechanisms exists in an open source community, where legal ownership is extremely fluid. A simple analogy from ecology suggests what might happen over time as modifications accumulate along different branching chains of software. Speciation—or what computer scientists call *code-forking*—seems likely. Lacking constraints of formal ownership or copyright, and given the explicit freedom to modify software code in any way that a user finds desirable, the software *should* evolve into branching incompatible versions, and synergies in development *should* be lost over time. This is very much what happened to Unix in the 1980s.<sup>24</sup> And if ego is a primary determinant of individual behavior, the temptation to code-fork away from the main branch of development must be particularly inviting to smart, creative, autonomous programmers. Why don't they more frequently depart from—or even worse, try to undermine—the collective project?

The explanation is not exclusively cultural/structural. Macroeconomic incentives connected to positive network externalities are part of the answer. If developers think of themselves as trading innovation for others' innovation, they will want to do their trading in the most liquid market possible. Forking would only reduce the size and thus the liquidity of the market. Viewing software as an “anti-rival” good implies a similar dynamic: the more open a project is and the larger its community of developers, the less tendency there will be to fork because each participant has the potential to add unique value, in addition to the value of sheer presence.

Finally, reputation comes into play. The potential forker's credibility is at stake in both attracting a talented and effective sub-group of developers, and in proving that the new group constitutes a viable and valuable alternative to the main one. Clearly, operating with such diminished resources, this forked development community could never credibly promise to match the rate of innovation taking place in the primary code base. It could not use, test, and debug software as quickly. And as a result it could not provide as attractive a “signaling incentive”—i.e., a payoff in reputation—to its developers, even if reputation were shared out more evenly within the forked community.<sup>25</sup>

Cultural and social norms do play a key role in the way these macro- and microeconomic pressures play out.<sup>26</sup> One prevalent norm in the community assigns decision-making authority based on the principle: *authority derives from responsibility*. The more an individual contributes

---

<sup>24</sup> The next section explores this history in more detail.

<sup>25</sup> Clearly there are parameters within which this argument is true. Outside of those parameters it could be false. It would be possible to construct a simple model to capture the logic, but it is hard to know—other than by observing the behavior of developers in the open source community—how to attach values to those parameters.

<sup>26</sup> Robert C. Ellickson provides a compelling argument about the falsifiability of normative explanations in *Order Without Law: How Neighbors Settle Disputes*. Cambridge MA: Harvard University Press, 1991, p. 270.

to a project and takes responsibility for pieces of software, the more decision-making authority that individual is granted by the community. In the case of Linux, Torvalds typically validates the grant of authority to “lieutenants” by consulting closely with them on an ongoing basis, particularly when it comes to key decisions about how subsystems are to work together in the software package.

2. *Leadership.* Leadership matters in setting a focal point, maintaining coordination around it, and giving the final word in disputes. Torvalds started the Linux process by providing a core piece of code. This was the original focal point. It functioned that way because, simplistic and imperfect as it was, it established a plausible promise of creativity and productivity: it *could* develop into something elegant and useful. The code contained interesting challenges and programming puzzles to be solved. Together, these characteristics attracted developers, who by investing time and effort on this project placed a smart bet that their contributions would be efficacious and that there would eventually be a valuable outcome.

In the longer term, leadership matters by reinforcing the cultural norms. Torvalds does, in fact, have many characteristics of a charismatic leader in the Weberian sense. He himself provides a convincing model of how to manage the potential for ego-driven conflicts among very smart developers. He downplays his own importance in the story of Linux; while acknowledging that his decision to release the code was an important one, he does not claim to have planned the whole thing or to have foreseen the significance of what he was doing or what would happen:

The act of making Linux freely available wasn’t some agonizing decision that I took from thinking long and hard on it; it was a natural decision within the community that I felt I wanted to be a part of.<sup>27</sup>

When it comes to reputation and fame, Torvalds is not shy and does not deny his status in any way. But he does make a compelling case that he was not motivated by fame and reputation—these are things that simply came his way as a result of doing what he believed in.<sup>28</sup> As his principle motivation, he continues to emphasize the fun and opportunities for self-expression in “the feeling of belonging to a group that does something interesting.” And he continues to invest huge effort in maintaining his reputation as a fair, capable, and thoughtful manager. It is striking how much effort Torvalds puts into justifying to the community his decisions about Linux, and documenting the reasons for his decisions in the language of technical rationality that is currency for this community. Would a leader with a more imperious attitude, who took advantage of his or her status to make decisions by fiat, have undermined the Linux community? Many in the community believe so (or believe that developers would exit and create a new community along more favorable lines).<sup>29</sup> The logic of the argument to this point supports that belief.

