The Political Economy of Open Source Software

Steven Weber

BRIE Working Paper 140
E-conomy Project™ Working Paper 15
June 2000

This is a draft, so please do not cite or quote. Comments welcome and needed!

Steven Weber is Associate Professor of Political Science, University of Berkeley, California.
Sweber@socrates.berkeley.edu
510/642.4654

Generous support for production of the BRIE Working Papers Series was provided by the Alfred P. Sloan Foundation.

I. The Analytic Problem of Open Source

Coca-Cola sells bottles of soda to consumers. Consumers drink the soda (or use it in any other way they like). Some consumers, out of morbid curiosity, may read the list of ingredients on the bottle. But that list of ingredients is generic. Coca-Cola has a proprietary 'formula' that it does not and will not release. The formula is the knowledge that makes it possible for Coke to combine sugar, water, and a few other readily available ingredients in particular proportions and produce something of great value. The bubbly stuff in your glass cannot be reverse-engineered into its constituent parts. You can buy it and you can drink it, but you can't *understand* it in a way that would empower you to reproduce it or improve upon it and distribute your improved cola drink to the rest of the world.

The economics of intellectual property rights provides a straightforward rationalization of why the Coca-Cola production 'regime' is organized in this way. The problem of intellectual property rights is about creating incentives for innovators. Patents, copyrights, licensing schemes and other means of 'protecting' knowledge assure that economic rents are created and that some proportion of those rents can be appropriated by the innovator. If that were not the case, a new and improved formula would immediately be available for free to anyone who chose to look at it. The person who invented that formula would have no claim on the knowledge or any part of the profits that might be made from selling drinks engineered from it. The system unravels, because that person no longer has any 'rational' incentive to innovate in the first place.

The production of computer software has typically been organized under a similar regime.[1] You can buy Microsoft Windows and you can use it on your computer but you cannot reproduce it, modify it, improve it, and redistribute your own version to others. Copyright provides legal protections to these strictures, but there is an even more fundamental mechanism that stops you from doing this. Just as Coca-Cola does not release its formula, most software developers do not release their *source code*, the list of instructions in a programming language that comprise the recipe for the software. Source code is the essence of proprietary software. It is a trade secret. Proprietary source code is the fundamental reason why Microsoft can sell Windows for a non-zero price, and distribute some piece of the rents to the programmers who write the code -- and thus provide incentives for them to innovate.

Open Source software inverts this logic. The essence of open source software is that source code is 'free' -- that is -- open, public, non-proprietary. Open Source software is distributed with its source code. The Open Source Definition (which I discuss in greater detail later) has three essential features:

- It allows free re-distribution of the software without royalties or licensing fees to the author
- It requires that source code be distributed with the software or otherwise made available for no more than the cost of distribution
- It allows anyone to modify the software or derive other software from it, and to redistribute the modified software under the same terms.

There exist several hundred or perhaps thousands of open source 'projects', ranging from small utilities and device drivers to Sendmail (an e-mail transfer program that almost completely dominates its market) to WWW servers (Apache) and a full operating system -- Linux. These projects are driven forward by contributions of hundreds, sometimes thousands of developers, who work from around the world in a seemingly unorganized fashion, without direct pay or compensation for their contributions.

[1] This has not always been the case, and I will discuss this history later in the paper.

Certainly Linux and Apache have attracted the most attention, primarily because of their technical and competitive success. Apache dominates its market: more than 60% of web servers run Apache as of March 2000, and Apache continues to grow relative to proprietary alternatives.[2] Linux has developed into a robust and highly stable operating system used by perhaps 20 million people worldwide, with an annual growth rate of nearly 200%. IT professionals praise Linux for its technical elegance and flexibility. Although Linux is generally considered not yet sufficiently 'user-friendly' for the non-technically oriented user of personal computers, even Microsoft believes that Linux poses the most significant challenge to its current near-monopoly control of the operating system market. In the last several years, almost all the major IT corporations including HP, Sun, Motorola, and most strongly IBM have made major commitments to Linux.[3] For-profit companies that provide auxiliary services around Open Source software are flourishing -- Red Hat and VA Linux were two of the biggest IPOs of 1999.

Computer enthusiasts and software engineers value Linux primarily for its technical characteristics. Some have taken a more general view and thought of Open Source in broadly political or sociological terms, trying to understand the internal logic and external consequences of a geographically widespread community that is able to collaborate successfully and produce a superior piece of software without direct monetary incentives. In early writings and analyses, mostly done by computer 'hackers' who are part of one or another open source project (and are often 'true believers'), Open Source has been characterized variously as:

- A particular methodology for research and development
- The core of a new business model (free distribution of software means that new mechanism for compensation and profit need to be created)
- The social essence of a community, a defining nexus that binds together a group of people to create a common good.
- A new 'production structure' that is somehow unique or special to a 'knowledge economy' and will transcend or replace production structures of the industrial era.
- A political movement.

Adding up pieces of these characterizations results in some lavish claims about the significance of the open source phenomenon: 'the result is a very different sort of software business that is potentially more responsive to individual customers' needs. Open source is a more socially responsible business model, leading to greater equity between producers and consumers, capital and labor, rich and poor."[4] Certainly these kinds of claims are over-stated. But we don't yet know exactly how, why, and by how much. It is not so easy to dismiss what is a more fundamental social science puzzle, which emerges more clearly and more soberly in two parts from the following formulation:

Collaborative Open Source software projects such as Linux and Apache have demonstrated, empirically, that a *large, complex system of code* can be built, maintained, developed, and extended in a *non-proprietary setting* where *many developers* work in a highly *parallel, relatively unstructured way* and without *direct monetary compensation*.

---

[2] www.netcraft.com/survey

[3] See for example: Lee Gomes, Microsoft Acknowledges Growing Threat of Free Software for Popular Functions," Wall Street Journal 3 Nov 1998 p. B6; Scott Berinato, "Catering to the Linux Appetite," PC Week 7 June 1999, p. 103; Denis Caruso, "Netscape Decision Could Alter Software Industry," New York Times 2 Feb. 1998, p. John Markoff, "Sun Microsystems is Moving to "Open Source" Model, New York Times 8 Dec. 1998, and the "Halloween Memo" in which Microsoft targets Linux as a major competitor: www.linux.miningco.com/library/blhalloween.html.. This memo was written by Vinod Valloppillil in August 1998.

[4] Mitchell Stoltz, "A Software Revolution", unpublished ms. page 3.

Mark Smith and Peter Kollock have called Linux 'the impossible public good'.[5] Linux is non-rival and non-excludable. Anyone can download a copy of Linux, along with its source code, for free, which means it is truly non-excludable. And because it is a digital product that can be replicated infinitely at no cost, it is truly non-rival. For well known reasons public goods tend to be underprovided in social settings.[6] The situation with Linux ought to be at the worse end of the spectrum of public goods since it is subject additionally to 'collective provision' -- the production of this particular collective good depends on contributions from a large number of developers. Stark economic logic seems to undermine the microfoundations for Linux. Why would any particular person choose to contribute -- voluntarily -- to a public good that he or she can partake of unchecked as a free-rider? Since every individual can see that not only her own incentives but the incentives of other individuals are thus aligned, the system ought to unravel backwards so that no one makes substantial contributions.

Linux also is an 'impossibly' complex good. An operating system is a huge, highly complicated and intricate piece of code that controls the basic, critical functions of a computer. It is the platform or the foundation upon which applications -- word processors, spread sheets, databases and so on -- sit and run. To design a robust operating system and to implement that design in code is a gargantuan task. Testing, debugging, and maintenance are frequently even harder. Computer users will run an operating system in a nearly infinite number of settings, with functionally infinite permutations leading to infinite possible paths through the lines of code. Complex software is less like a book and more like a living organism that must continually adapt and adjust to the different environments and tasks that the world confronts it with.

There was a time when a single determined individual could write a reasonable operating system, but given the demands of computer applications at present that is no longer possible. The task needs to be divided, which immediately becomes a problem of coordination within a division of labor. The standard answer to this question has been to organize labor within a centralized, hierarchical structure -- i.e. a firm. An authority makes decisions about the division of labor and sets up systems that transfer needed information back and forth between the individuals or teams that are working on particular chunks of the project. The system manages complexity through formal organization and explicit decisional authority.[7] While transaction costs (particularly of moving information and tacit knowledge around) reduce the efficiency of hierarchical coordination in a complex task like software development, the job gets done and an operating system -- imperfect and buggy, but functional -- is produced.

Eric Raymond, computer hacker turned unofficial ethnographer of the Open Source movement, draws a 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. Linux seems to confound this hierarchical model. It looks, at least on first glance, much more like a 'great babbling bazaar of different agendas and approaches'.[8] Yet this bazaar has produced a highly complex and integrated operating system that develops 'from strength to strength at a speed barely imaginable to cathedral builders.'[9] Many computer programmers believe that Linux has evolved into code that is technically superior to what hierarchical organizations can produce. Although the quality of an operating

---

[5] Marc A. Smith and Peter Kollock, eds. Communities in Cyberspace. London: Routledge, 1999, p. 230.
[6] The classic statement is Mancur Olson, The Logic of Collective Action. Cambridge MA: Harvard University Press, 1965. An excellent broad overview of the problem is Russell Hardin, Collective Action. Baltimore: Johns Hopkins University Press, 1982.
[7] 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.
[8] 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.
[9] Raymond, "The Cathedral and the Bazaar," p. 30.

system is to some degree a subjective judgement (like 'good art'), their view gains support from the fact that Linux is taking market share away from other operating systems.

To summarize and set the problem, Linux poses three interesting questions for social scientists.

- **Motivation of Individuals**: The social microfoundations of Linux depend on individual behavior that is at first glance startling. Public goods theory predicts that non-rival and non-excludable goods ought to encourage free-riding. Particularly if the good is subject to collective provision, where many people must contribute together in order to get something of value, the system should unravel backwards towards 'underprovision'. Why, then, do highly talented programmers choose voluntarily to allocate some or a substantial portion of their time and mind-space (that are both limited and valuable resources) to a project for which they will not be compensated?
- **Coordination**: How and why do these individuals coordinate their contributions on a single 'focal point'? Authority within a firm and the price mechanism across firms are standard means to efficiently coordinate specialized knowledge in a complex division of labor -- but neither is operative in open source. Any individual can freely modify source code and redistribute modified versions to others. A simple analogy from ecology suggests what might happen over time as modifications accumulate along different branching chains. Speciation -- or what computer scientists call code-forking -- seems likely. In effect, the system evolves into incompatible versions, synergies in development are lost, and any particular developer has to choose one or another version as the basis for her future work. Indeed, this is precisely what happened to UNIX in the 1980s.[10]
- **Complexity**: 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.[11] He explained this with an argument that is now known as Brook's Law: as you raise the number of programmers on a project, work performed scales linearly (by a factor n). But complexity, communication costs, and vulnerability to bugs scales geometrically, by a factor of n squared. This is inherent in the logic of the division of labor -- n squared represents the scaling of the number of potential communications paths and interfaces between the bodies of code written by individual developers. How has Linux managed the implications of this 'law' among a geographically dispersed community that is not subject to hierarchical command and control?

This paper has four major sections. The next section is a brief history of open source or 'free' software that sets the historical context for developments in the 1990s. Section three describes the open source process and characterizes more fully the economic, technological, and social systems that together constitute this distinct mode of production. Section four explains the open source process, by answering the three questions I posed about individual motivations, coordination, and complexity. I construct a compound argument of microfoundations, economic logic, and social/political structure. Section four, the conclusion, summarizes and characterizes the argument, explores its implications for several relevant debates, and finally speculates a bit about the possibilities of extending the open source model beyond computer software.

[10] The next section explores this history in more detail.

[11] Frederick P. Brooks, The Mythical Man-Month: Essays on Software Engineering. Reading Ma: Addison Wesley, 1975.

II. A Brief History of Open Source

The concept of 'free' software is not new. In the 1960s and 1970s, mainframe computers sitting in university computer science departments (particularly the Artificial Intelligence Lab at MIT and UC Berkeley) and in corporate research facilities (particularly Bell Labs and Xerox PARC) were thought of and treated as tools for research. The idea of distributing source code freely was seen as a natural offshoot of standard research practice; indeed it was mostly taken for granted.

This was a cultural frame with a both a pragmatic and an economic foundation. The pragmatic piece was a major push towards compatibility among different computer platforms. MIT had been using an operating system called ITS -- incompatible time sharing system -- which was an icon for the much broader problem that operating systems typically had to be reengineered for different hardware. As computer technology spread, the burdens of incompatibility clashed with the scientific ethic of sharing and cumulation of knowledge, as well as the simple practical problem of having to re-write extensive amounts of code for different machines. Bell Labs led the way forward by focusing effort on development of an operating system (UNIX) and an associated language for developing applications (C) that could run on multiple platforms.[12]

Under the terms of its regulated monopoly, AT+T could not engage in commercial computing activities and thus could not sell Unix for profit. It seemed almost natural to give away the source code to universities and others who the Bell Labs engineers believed could help them improve the software.[13]

The economics of mainframe computing reinforced the cultural frame. Mainframe software, typically under copyright, nonetheless was distributed for free in most cases along with the source code. Computer operators running software at any number of different sites would discover (and sometimes fix) bugs, innovate and modify the source code, and send these modifications back to the original distributor of the software, who would then include improvements in future releases of the software to others.

Concrete incentives supported this very casual and informal treatment of copyright. The behavior made sense to the owner of the copyright, since software was seen at this time not as a revenue generator itself but principally as a hook to encourage people to buy hardware. Give away better software, and you can sell (or in the case of IBM, mostly lease) more computers. It also made sense to the individual innovator to freely give ideas back to the software owner. If these innovations were incorporated into future software releases, the individual operator would not have to bother reintegrating improvements into each software update.

The logic of free software began to break down in the late 1960s. In 1969 the US Department of Justice filed a massive antitrust suit against IBM. IBM responded in a pro-active way. To pre-empt charges that the company was unfairly leveraging its very strong market position in hardware, IBM decided to unbundle its 'solutions' and begin charging separately for software.[14]

---

[12] Ken Thompson is usually given credit for being the 'inventor' of Unix and Dennis Ritchie is given credit for C. Both were employees of Bell Labs.

[13] The UC Berkeley group was particularly influential. Bill Joy, who would go on to found Sun Microsystems, headed the first Berkeley Software Distribution (BSD) project of Unix in 1978.

[14] This is the case that would drag on for 13 years, before finally being dismissed by the new Reagan administration in 1981. See Richard Thomas DeLamarter, Big Blue: IBM's Use and Abuse of Power. New York: Dodd, Mead, 1986.

This represents in a real sense the almost inadvertent birth of the modern commercial software industry. Microsoft (founded in July 1975) at least in it early years was the exemplar of this trend: a company that for all intents and purposes simply wrote and sold software. The arrival of the personal computer (PC) in the early 1980s and its rapid widespread distribution onto desktops in the business world reinforced this trend. Software that at one time had been traded freely among developers, was now an extraordinarily valuable and lucrative product.

AT+T was not blind to these developments, and the company started in the early 1980s to enforce more severely intellectual property rights related to UNIX. When the Department of Justice broke up AT+T in 1984, it was no longer legally restricted to being a telephone company. AT+T then decided, almost naturally, to try to make money by selling licenses to Unix. What had been free, was now becoming proprietary.

The Free Software Foundation

Steven Levy's book Hackers gives a compelling account of the impact these changes had on the programmer community, particularly at MIT. Many of the best programmers were hired away into lucrative positions in spin-off software firms. MIT began to demand that its employees sign non-disclosure agreements. 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 non-disclosure 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 easily to modify software for his particular needs, Stallman in 1984 founded a project to revive the 'hacker ethic' by creating a complete set of 'free software' utilities and programming tools.[15] Stallman founded the Free Software Foundation (FSF) to develop and distribute software under what he called the General Public Licence (GPL), also known in a clever word-play as "copyleft".

The central idea of GPL is 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 Free Software Foundation says: "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*."[16]

Stallman and the Free Software Foundation created some of the most widely used pieces of Unix software, including the Emacs text editor, 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 to

[15] 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.

[16] Free Software Foundation, "GNU General Public License, v. 2.0" 1991. www.gnu.org/copyleft/gpl.html. Emphasis added. There are several different modifications to these specific provisions, but the general principle is clear.

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 was because of the moral fervor underlying Stallman's approach -- not all programmers found his strident libertarian attitudes to be 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 also a deeper problem in the all-encompassing nature of the GPL. 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 to problems.

The Birth of Linux

Open source software depends heavily on communications tools, since it is users who modify, innovate, and evolve the code and these users can be located anywhere. The ARPAnet provided a tolerable platform for this kind of communication, but with the rapid spreading of the much higher capacity and much more broadly distributed Internet in the early 1990s there was a burst of energy in open source activity. Linux began during this period, and from shockingly modest roots.

Linus Torvalds, a 21 year old computer science student at University of Helsinki, strongly preferred the technical approach of UNIX style operating systems over the DOS system commercialized by Microsoft.[17] But Torvalds did not like waiting on long lines for access to a 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 complicated for the PC's of the time.