3. *Sanctioning.* Sanctioning mechanisms do exist to support the nexus of incentives, cultural norms, and leadership roles that sustain coordination. In principle, the GPL and other licenses

---

<sup>27</sup> “What Motivates Free Software Developers,” interview with Linus Torvalds, *First Monday*, Issue 3.

<sup>28</sup> The documented history, particularly the archived e-mail lists, supports Torvalds on this point.

<sup>29</sup> Examples of this process in my forthcoming book.

could be enforced through legal remedies (and this threat may lurk and constrain behavior even if it is not invoked). In practice, no one knows precisely how enforceable in the courts some aspects of these licenses would be.<sup>30</sup>

The sanctioning mechanisms most visibly practiced within the community are two: “flaming” and “shunning.”<sup>31</sup> Flaming is “public” condemnation (usually over e-mail lists) of people who violate norms. “Flamefests” can be quite fierce in language and intensity, but tend ultimately to be self-limiting.

Shunning is the more functionally important sanction. To shun someone—refusing to cooperate with them after they have broken a norm—cuts them off from the benefits offered by the community. It is not the same as exclusion: someone who is shunned can still use Linux. But that person will suffer substantial reputational costs. They will find it hard to gain cooperation from others. The threat is to be left on your own to solve problems, without the community’s collective experience and knowledge. This is clearly a strong disincentive to code-forking, for example, but it also constrains other less egregious forms of counter-normative behavior (such as aggressive self-promotion).

## **B. Managing Complexity**

Designing robust, complex software is a gargantuan task. Testing, debugging, and maintaining code is generally even harder. The standard industrial response to the increasing complexity of software has been to organize labor within a centralized, hierarchical structure—i.e., that of a firm. The firm then manages complexity through formal organization and explicit decisional authority.<sup>32</sup>

With complex knowledge goods, this is a very imperfect solution. In *The Mythical Man-Month*, a classic study of the social and industrial organization of programming, Frederick Brooks noted that when large organizations add manpower to a software project that is behind schedule, the project typically falls even further behind schedule.<sup>33</sup> He explained this with an argument now known colloquially as Brooks’ Law: as you raise the number of programmers on a project, work performed scales linearly by a factor of  $n$ , but complexity, communication costs, and vulnerability to error scales geometrically by a factor of  $n$  squared. This is inherent in the logic of the division of labor—the number of potential communications paths and interfaces between developers increases exponentially as the number of developers increases linearly. How does the open source process manage the implications of Brooks’ Law among a geographically dispersed community that is not subject to hierarchical command and control?

---

<sup>30</sup> David McGowan, “Copyleft and the Theory of the Firm,” See also Robert P. Merges, “The End of Friction? Property rights and Contract in the “Newtonian” World of On-Line commerce. (Digital Content: New Products and New Business Models) *Berkeley Technology Law Journal* v12, n1 (Spring, 1997):115-136.

<sup>31</sup> Eric Raymond, “Homesteading the Noosphere,” p. 129.

<sup>32</sup> Of course organization theorists know that a lot of management goes on in the interstices of this structure, but the structure is still there to make it possible.

<sup>33</sup> Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Reading Ma: Addison Wesley, 1975.

1. *The Linux solution.* Eric Raymond draws a too-stark contrast between “cathedrals” and “bazaars” as icons of organizational structure. Cathedrals are designed from the top down, built by coordinated teams who are tasked by and answer to a central authority. Open source projects seem to confound this hierarchical model. Linux appears, at least on first glance, to appear much more like a “great babbling bazaar of different agendas and approaches.”<sup>34</sup> But there has evolved within the Linux community a clear hierarchy of decision-making authority, where a decision pyramid leads from the dispersed developer base up through trusted lieutenants who have authority over particular parts of the code, and ultimately to Linus Torvald whose decisions are effectively final. This hierarchy was instituted in the mid-1990s, precisely in response to the growth of the project beyond a point where Torvalds could realistically manage the complexity on his own. (Programmers explain this with the sly phrase “Linus doesn’t scale.”) In practice, some of his authority has devolved down the rungs of the hierarchy and decisions made at those levels in effect bear his imprimatur. There is more hierarchical authority here than the popular image of a bazaar captures, even though it rests neither on corporate command and control nor on the power of money.