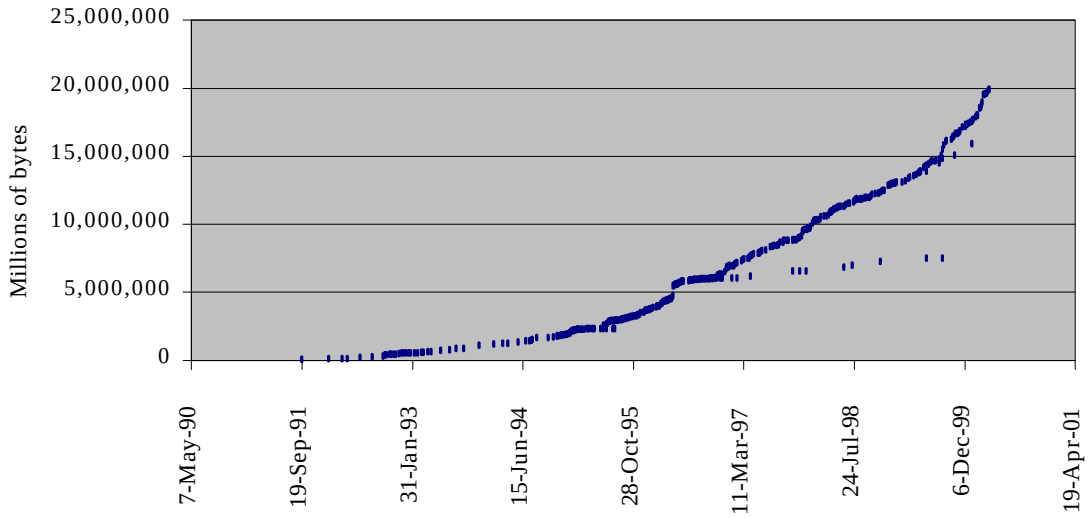
In late 1990 Torvalds came across Minix, a simplified Unix clone that was being used for teaching purposes at Vrije University in Amsterdam. Minix ran on PC's, and the source code was available. Torvalds installed this system on his IBM AT, a machine with a 386 processor and 4 MB of memory, and went to work building the kernel of a Unix-like operating system with Minix as the scaffolding. In autumn 1991, Torvalds let go off the Minix scaffold and released the source code for the kernel of his new operating system, which he called Linux, onto an Internet newsgroup, along with 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.'[18]

[17] "'Task - switching" is one major difference between the two systems, that was of interest to Torvalds. Unix allows the computer to switch between multiple processes running simultaneously.
[18] Linus Torvalds, "Linux History", 1999. www.li.org/li/linuxhistory.html

The response was extraordinary (and according to Torvalds, mostly unexpected). By the end of the year nearly 100 people worldwide had joined the Linux newsgroup; many of these people were active developers who contributed bug fixes, code improvements, and new features. Through 1992 and 1993 the community of developers grew at a gradual pace -- even as it became generally accepted wisdom within the broader software community that the era of Unix-based operating systems was coming to an end in the wake of Microsoft's increasingly dominant position.[19] In 1994, Torvalds released the first official Linux, version 1.0. The pace of development accelerated (as I explain later) with updates to the system being released on a weekly, or sometimes even a daily basis. The most recent Linux kernel contains more than three million lines of code. Figure 1 illustrates the dramatic growth of the program:

Linux Kernel Size (Compressed Code)



The energy behind Linux grew in the mid 1990s almost in parallel to Microsoft's increasing market share in operating systems, application programs, and internet software. Microsoft's principal business asset was its private ownership of the source code for Windows. Bill Gates leveraged that asset (perhaps too aggressively, according to the US Justice Department) in a manner that many computer programmers in particular found galling. But what was most troubling to the programmer community, including many of those within Microsoft itself, was the result of this monopolistic strategy in terms of the poor technical quality of the resulting software. Windows and many of the Microsoft applications that ran on top of it ballooned into huge, inefficient, and unreliable mountains of code. In what seemed a continuous vindication of Brooks' law, Microsoft grew into a fabulously wealthy corporation that employed an army of top-notch programmers whose best efforts produced buggy, awkward, inelegant software -- behind schedule and over-budget.

[19] Raymond, "A Brief History of Hackerdom", in The Cathedral and the Bazaar, p. 22-23

This was a structural problem. Regardless of how powerful it grew, Microsoft could not employ enough testers, designers, and developers to understand how its software would actually be used under an almost infinite number of conditions in the 'real world'. Because the source code was secret, and because the business culture that Gates promoted made his company a nemesis for many technically-sophisticated computer users, Microsoft actually blocked the kind of constant interaction between users able to repair and improve a piece of software that Linux and other open source projects were able to muster. Almost inevitably in this context, the software deteriorated in quality as the marginal returns from throwing more money and more developers at the problem declined.[20] And Microsoft's increasingly aggressive business practices further alienated many of the software engineers and developers who had a different vision of what software was supposed to be.

The Open Source Initiative

Much of this energy rebounded to the benefit of Linux and other open source projects. But there was still a problem with the all-encompassing nature of the GPL as well as the lack of flexibility inherent in the Free Software Foundation's unforgiving ideological stance. This began to change in the mid-1990s as alternative structures for 'free' software developed within the community.

The "Open Source Definition", the defining document of the Open Source Initiative, has its origins in Debian -- an organization set up in the mid 1990s to disseminate an early Linux system known as the Debian GNU/Linux Distribution. Debian was to be built entirely of 'free software'. But because there were a variety of licenses other than the GPL which professed to be free but had their own singular characteristics, there were problems defining precisely the intellectual property rights and licenses around different parts of the software package. The leader of Debian, Bruce Perens, in 1997 proposed a Debian Social Contract and Debian Free Software Guidelines to address the confusion. After Perens solicited widespread comment and criticism on these documents and incorporated feedback (in the spirit of free software), a new and more flexible licensing procedure emerged which allowed the bundling of free software with proprietary code.

What became known as the "Open Source Definition" took this several important steps further.[21] While the GPL *requires* any redistribution of GPL'ed software to be released under the same terms (and thus insures that no GPL code can ever be used as part of a proprietary software project), the Open Source Definition simply *allows* redistribution under the same terms, but does not require it. Some licenses that fall under the Open Source Definition (for example, Netscape's Mozilla Public License or MPL) entitle a programmer to modify the software and release the modified version under new terms, that include making it proprietary.

This critical change removed the viral impact of the GPL. It also codified a key tenet of the evolving open source software culture: pragmatism in the development of technically sophisticated software would trump ideology. Not everyone shared this view or saw it as a progressive change -- least of all Richard Stallman, who continued to stress the theme of 'freedom, community, and principles' as more important

[20] Consistent with Brook's law. Microsoft was 'saved' to a certain extent by extraordinary expansion of the power and capacity of personal computers -- which created a relatively forgiving environment for poorly engineered software. Also, the vast growth of personal computers engaged a technologically unsophisticated base of users who really didn't know that they should expect better, since they were unaware of the standards of reliability, efficiency, and functionality that had previously been established in mainframe computer software.

[21] See http://opensource.org/osd.html.

than simply good software.[22] But the philosophical core of the Open Source Initiative was in a very different place. Eric Raymond put it this way:

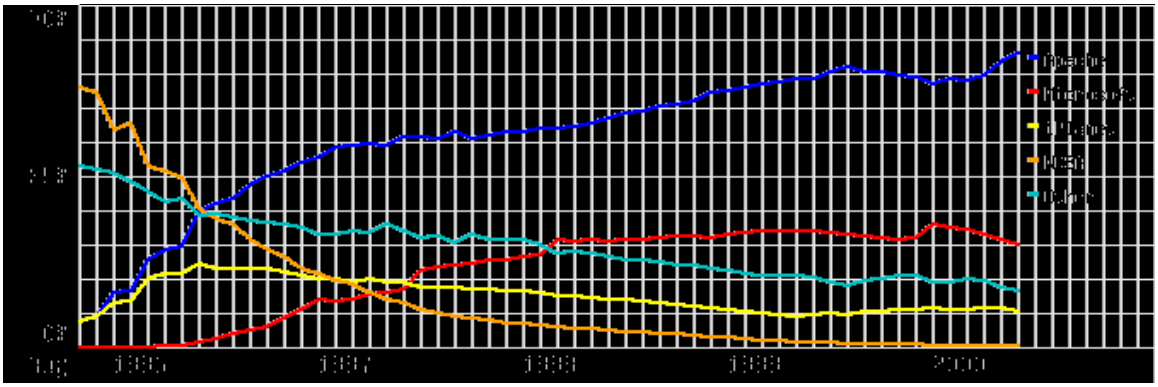
> "It seemed clear to us in retrospect that the term 'free software' had done our movement tremendous damage over the years. Part of this stemmed from the well-known 'free-speech/free-beer' ambiguity. Most of it came from something worse -- the strong association of the term 'free software' with hostility to intellectual property rights, communism, and other ideas hardly likely to endear themselves to an MIS manager'.[23]

The Open Source Initiative promoted a narrative of economic competitiveness and aimed it directly at the mainstream commercial and corporate world.[24] What mattered about open source software was simply that it was technically excellent. Open Source was about high reliability, low cost, and better features. Equally important, a corporation using open source software could avoid becoming locked in to a controlling monopolist like Microsoft. It would gain autonomy through control of the guts of the information systems that were the increasingly the core asset of almost any business. OSI began to promote this message in a self-consciously aggressive fashion. Instead of preaching to the already converted or easily converted information systems managers, OSI aimed its message at the CEOs of Fortune 500 companies and mainstream publications like The Wall Street Journal and Forbes.

And the corporate world began to respond directly. In January 1998 Netscape announced (as part of a strategic effort to reverse its declining competitive position vis-à-vis Microsoft) that it would release the source code for its web browser as open source code. By summer 1998 both Oracle and Informix, two of the largest and most influential independent software vendors for corporate applications and databases, announced that they would port their applications to Linux.[25] Over the next several months, other first tier independent software vendors (including SAP and Sybase) made similar announcements. In the first half of 1999 IBM began focusing on Linux as the operating system for its servers.[26] IBM also became a major supporter of "Beowulf" supercomputing, a 'cluster' approach to high intensity business applications that distributes a task over a network of computers running Linux.[27] Major hardware vendors (Compaq, Dell, Hewlett Packard, Silicon Graphics) and the venerable Intel all have made major commitments to Linux. Two for-profit companies that provide auxiliary services and support for Linux -- Red Hat Software and VA Linux -- were two of the largest IPO stock offerings of 1999. Meanwhile, Apache (the most prominent open source web server software) continued to increase its lead in the web server market just as the web itself was exploding in popularity:

---

[22] See Richard Stallman, "The GNU Operating System and the Free Software Movement," Open Sources, p. 53-70. Quote from p. 70.
[23] Raymond, "The Revenge of the Hackers," Open Sources p. 212.
[24] "We think the economic self-interest arguments for Open Source are strong enough that nobody needs to go on any moral crusades about it". www.opensource.org
[25] To 'port' means to write a version of the software that runs on a particular operating system.
[26] See Scott Berinato, "Catering to the Linux Appetite, PC Week 7 June 1999 p. 103. See also "Linux Graduates to Mainframes," Industry Standard 17 May 2000.
[27] "IBM to Join in Linux Supercomputing Effort," Cnet, 21 March 2000.



Source: www.netcraft.com/survey

Microsoft itself began to see open source development strategies in general, and Linux in particular, as the primary credible threat to the dominance of Windows and particularly to the more technically sophisticated Windows NT.[28] A high-level internal Microsoft memorandum of summer 1998, that was leaked and became known as "the Halloween Memo", portrayed Open Source Software as a direct short term threat to Microsoft's revenues and to its quasi-monopolistic position in some markets. More important for Microsoft, Open Source represented a long term strategic threat because "the intrinsic parallelism and free idea exchange in OSS [Open Source Software] has benefits that are not replicable with our current licensing model." The Halloween Memo explicitly states that:

- OSS is long term credible…. (because) the real key to Linux isn't the static version of the product but the process around it. This process lends credibility and an air of future-safeness to customer Linux investments.
- Linux has been deployed in mission critical, commercial environments with an excellent pool of public testimonials…. Recent case studies provide very dramatic evidence that commercial quality can be achieved/exceeded by OSS projects.
- The Internet provides an ideal, high visibility showcase for the OSS world.
- The ability of the OSS process to collect and harness the collective IQ of thousands of individuals across the Internet is simply amazing. More importantly, OSS evangelization scales with the size of the Internet much faster than [Microsoft's] evangelization efforts appear to scale.

Certainly, we now know that Linux 'works' -- in a technical and economic sense, as well as a sociological sense. The next section of this paper describes briefly how Linux works, or more precisely the functional rules and principles that sustain the Open Source community and the production structure that yields software.

[28] See Lee Gomes, "Microsoft Acknowledges Growing Threat of Free Software for Popular Functions," Wall Street Journal 3 November 1998 p. B6; and the "Halloween Memo", at linux.miningco.com/library/blhalloween.html.

III. How does Open Source Work?

Eric Raymond describes OS development as a '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."[29] This contrasts sharply with a 'cathedral' model -- the exemplar of a hierarchically-organized, authoritatively-ordered division of labor.

The images are important. They point to the essence of Open Source, which is to create a *process*. The software *product* itself is valuable but is not the key to understanding open source. The process is what matters most.

The Open Source process revolves around a core code base. The source code is available and users can modify it freely. From here, the process varies for different OS projects. For example, this is where the Berkeley Software Distribution (BSD) stops. In BSD, typically, a relatively small and closed 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 the public users, and there is no regularized process for doing that. BSD is open source because the source code is free, but it is not a collaborative project or a 'bazaar' in the same sense as is Linux.

The key element of the Linux process is that the public or general user base can and does propose 'check-ins', modifications, bug fixes, new features, and so on. There is no meaningful distinction between users and developers. The process encourages extensions and improvements to Linux by a potentially huge number of decentralized developers. There are low barriers to entry to the debugging and development process. This is true in part because of a common debugging methodology derived from the GNU tools (created under the auspices of Richard Stallman's Free Software Foundation), 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 come across in daily use; while others make debugging and developing Linux a hobby or vocation. In either case, users typically submit the patches, modifications, or new features back to the core code base. Some of these changes are incorporated into the general distribution of the software according to a set of informal decision rules that I describe below.

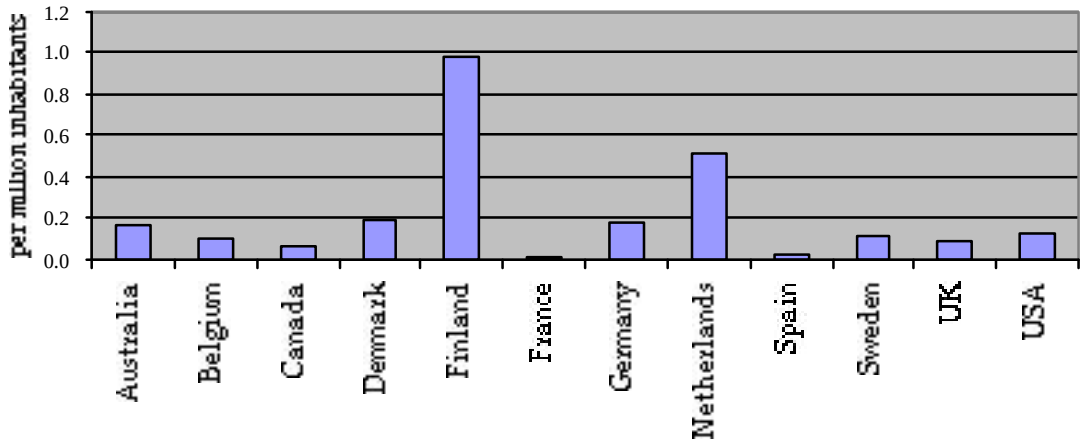
To characterize the Linux process, I pose four questions: Who are the people? What do they do? How do they collaborate? And how do they resolve conflicts?

Who are the People?

The community of developers that contribute to Linux is geographically far flung, extremely large and notably international: the credits file for the March 2000 release names contributors from at least 31 different countries.

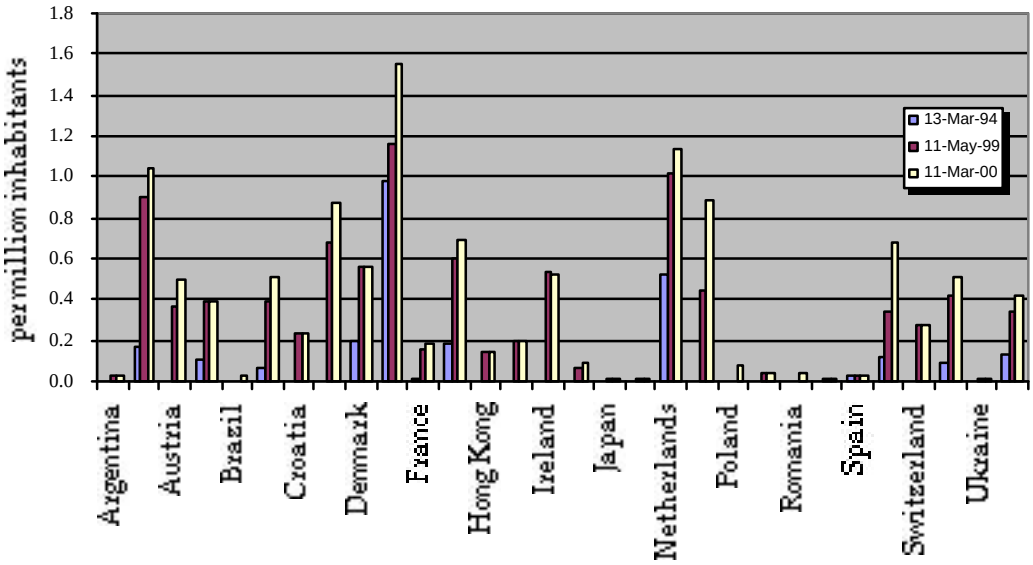
[29] Raymond, "The Cathedral and The Bazaar,", p. 30.