Complementing this line of authority, Linux’s public or general user base can and does propose “check-ins,” modifications, bug fixes, new features, and so on. There is no formal distinction between users and developers (a fact amply represented by the many Linux archive sites, which take submissions from literally anyone). There are low barriers to entry for debugging and development. This is true in part because of a common debugging methodology and in part because when a user installs Linux the debugging/developing environment comes with it (along with the source code, of course). Some users engage in “impulsive debugging”—fixing a little problem (shallow bug) that they encounter in daily use; while others make debugging and developing Linux a hobby or vocation.

2. *Other solutions.* The Linux decision-making system is just one example of pragmatic, experimental adaptations to this problem. In fact open source communities manage complexity in diverse ways. Consider the case of the Berkeley Software Distribution (BSD) model, another open-source flavor of Unix.<sup>35</sup> In BSD, typically, a relatively small, committed team of developers writes code. Users may modify the source code for their own purposes, but the development team does not generally take “check-ins” from public users, and there is no regularized process for doing that. Apache takes contributions from a wider swathe of developers who rely on a decision-making committee that is constituted according to formal rules, a de facto constitution. The Perl scripting language relies on a “rotating dictatorship” where control of the core software is passed from one member to another inside an inner circle of key developers.

3. *Modular design.* Within the software itself, the key to managing the level of complexity is modular design. This technical model directly influences the organizational model. A major tenet of the Unix philosophy, passed down to Linux, is to keep programs small and unifunctional (“do one thing simply and well”). A small program will have far fewer features than a large one, but will be easy to understand and easy to maintain, will consume fewer hardware system resources,

---

<sup>34</sup> Eric Raymond, “The Cathedral and the Bazaar,” in *The Cathedral and the Bazaar: Musings On Linux and Open Source by an Accidental Revolutionary*. Sebastopol: O’Reilly Publishing, 1999, p. 30.

<sup>35</sup> There are now several BSD projects, which I discuss in detail in my forthcoming book.

and—most importantly—can be combined with other small programs to enable more complex functionalities.

The technical term for this development strategy is “source code modularization.” A large program works by calling on relatively small and relatively self-contained modules. Good design and engineering is about limiting the interdependencies and interactions between modules. Programmers working on one module know two things: that the input and output of their module must communicate successfully with other modules, and that (ideally) they can make changes in their own module to debug it or improve its functionality without requiring changes in other modules, as long as they get the communication interfaces right. This reduces the complexity of the system overall because it limits the reverberations that might emanate from a code change.

These engineering principles are important because they reduce organizational demands on the political structure of the community. They comprise a powerful framework for autonomous actors working in parallel on many different parts of a project at once, since a programmer can control the development of a specific module of code without creating problems for other programmers working on other modules. When Linux moved to this model (from a monolithic kernel) in release 2.0, Torvalds wrote that “managing people and managing code led to the same design decision. To keep the number of people working on Linux coordinated, we needed something like kernel modules.”<sup>36</sup>

### C. Resolving Conflicts

While relatively high levels of trust may reduce the amount of conflict in the system, complicated and informal arrangements of this kind are certain to generate disagreements. Anyone who has dabbled in the software community recognizes that a large number of very smart, highly motivated, self-confident, and deeply committed developers, trying to work together, creates an explosive mix. Conflict is common, even customary in a sense. It is not the lack of conflict in the open source process but rather the successful management of substantial conflict that needs to be explained—conflict that is sometimes highly personal and emotional as well as intellectual and organizational.<sup>37</sup>

Eric Raymond observes that conflicts center for the most part on three kinds of issues:

- Who makes the final decision if there is a disagreement about a piece of code
- Who gets credited for precisely what contributions to the software
- Who can credibly and defensibly choose to “fork” the code<sup>38</sup>

---

<sup>36</sup> Linus Torvalds, “The Linux Edge,” in *Open Sources*, p. 108.