People in the first credits file (1994)



The international aspect of Linux development has certainly not decreased over time and with success:

People in the credits file



As in most communities, there exist a large number of moderately committed individuals who contribute relatively modest amounts of work and participate irregularly; as well as a smaller but much more highly committed group that forms an informal core. A July 2000 survey of the Open Source community identified approximately 12,000 developers working on open source projects. Although the survey recognizes difficulties with measurement, it reports that the top 10% of the developers are credited with

about 72% of the code -- loosely parallel to the apocryphal "80/20 rule" (where 80% of the work is done by 20% of the people).[30] Linux user/developers come from all walks of life: hobbyists, people who use Linux or related software tools in their work, committed 'hackers'; some with full time jobs, and some without. Without delving deeply into the motivations of these individuals (I discuss this in the next section of the paper), there is a shared general goal that binds them, however loosely, together. It is simply this: to create (and evolve) a Unix - style operating system that is reliable, stable, technically elegant, pragmatically useful, and responsive to changing requirements.

What do they do?

Linus Torvalds created the core code for the Linux kernel and released it onto an Internet newsgroup in 1991. Other users downloaded the software. They used it, played with it, tested it in various settings, and talked about how they might extend and refine it. Flaws appeared: bugs, security holes, desired features. Users (who might be motivated simply by an interest in the subject, a desire to participate, or simply a need to solve an immediate problem for their own purposes) sent in reports of the problem; or they might propose a fix and send the software patch on to Torvalds. Often they wrote code to implement a new feature. Over the next ten years, the process iterated and scaled up to a degree that just about everyone, including its ardent proponents, found startling.

The logic behind this process is both functional and behavioral. It came about largely through a game of trial and error -- but a game that was played out by people who had a deep and fine-grained if implicit understanding of the texture of the community they were trying to mobilize. It did not emerge directly or deductively from a theory of how to sustain large scale, decentralized collaboration.

Over time, observers (particularly Eric Raymond) analyzed the emergent process and tried to characterize (inductively for the most part) the key features that made it successful. Drawing largely on Raymond's analysis and a set of interviews, I propose seven key principles for what people do in Open Source:

Pick Important Problems and Make Them Interesting

Open source user-developers tend to work on projects that they judge to be important, significant additions to software. 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 an newly elegant way. There seems to be an important and somewhat delicate balance around how much and what kind of work is done up-front by the project leader(s). User-developers look for signals that any particular project will actually generate a significant product, not turn out to be an evolutionary dead end, but also contain lots of interesting challenges along the way.

"Scratch an itch"

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 it is they want to work on. This means that they will tend to focus on an immediate and tangible problem (the 'itch') -- a problem that they themselves want to solve. The Cisco enterprise printing system (an older open source style project) evolved directly out of an

[30] 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.

immediate problem -- system administrators at Cisco were spending an inordinate amount of time (in some cases half their time) working on printing problems.[31] Torvalds (and others as well) sometimes put out a form of request, as in "isn't there somebody out there would wants to work on 'X' or try to fix 'y' problem?) The underlying notion is that in a large community, someone will find any particular problem of this sort to be an itch they actually do want to scratch.

Reuse whatever you can

Open source user-developers are always searching for efficiencies: put simply, because they are not compensated (at least not monetarily) for their work, there is a strong incentive never to reinvent the wheel. An important point is that there is less pressure on them to do so. This is simply because under the open source rubric they can know with certainty that they will always have access to the source code and thus do not need to re-create any tools or modules that are already available in OS.

Use a parallel process to solve problems

If there is an important problem in the project, 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. It is then possible for Linux to incorporate the best solution and refine it 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 be measured in a realistic fashion. Evolution is messy, and this process recapitulates much of what happens in an evolutionary setting. What is clear is that the stark alternative -- a nearly omniscient authority that can predict what route is the most promising to take toward a solution, without actually travelling some distance down at least some of those routes -- is not a realistic counterfactual for complex systems.

Leverage the sheer numbers

The Linux process relies on a kind of law of large numbers to generate and identify software bugs, and then to fix them. Software testing is a messy process. Even a moderately complex program has a functionally infinite number of paths through the code. Only some tiny proportion of these paths will be generated by any particular user or testing program. As Paul Vixie puts it, "the essence of field testing is *lack* of rigor".[32] The key is to generate patterns of use -- the real world experiences of real users -- that are inherently unpredictable by developers. In the Linux process, a huge group of users constitutes what is essentially an on-going huge group of beta testers.

Eric Raymond says "given enough eyeballs, all bugs are shallow."[33] Implied in this often-quoted aphorism is a prior 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 a parallel debugging process.

[31] http://ceps.sourceforge.net/index.shtml
[32] Paul Vixie, "Software Engineering," in Open Sources, p. 98. Emphasis added.
[33] Raymond, "The Cathedral and the Bazaar," p. 41.

Document 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 from code) what functions a particular piece of code plays in the larger scheme of things. Documentation is a time consuming and sometimes boring process. But it is considered essential in any scientific research enterprise, in part because replicability is a key criterion.

"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 do open source developers that users are often the best testers, so the principle of release early release often makes sense in that setting as well. The countervailing force in the commercial setting are expectations: customers who are paying a great deal of money for software may not like buggy beta releases, and may like even less the process of updating or installing new releases on a frequent basis.

Open source user-developers have a very different set of expectations. In this setting, bugs are more an opportunity and less a nuisance. Working with new releases is more an experiment and less a burden.[34] The result is that open source projects typically have a feedback and update cycle that is an order of magnitude faster than commercial projects. In the early days of Linux, there were often new releases of the kernel on a weekly basis -- and sometimes even daily!

How do they collaborate?

Networking has long been an essential facilitator for open source development. Before the Internet (and even before there was extensive use of other computer networking protocols that were not always widely compatible) prototypical open source communities grew up in spaces that were bounded by geography (and sometimes by corporate borders) -- such as Bell Labs, the MIT AI Lab, and UC Berkeley. Extensive sharing across these boundaries was frequently difficult, sometimes expensive, and slow. The Internet was the key facilitating innovation, by wiping away networking incompatibilities and the significance of geography (at least for sharing code).

The widespread diffusion of Internet lowers transaction costs dramatically in the sense that anyone with an IP connection can instantaneously access a variety of tools for participation in open source development -- email lists, newsgroups, web pages. Equally important, the Internet scales in a way that physical space does not. Put 25 people in a room, and the communication slows down -- whereas an email list can communicate with 25 people just as quickly as it communicates with 10 -- or 250.

It is important to be clear about the limits of the argument on this point. Clearly the Internet does not create the open source process. It is an enabler -- probably a necessary but certainly not anything approaching a sufficient condition. And there are new issues that arise. For example, reducing transaction costs and lowering barriers to entry may at the same time exacerbate coordination problems among large numbers of players. And there is no compelling logic or evidence to suggest that the Internet

[34] 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.

in and of itself in any sense 'solves' Brooks' Law. Other mechanisms and aspects of the open source process (that I consider in the final section of the paper) need to be invoked to explain how these problems are solved.

How do they resolve conflicts?

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.[35]

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.[36] Similar issues of course arise when software development is organized in a corporate setting. Standard theories of the firm explain various ways in which potential conflicts are settled or at least managed by formal authoritative organizations.[37]

In open source, much of the important conflict management takes place through behavioral patterns and norms that I discuss in the next section of the paper. The 'formal' organizational structure, which deals mostly with Raymond's first problem (decision making about what code to include), is generally sparse. It is also different for different projects. Apache developers rely on a decision making committee that is constituted according to formal rules, a de facto constitution. Perl 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.

Linux, in its earliest days, was run unilaterally by Linus Torvalds. Torvald's decisions were essentially authoritative. As the program and the community of developers grew, Torvalds delegated responsibility for sub-systems and components to other developers, who are known as 'lieutenants'. Some of the lieutenants onward-delegate to 'area' owners who have smaller regions of responsibility. The organic result is what 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.