<sup>37</sup> Indeed, this has been true from the earliest days of Linux. See for example the e-mail debate between Linus Torvalds and Andrew Tanenbaum from 1992, reprinted in *Open Sources* pp 221-251. Torvalds opens the discussion by telling Tanenbaum “you lose,” “linux still beats the pants off minix in almost all areas,” “your job is being a professor and a researcher: That’s one hell of a good excuse for some of the brain-damages of minix.”

<sup>38</sup> Eric Raymond, “Homesteading the Noosphere,” in *The Cathedral and the Bazaar*, pp. 79-137.

Similar issues of course arise when software development is organized in a corporate setting. Standard theories of the firm explain various ways potential conflicts are settled, or at least managed, by formal authoritative organizations.<sup>39</sup> The open source community depends on quite different norms.

1. *Seniority rules.* Open source developers prefer to settle major conflicts through a “battle to consensus.” Programmers devote extraordinary time and energy to this process, trying to convince each other that there are firm technical grounds for preferring one solution or development path to another. The process doesn’t always succeed—in part because the technical criteria are not definitive; in part because personalities get in the way. In these instances an additional, auxiliary norm gets called into play: *seniority rules*. As Raymond explains: “If two contributors or groups of contributors have a dispute, and the dispute cannot be resolved objectively, and neither owns the territory of the dispute, the side that has put the most work into the project as a whole...wins.”<sup>40</sup>

2. *Let the code decide.* But what does it mean to resolve a dispute “objectively”? The notion of objectivity draws on its own, deeper normative base. The open source developer community shares a general conception of technical rationality. Like all technical rationalities, this one exists inside a cultural frame. The cultural frame is based on shared experience in Unix programming. Unix was birthed on assumptions of compatibility between platforms, ease of networking, and positive network effects.<sup>41</sup> Unix programmers have a set of common standards for what is “good code” and what is not-so-good code.<sup>42</sup> These standards draw on pragmatism and experience—the Unix “philosophy” is a philosophy of what works and what has been shown to work in practical settings over time.

The Open Source Initiative codified this cultural frame by establishing a clear priority for pragmatic technical excellence over the kind of ideology or zealotry characterized by the FSF. A cultural frame based on engineering principles (not on an anti-commercial ideology), and focused on high reliability and performance, gained much wider traction within the developer community. It also underscored the rationality of technical decisions driven at least in part by the need to sustain successful collaboration—hence legitimizing the importance of “maintainable” code, “clean” interfaces, and clear and distinct modularity.<sup>43</sup> The preeminence of technical rationality in this framework is summed up in its creed: “Let the code decide.”

3. *The final authority.* When a conflict cannot be resolved through these norms, a conflict may be submitted to leadership. Of course, it is a style of leadership that has to justify itself and its

---

<sup>39</sup> David McGowan provides a good summary discussion in “Copyleft and the Theory of the Firm,” University of Michigan Law School, May 29 Manuscript, forthcoming in *Illinois Law Review*.

<sup>40</sup> Raymond, “Homesteading the Noosphere,” p. 127. One interesting additional piece of evidence for these norms is what has happened when the two norms point in different directions. Raymond recalls one such fight of this kind and says “it was ugly, painful, protracted, only resolved when all parties became exhausted enough...I devoutly hope I am never anywhere near anything of the kind again” (p. 128).

<sup>41</sup> Indeed, Unix was developed in part to replace ITS (incompatible time sharing system). The idea in 1969 was that hardware and compiler technology were getting good enough that it would now be possible to write portable software—to create a common software environment for many different types of machines.

<sup>42</sup> Mike Gancarz, *The Unix Philosophy*.

<sup>43</sup> Ilkka Tuomi, “Learning from Linux.”

decisions to skeptical, independent-minded followers who are free to break away if they so choose.

Linux, in its earliest days, was run unilaterally by Linus Torvalds. Torvalds' decisions were essentially authoritative. As the program and the community of developers grew, Torvalds delegated responsibility for sub-systems and components to other developers, known as "lieutenants." Some of the lieutenants onward-delegate to "area owners" who have smaller regions of responsibility. The organic result looks and functions very much like a hierarchical organization where decision making follows fairly structured lines of communication. Torvalds sits atop the hierarchy as a "benevolent dictator" with final responsibility for managing conflicts that cannot be resolved at lower levels.

Torvalds' authority rests on a complicated mix. History plays a part: as the originator of Linux, Torvalds has a presumptive claim to leadership that is deeply respected by others. Charisma in the Weberian sense is also important, although it is notably limited by the fact that Torvalds goes to great lengths to document and justify his decisions about controversial matters. He admits when he is wrong. It is a kind of charisma that has to be continuously recreated through consistent patterns of behavior. Linux developers will also say that Torvalds' authority rests on its "evolutionary success." The fact is, the "system" that has grown up under his leadership has produced a first-class outcome. This in itself is a strong incentive not to fix what is clearly not broken.

Ultimately, decisions to accept Torvalds' authority can be traced back to definable incentives—but the incentives themselves depend heavily on the social structure created by the GPL license and by the constructed authority of the leader.

*4. Voluntary participation.* Conflict is expected; indeed it is normatively sanctioned. When an argument ends, as it is also expected to do at some point, the loser has essentially three options. She can accept the decision and move on; she can drop her involvement in the project; or she can fork the code.

If she drops out of the project, she loses the opportunity to accrue reputation and affect future decisions. The community loses the involvement of a particular individual, but not more than that (if she is an important individual, obviously the leader has strong incentives to try to heal the wound). Between the other two alternatives, the choice of whether to accept the decision or fork the code depends upon the calculations discussed under "Coordination and Leadership," above.

The open source development process builds momentum as it grows. The larger and more open a project, the higher the threshold for a rational decision to fork the code. The network externalities of the technology have essentially migrated into the social structure that surrounds it.

## V. The Commercialization and Spread of Open Source?

---

The success and increased visibility of open source software over the last several years have brought with them new pressures for formal organization. The process by which this has happened looks very much like (a limited) version of standard sociological accounts of institutional isomorphism. DiMaggio and Powell argue that institutions that interface frequently and deeply with each other will tend to adopt similar organizational structures as a means of improving communication and reducing a broad range of transaction costs.<sup>44</sup> As Linux increased in popularity at the end of the 1990s and was adopted by large commercial interests, key developers within the community began to argue that it was important to create an impression of organizational credibility for open source that would appeal to and reassure commercial users. Although Linux is now good enough to require less support, commercial users still worry a great deal about service and need reassurance that product support—even if it is informal in some sense—will be there, for the long term, when they need it.

In part due to these efforts, at the start of 2001 open source software is becoming a mainstream element of corporate information technology systems. In February 1998, a core group of open source developers joined together to create the Open Source Initiative. OSI is quite explicit about its goal: to establish a firm public relations base for open source software that is deeply credible to standard commercial users, and that lies outside the realm of morality and politics (particularly as those messages were associated with the Free Software Foundation). The organization's manifesto states: "We think the economic self-interest arguments for open source are strong enough that nobody needs to go on any moral crusades about it."<sup>45</sup> The Apache Software Foundation is now formally incorporated as a nonprofit and led by a board of directors.<sup>46</sup> Meanwhile, major IT companies including Hewlett Packard, Sun Microsystems, Motorola, and—most decisively—IBM have made corporate commitments to Linux, Apache, and other open source software for large computing systems, supercomputer equivalents, and new, small, handheld or household computing devices.

Some degree of institutional isomorphism reduces the complexity of relationships between the open source process and the growing number of organizations that use open source software. How far this process can and will go are important questions. They are particularly important given the huge financial stakes now manifest in for-profit companies like Red Hat and VA Linux, which are attempting to make money by packaging, marketing, supporting, assembling, refining, and ultimately creating open source-based solutions for the mainstream market.

---

<sup>44</sup> Paul J. DiMaggio and Walter W. Powell, "The Iron Cage Revisited: Institutional Isomorphism and Collective Rationality in Organizational Fields," *American Sociological Review* 48, 1983.