Torvald's authority rests on a complicated mix. History is a part of this -- 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. It is notably limited in the sense that Torvalds goes to great lengths to document and justify his decisions about controversial matters. He admits when he is wrong. It is almost a kind of charisma that has to be continuously recreated through consistent patterns of behavior. Linux developers will also say that Torvald's authority rests on its 'evolutionary success'. The fact is, the

[35] 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 p. 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."
[36] Eric Raymond, "Homesteading the Noosphere," in The Cathedral and the Bazaar pp. 79 - 137.
[37] 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.

'system' that has grown up under his leadership worked to produce a first-class outcome and this in itself is a strong incentive not to fix what is clearly not broken.

These are all functional characterizations. Together they describe reasonably well important interactions among the (remarkably self-aware) developers who create open source software. Clearly they do not constitute by themselves a robust explanation. It may be that the open source process has happened onto some kind of a dynamically stable equilibrium that constitutes an alternative structure of organizing production. A deeper explanation needs to elucidate more precisely the basis of that equilibrium, and to illustrate either why it is not challenged or why challenges do not disrupt it. The next section of the paper does this.

IV. Explaining Open Source

If altruism were the primary driving force behind open source software, no one would care very much about who was credited for particular contributions or who was able to license what code under what conditions.[38] If Linux were simply the collective creation of 'like-minded' individuals who cooperate easily because they are bound together by semi-religious beliefs, there would be little disagreement in the process and little need for conflict resolution among developers.[39] Beyond these overly simplistic and naïve tales lies a more subtle story of social and political economy, that has important implications for arguments about the logic of production in the 'new' economy.

A good explanatory story must be social because Linux is a collective good: it is a body of code built, maintained, developed, and extended in a non-proprietary setting where many developers work in a highly parallel way. It must be political because there exist both formal and informal structures and organizations, that function to allocate scarce resources and manage conflicts as well as promote certain practices and values. And self-evidently it must be economic, because at the core of the Linux process are individuals who engage in some kind of cost-benefit analyses and are maximizing or at least satisficing along some kind of utility function.

No single piece of this story -- even if 'correct' -- would by itself explain the outcome. Another way to put this point is simply to argue that Linux rests on a set of microfoundations -- the motivations of individual humans that choose freely to contribute -- as well as on macrofoundations -- social and political structures that channel these contributions to a collective end.

In addition to elucidating the logic of both, I will argue in this section that the two are autonomous from each other. The macro-logic of open source does not reduce to an aggregation of the motivations of the individuals who participate. More simply, this is not a self-organizing system. And the microfoundations that make up individual utility functions do not follow directly from a social and political structure that is somehow exogenous. I construct my explanation for the Linux process in three steps:

[38] Popular media often portray the open source community in this light, but fail to account for the fact that many 'beneficiaries' of this altruism (apart from the developers themselves) are major corporations that use Linux software.
[39] For example see "After the Microsoft Verdict," The Economist April 8 2000.

- What are the core microfoundations, the motivations that make up the guts of cost-benefit analyses and utility functions?
- What is the economic logic situating the collective good at play?
- What are the social and political structures that, in the context of individual motivations and the economic logic of software creation, drive the two essential dynamics that lead to successful open source development: maintaining coordination on the focal point, and managing complexity.

Previous analytic efforts have contributed important insights that make up pieces of the story. I start from the presumption that analysts should pay close attention to what the people in the community under analysis think and say to each other about what they are doing. The community of developers is, in fact, quite self-reflective and individuals within that community (Eric Raymond is of course the best known but by no means the only author) have made significant arguments about the process. Stallman has emphasized a normative argument about the nature of software as scientific knowledge, not a proprietary product -- and thus something to be shared and distributed "like sharing of recipes among cooks".[40] But exhorting his colleagues to shift their normative frames -- even if it is done in the context of a compelling conception that the world would be a better place if they did so -- does not explain why so few people in fact think of or treat software in this way.

Other developers observe that hackers are motivationally very much like 'artists', in the sense that they seek fun, challenge, and beauty in their 'work'. This is important, as I will argue later. But artists typically don't give away their art for free, and they do not show everyone how they made the art. Nor are artists typically thought of as successful collaborators in large, multifaceted projects without authoritative direction.[41]

Some developers emphasize the importance of a strongly shared commitment to oppose Microsoft. Clearly Microsoft evokes strong feelings. Many in the Linux community view the company as the apotheosis of precisely what is wrong with software development both socially (ruthlessly competing at any cost to maximize profit, as opposed to fostering real innovation) and technically (producing clumsy, complicated, bulky software that locks in customers without meeting their real needs as computer users). But it is not at all clear that open source development is the best way to fight Microsoft, if that were developers' primary motivation. And hatred of Microsoft (even if shared) by itself does not explain the extensive coordination that has produced Linux, rather than a diffuse spray of anti-Microsoft efforts.

Developers also tend to emphasize the importance of ego, in a way that is related to but goes beyond the argument about artistic motivations. In trying to create a legacy as a great programmer, many developers believe deeply that 'scientific' success will outstrip and outlive financial success. The *truly* important figures of the last phase of the industrial revolution were not the Rockefellers and the Fords, but Einstein.

A high intensity race for what is sometimes called 'ego-boo' (short for ego boosting) certainly explains some of the energy that developers devote to open source work. But it does not explain coordination well

[40] See Richard Stallman, "The GNU Operating System and the Free Software Movement," in Open Sources pp. 53-71. Lawrence Lessig also takes a primarily normative stance toward open source -- he is less interested in explaining it per se, than in the values at stake and the possibility (which he extols) that open source could be a major hindrance to government and corporate regulation of the internet via code. See "Open Code and Open Societies: Values of Internet Governance," Chicago-Kent Law Review 74. 1999. Pp. 101-116.

[41] A good example is the Orpheus symphony, which works without a conductor. Jutta Allmendinger estimates that Orpheus spends three to four times as many hours rehearsing per hour of performance time, as does an orchestra with an authoritative conductor.

at all. If ego is the primary driver, each person should be trying to promote him or herself -- which ought to make coordination among large numbers quite difficult. And ego is a very tricky thing to manage, even in a centralized setting (and much more so in a decentralized setting). Raymond's vision of how ego works has a ratchet-up flavor to it: developers' egos get boosted when they write a good piece of code and are recognized for doing so. But this leaves aside the question of what happens when they work for weeks on a patch that is then rejected, or when another developer writes code that supercedes and replaces their previous contribution. Don't egos get damaged, and if they do, why don't the owners of those egos walk away from -- or even worse, try to undermine -- the project?

In the last two years or so, open source has begun to attract more attention among social scientists, lawyers, and other analysts outside the community of software developers itself. The main arguments generated in this emerging literature tend, as I suggested, to focus strongly either on microfoundations (by problematizing and then trying to make sense of programmers' motivations) or macrofoundations (by privileging the particular economic logic of collective action in software development).

Microfoundations

Lerner and Tirole in "The Simple Economics of Open Source" make what is probably the most forceful microfoundational argument.[42] They portray an individual programmer engaged in straightforward cost-benefit analysis. The immediate benefits are simply private benefits: creating a fix for the specific problem that the programmer faces ('scratching the itch') as well as possibly a monetary benefit in some cases (for example if the programmer is working for a commercial firm where open source software is used). The primary cost is the opportunity cost of the time and effort that the programmer spends contributing to an open source project.

Open source modifies the cost-benefit analysis in two ways. First, the 'alumni effect' tends to lower the cost for working on open source relative to proprietary software.[43] Since Unix syntax and tools are a standard part of most programmers' educational training, the costs of learning to program in open source are relatively low. Second, there are 'delayed' benefits that together make up a strong 'signaling incentive'.[44] These benefits accrue to the programmer's career in ways that are ultimately transformed -- or *can be transformed* -- into money. The story goes like this: ego gratification is important because it stems from peer recognition. Peer recognition is important because it creates a reputation. A reputation as a great programmer is monetizable -- in the form of job offers, privileged access to venture capital, etc.

The key point in this story is the signaling argument. 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. To know what is really good code and thus to assess the talent of a particular programmer takes a reasonable investment of time. The best programmers have a clear incentive to reduce the energy that it takes for others to see and understand just how good they are. Hence the importance of signaling. The programmer participates in an open source project as a demonstrative act to show the quality of her work.

Reputation within a well-informed, committed, and self-critical community is one proxy measure for that quality. Lerner and Tirole argue that the signaling incentive will be stronger when the performance is visible to the audience; when effort expended has a high impact on performance; and when performance

[42] 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' that invites further research.
[43] Lerner and Tirole, p. 11.
[44] Lerner and Tirole p. 15.

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 all 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 distinguish between merely good and excellent solutions to problems. As Lerner and Tirole note, this argument seems consistent with the observation that open source has flourished (at least first) in more technical settings like operating systems and not in end-user applications.