<sup>45</sup> <http://opensource.org/for-hackers.html#marketing>

<sup>46</sup> Directors are elected by members. Members are selected by existing members on the basis of "meritocracy, meaning that contributions and skills are the factors used to judge worthiness, candidates are expected to have proven themselves by contributing to one or more of the Foundation's projects." <http://www.apache.org/foundation/members.html>

## Conclusion: Lessons from the Open Source Model

---

Arguments about innovative business models in open source software are interesting and relevant, as are the various legal issues surrounding intellectual property rights and the licensing regimes.<sup>47</sup> The analytic risk is that by focusing too closely on these intriguing and immediate problems, we lose sight of a much bigger and ultimately more significant story about what open source represents in the emerging information economy.

The open source software movement is at once a product of the rising network economy, and an important source—perhaps the most important source available to us today—of organizational innovations for that economy. It is a functioning, pragmatic demonstration of a production process that is quite distinct from production modes characteristic of the pre-digital era. The difference is not one of degree but one of kind. This *new production process*, I believe, will turn out to be more significant than the software that is its immediate output.

The open source production process depends upon and uses the Internet to enable not just a more finely grained division of labor, but truly *distributed innovation*—a revolutionary way of thinking about economic production. A traditional capitalist division of labor, no matter how finely grained, still contains a value chain (even if part is located in Cupertino and part in Bangalore). The production challenge is still about getting the weakest link in that chain to deliver.

But in parallel distributed innovation, at the limit there is no weak link because there is no chain. Coordination costs may still constrain to some extent the functional distribution of innovation, thus constraining in turn the maturation of parallel processing as a mode of economic production. But that is now a function of getting the social organization “right,” not of technology per se.<sup>48</sup> And the open source community is at the forefront of practical experimentation with that social organization.

A fundamentally new production process will pose new and surprising challenges for existing economic and legal structures, across national/international divides. An immediate issue concerns the politics and particularly the international politics of standard-setting. These politics are typically analyzed around bargaining power between and among firms, national governments, and international organizations. Open source adds an interesting twist. The open source community is not a firm (although there are firms that may try to represent some of the interests of the community). And the community is not represented by a state, nor do its interests align particularly with any individual state. It bears no allegiance to any kind of international organization. The technological community that produces open source software was international from the start and remains highly international in scope. We simply do not know how this community will interact with formal standards processes, which are embedded deeply in national, international, and global politics. Certainly, existing national and international institutions will try to promote and manipulate the dynamic in ways that yield advantage to particular players; yet it

---

<sup>47</sup> I take up these arguments in my forthcoming book.

<sup>48</sup> The underlying argument is from F.A. Hayek, “The Use of Knowledge in Society,” *American Economic Review* 35, September 1945, pp. 519-530.

is difficult to see right now a viable strategy by which governments could achieve lasting advantage in this way.

New production processes bring new possibilities as well. The open source process intertwines community and commerce more tightly than almost any current e-business model has yet dared to consider. It offers possibilities for economic bootstrapping in developing countries which might otherwise be locked out of an information economy characterized by much more expensive tools, and thus by high barriers to entry. It may accelerate the trend toward ubiquitous computing, the embedding of smart systems into a broad range of products and processes in human life. These are just some of the more obvious possibilities.<sup>49</sup>

Ultimately the most intriguing question about open source is how this process of knowledge production and coordination will extend to other realms of production in the twenty first century economy. The key concepts—user-driven innovation that takes place in a parallel distributed setting, distinct forms and mechanisms of cooperative behavior, and the economic logic of “anti-rival” goods—are generic enough to suggest that software is not the only place where the open source process could flourish. The process of annotating the human genome, which is really just a complex piece of code—an “operating system” for a biological, carbon-based “processor,” if you will—is one obvious application where a new, knowledge-based production process could trump government-sponsored and commercial alternatives. If the open source production process is indeed a window into the revolutionary potential of a network economy, then this kind of application is likely to be only the beginning.

The next version of this paper will expand at length on these broader applications and implications.

*Dr. Steven Weber is a GBN practitioner specializing in the political economy of global economic change and a professor of political science at the University of California, Berkeley.*

---

Copyright © Global Business Network and Steven Weber, March, 2000. GBN publications are for the exclusive use of Global Business Network members. To request permission to reproduce, store in a retrieval system, or transmit this document in any form or by any means, electronic, mechanical, recorded or otherwise, please contact Global Business Network.

---

<sup>49</sup> I expand on these and others in my forthcoming book *The Success of Open Source*.