Macro-level organization plays less a principal role in Lerner and Tirole's account; it seems to follow for the most part from individual incentives. They certainly acknowledge that open source projects appear to require credible leadership and organization to provide a vision, divide up a project into modules, attract other programmers, and keep the project together. But they retreat into what is essentially a Weberian notion of charisma to outline some of the attributes that might characterize a successful leader. This is not wrong per se; Torvalds as an example certainly has some of these characteristics. But charisma comes in many different forms, and it can work in many different ways. In any case it captures only part, and probably a small part, of the macro-logic that keeps an open source project together.

Macro-organization

Ko Kuwabara puts together a compelling synthetic account of some of the major arguments about macrofoundations of open source.[45] He uses a metaphor of complex adaptive systems and an evolutionary engine of change to describe the process. The account boils down to a series of causal steps, most of which were described by Raymond in his several articles. Programmers are motivated by a 'reputation game' similar to what Lerner and Tirole depict. But the key to understanding open source is the social structure within which that game is played out. Because online communities live in a situation of abundance not scarcity, they are apt to develop "gift cultures" where social status depends on what you give away, rather than what you control.[46] 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 property rights may constitute a sufficient framework for successful and productive collaboration. The central elements to this story are ownership customs or property rights, gift cultures, and self-organization.

Self-Organization

In general, the concept of self-organization by itself is a placeholder for an undetermined or underspecified mechanism. If organization arises 'organically' or endogenously out of the interactions among individuals, then the argument reduces to a claim about the driving forces behind those individuals' behavior. The same is true if Lockean property rights arise 'organically' in many different contexts where central authority is lacking or too weak to authoritatively allocate scarce and valuable

[45] Ko Kuwabara, "Linux: A Bazaar at the Edge of Chaos," First Monday March 2000.
[46] The 'gift culture' argument is taken principally from 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.

goods.[47] But we know from observation that not all groups of programmers 'self-organize' into open source communities -- Microsoft programmers certainly don't -- and in fact open source communities represent the exception rather than the rule. There is something more, something else in the social structure that needs to be uncovered.

Gift Cultures

The gift culture idea is one important hypothesis about that missing piece. Gift economies -- where social status depends more on what you give away than what you have -- are a reasonable adaptation to conditions of abundance. Raymond notes that gift economies are seen among aboriginal cultures living in mild climates and ecosystems with abundant food, as well as among the extremely wealthy in modern industrial societies.[48] And the culture of gift economies does share some notable characteristics with open source: gifts bind people together, they encourage diffuse reciprocity (often with generous norms such as 'give more than you receive'), and they support a concept of property that is more like 'stewardship' than it is like 'ownership' per se. Raymond suggests that the gift culture logic might work particularly well in software, since the value of the gift (in this case a complex technical artifact) cannot be easily measured except by other members of the software community, who are in a position to determine just how fine a gift it really is. "Accordingly, the success of a giver's bid for status is delicately dependent on the critical judgement of peers."[49]

The gift economy notion is important, and the culture of open source communities has at least some of these characteristics. But there is a key flaw in the argument that Raymond and others have missed. What, exactly, is the nature of abundance in this setting? Of course there is plenty of bandwidth, diskspace, and processing speed. The experience of the last decade (and there is no reason to believe that this will change in the immediate future) is that each of these things gets more abundant and less expensive over time, according to variations of Moore's Law, Metcalfe's Law, or some other statement of abundance.

But computing power is *not* the key survival necessity or value in this 'ecosystem'. In part because of its abundance and cheapness it is devalued. When anyone can have a supercomputer on her desk, there is little status connected to that 'property'. And the computer by itself does nothing. It cannot write its own software (at least not yet) or use that software to produce things of value. These are the things that add value in any economy -- gift or otherwise. *They depend on human mind space and the commitment of time by very smart people to a creative enterprise*.

The time and brainspace of smart, creative people are not abundant. They are scarce, possibly becoming more scarce as demand for their talents increase in proportion to the amount of abundant computing power available. Canada has plenty of trees, and anyone can put a stack of paper on their desk for a very small price. That does not translate directly into an abundance of great writing. And great authors do not typically give away the fruits of their hours of labor for free. That is precisely because the hours are scarce and costly, no matter how abundant and cheap is the paper. Nor is a reputation for greatness typically abundant, because only a certain number of people can really maintain a reputation for being 'great writers' at any given point in time.[50]

---

[47] Raymond, "Homesteading the Noosphere," p. 94.
[48] Raymond, "Homesteading the Noosphere," p. 99.
[49] Raymond, "Homesteading the Noosphere," p. 103.
[50] I say this because standards of 'greatness' are themselves endogenous to the quality of work that is produced in a particular population. If there is a normal distribution of quality, and the bell curve shifts to the right, what would

Property Rights

Finally, property rights or ownership customs must be a key part of the macrofoundations of open source. The question of who owns what in this setting is critical to determining the things that actually matter about ownership in the open source economy: who has the right to make decisions about which pieces of code are included in public distributions, who gets credited for what work, and who can make a legitimate choice, sanctioned by the community, to 'fork' the code and claim to be 'the' product.

Raymond observes that there are three typical ways in which someone (or some group) acquires ownership of an open source project. The first is simply to found the project -- as long as the founder is active, that person is the clear owner. The second is to have ownership passed on to you by the existing owner. There are customary subnorms in this process -- centering on the idea that the owner has not only the right but the responsibility to bequeath a project to a competent and respected successor when the original owner no longer is willing or able to perform his or her role in development, maintenance, and leadership. The third way to acquire ownership is to pick up a project that needs work, where the presumptive owner has disappeared, lost interest, or otherwise moved on. Again, there are subnorms -- it is customary for someone who wants to do this to make substantial public efforts to find the owner and wait a reasonable period of time before claiming new ownership. It is also generally recognized that ownership acquired in this third way is not fully legitimate until the new owner has made substantial improvements to the project and brought them out to the open source community.[51]

Raymond makes a powerful observation about these customs.[52] They bear a striking resemblance to Anglo-American common law about land tenure, that John Locke systematized in the late 1600s. Lockean property theory says that there are three ways to acquire ownership of land. You can homestead on a piece of frontier territory that has never had an owner. Or you can receive title from a previous owner. Last, if title is lost or abandoned, you can claim it through adverse possession by moving onto the land and improving it.

The resemblance is uncanny. And it may be that the *descriptive* economics behind it are quite simple: property customs like these can arise in settings where property is valuable, there is no authoritative allocation, and the expected returns from the resource exceed the expected cost of defending it. A satisfactory economic *explanation* is more difficult, even if a functional explanation were acceptable (which I believe it is not). We know that the first two of these conditions (valuable property, no authoritative allocation) will often hold for open source projects, but the third is problematic. What, indeed, are the expected returns from owning a project and how can we know, a priori, that they exceed the costs of defense? This is not a simple question, particularly when we see the extraordinary amount of time, energy, and emotion that breaks loose on email lists and newsgroups when the norms are threatened or breached. Successful collaboration is not in any sense foreshadowed by Lockean property customs -- after all, even with these norms in place and widely accepted the frontier was not a particularly peaceful place. And for collaborative engagement to provide collective goods, typically authoritative institutions grew on top of the common law regime. Open source software projects may have a set of Lockean-type norms, but these norms do not explain what is really notable about the success of Linux as a collaborative project.

---

have been thought 'excellent' in the past is now merely good. The tails of the distribution define excellence in any setting, and they remain small.

[51] Raymond, "Homesteading the Noosphere," p. 90-91.

[52] Raymond, "Homesteading the Noosphere," p. 94.

Three Steps to Explanation

Open source depends upon the motivations of individuals, the economic logic of a distinctive production process, and a set of social and political structures that maintain coordination and manage complexity. The existing literature is stronger on the first of these points than it is on the second and third. I consider each in turn. This is an analytic convenience only. As I argued before, none of these three elements could create Linux without the other two. They are interdependent in ways I will highlight below.

Individual Motivations

The fun, enjoyment, and artistry of solving interesting programming problems clearly motivate open source software developers. To the extent that choosing your own work reduces perceived drudgery, it might be better to commit your time to an open source project than to a hierarchically-managed project where someone else can tell you what to do. Drudgery-type tasks should get taken of, in the aggregate, by two means. There may be someone in the community that finds a particular task challenging and interesting, even if it appears annoying to most. And there may be others for whom solving a particular problem, even if mundane, is important to a job responsibility. In both cases, the system benefits from increased numbers of users: there is more variation in what people find inherently interesting, and as Linux use expands and more people use it in their organizations they will write patches and drivers to help them in their daily work and share those 'boring' pieces of code with others.

Reputation also is clearly an important motivator. Of course this depends on a de facto intellectual 'property rights' system, in the sense that contributors must be assured in advance that they will get appropriate credit for what they do. As there is no law or patent system to provide this assurance, the system rests upon trust in a leader who reliably grants credit in a readme or history file and does so over time in ways that are perceived as fair. This supports directly the Lerner/Tirole argument about signaling. To the extent that programmers seek reputation with the intention later to monetize that reputation, reputations must clearly signal the importance and technical sophistication of a programmer's particular contribution. Peer review makes sense, as Lerner/Tirole point out, because excellent code is craftsmanship and there is no precise metric for quality outside the judgement of clever peers.

Reputation of this sort also functions to attract more cooperators. The logic here is that people seeking to generate good reputations for themselves will preferentially cooperate with other high quality coders since the marginal payoff from hard work in a joint project is higher when your collaborators are as good or better than you are. The best developers are motivated to stay in this relationship because they can move upwards within the informal pecking-order and perhaps even become 'lieutenants' with authority over sub-systems of Linux, a position with substantial reputational payoffs.

Within the open source community reputation is clearly more than instrumental. Apart from monetizable benefits in job offers or access to capital, or even the perception of personal efficacy that results from being part of a successful creative enterprise, ego-boosting is valued in and of itself (although harder to 'quantify'). The trick, as I pointed out earlier, is for the system to generate ego-boosts for good performance without causing too much damage to egos when code is rejected or superceded. The open source community manages this trick at least in part by focusing its criticism on the *code* rather than on the *person* who produces it (a notably different norm than exists in academia, where arguments are tightly associated -- on the upside *and* the downside -- with the individuals who create them).[53]

[53] Raymond puts it this way: "bug hunting and criticism is always project-labelled, not person-labelled". "Homesteading the Noosphere", p. 110.

Reputation is a powerful motivating force for open source developers. But there are strong reasons to believe that reputational concerns by themselves cannot explain successful collaboration. If reputation were the primary motivation, we should be able to tell some version of the following story. Assume that project leaders like Torvalds receive greater reputational returns than general programmers do. Then programmers should compete to become project leaders, particularly on high profile projects, of which there are a limited number. This competition could take at least two forms. We would expect to see a significant number of direct challenges to Torvald's leadership -- but in fact there have been few such challenges, none serious. Alternatively, we could see 'strategic forking'.[54] A strategic forker would fork a project not for technical reasons per se, but rather simply to create a new project that he or she could lead. The problem of how to attract other programmers would be managed by credibly promising to maximize other programmer reputations on the new project -- for example, by sharing some of the gains with anyone who joins. In that case, a new programmer would be motivated to join the forked project rather than Linux.

The problem with this kind of story is that it simply hasn't happened. There are no significant examples of this kind of behavior in the Linux history. And given the story-telling culture of the community, it is certain that any such story would have become part of the general lore. Nor does it seem that the 'system' or perhaps Torvalds has anticipated this kind of pressure and pre-empted it by his own strategic behavior, since the logic here would drive a leveling of reputational returns throughout the community. The bottom line is that there simply is not as much strategic behavior in reputation as we would expect, if the Lerner/Tirole emphasis were correct.

Part of the counterpoint to a competition in reputation comes from strong elements of shared identity within the community of developers. Steven Levy in his 1984 book Hackers chronicled the development of 'hacker culture' around MIT in the early 1960s.[55] Levy described the following key tenets:

- *Access to computers should be unlimited and total -- the so-called 'hands-on' imperative*. This notion was considered radical in the pre-PC era, when computer usually meant a single mainframe machine controlled by a 'priesthood' of sorts that rationed time and access to officially sanctioned users. Today the idea of access seems obvious to just about everyone -- at least when it comes to hardware.
- *Information should be 'free'*. Richard Stallman would later become the most vocal champion of the principle that software -- as an information tool that is used to create new things of value -- should flow as freely through social systems as data flows in bits through a microprocessor.
- *Mistrust authority and promote decentralization*. In the 1960s it was not Microsoft but rather IBM that was the icon of centralized, hierarchical authority relations in the computing world. Hierarchies are good at controlling economic markets and more importantly computing power. Control stifles creativity -- which is ultimately what information processing ought to be about.
- *Judge people only on the value of what they create not who they are or what credentials they present*. This is the essence of a relatively pure meritocracy -- anyone can join and be recognized for the quality of their work, regardless of age or degree status or any other external sign. There are costs involved -- since credentials often act as proxies that save time and energy spent evaluating the substance of an individual's capabilities. But there are also benefits -- more energy goes into creating code and less into self-promotion and bragging -- since your work brags for you.
- *People can create art and beauty on computers*. In the hacker community, software by itself took on the status of an artifact or work of art, that was valued not only for what it allowed a user to do, but also for its own inherent beauty and elegance. There is a telling aesthetic vocabulary in use to describe different programs, such as 'clean' code and 'ugly' code (rather than 'efficient/inefficient', for

[54] Ilkka Tuomi uses this phrase in "Learning from Linux: Internet, Innovation, and the New Economy,", an important working paper of April 15 2000.
[55] Steven Levy, Hackers: Heroes of the Computer Revolution. Garden City NY: Doubleday, 1984.

example). Some programmers talk about a "Unix philosophy" which they describe in similar aesthetic terms.[56]

- *Computers can change human life for the better*. There is a simple and familiar instrumental aspect to this belief -- that computing takes over repetitive time consuming tasks and frees people up to do what is more creative and interesting. There is also a broader notion, that cultural models and practices learned in working with information systems could transfer, for the better, to human systems. This belief obviously varies in intensity among developers -- as I suggested earlier, the Open Source Initiative has worked to downplay the explicit political agenda that was so central to the Free Software Initiative. But the political message comes through, at least implicitly, in almost any interview and causal discussion with open source developers. With different degrees of self-consciousness, these individuals know they are experimenting -- with economic and social systems of production -- and that the results of these experiments *could*, at least, have broader ramifications beyond the realm of computer software.

In summary, individual motivations are a key part of the explanation for open source software. Previous authors have pieced together much of these microfoundations. Individuals' motivations are supported by, and in turn support, a set of roughly shared beliefs that make up an informal hacker culture. This culture now has an extensive history and is part of a strong and highly valued legacy among a segment of software developers. But none of this is sufficient to explain Linux, if for no other reason than the fact that these motivations and cultural attributes characterize only a subset of the software developer population. Even if a system resting on these microfoundations were internally stable (and I have not shown that it would be so), it would be vulnerable to 'invasion' in the game theoretic sense of that term, from actors who sit outside the community and are motivated by other more crass considerations. The relative absence of 'strategic forking' and other kinds of expected behaviors in that story point to the importance of a macro-structure, exogenous to and autonomous from individual motivations of developers. There are economic and social/political elements to that macro-structure.

Economic Logic

The starting point for most economic analyses of open source is a standard collective action type analysis.[57] Open source software is the epitome of a non-excludable good. It is also non-rival in the sense that any number of users can download and run Linux, without decreasing the supply that remains for others to use. In this context the economic puzzle is straightforward. For well-known reasons public goods (non-rival and non-excludable) tend to be underprovided in non-authoritative social settings. The situation with Linux ought to be at the worse end of the spectrum of public goods since the software depends on 'collective provision' -- it would not exist without contributions from a large number of developers. Why would any particular person choose to contribute -- voluntarily -- to a public good that he or she can partake of unchecked as a free rider? Since every individual can see that not only her own incentives but the incentives of other individuals are thus aligned, the system ought to unravel backwards so that no one makes substantial contributions, and there is no public good to begin with.

Part of the answer lies in individual motivations I explained above, that go beyond the narrow version of rational calculations that generate standard arguments about collective action problems. But there is also at play a larger economic logic that reframes the problem of collective action in this particular setting.

---

[56] Mike Gancarz, The Unix Philosophy. Boston: Butterworth-Heinemann, 1995.

[57] See the summary and intelligent if sometimes polemical critique by Eben Moglen, "Anarchism Triumphant: Free Software and the Death of Copyright," First Monday Issue 4.

Rishab Aiyer Ghosh takes an important step toward explicating that logic.[58] Using the image of a vast tribal cooking pot, he tells a story about collective action where one person puts in a chicken, someone else puts in carrots, another person puts in some onions…and everyone can take out delicious stew. Non-excludability is an issue because if anyone can take bowls of stew out of the pot without necessarily putting anything in, there might be nothing left for those who did contribute and thus they are unlikely to put anything in in the first place.

But what if the cooking pot was magically non-rival -- if production of stew simply surged to meet consumption without any additional expenditure of effort. Then everyone knows that there will always be stew in the pot and that even if people who don't put anything in take some out, there will still be plenty left for those who did contribute. In this setting an individual faces a different calculus: "you never lose from letting your product free in the cooking pot, as long as you are compensated for its creation."[59]

The Internet makes a digital product like software 'magically' non-rival. Because a digital product can be copied an infinite number of times at no cost, this is a cooking pot that creates as many bowls of stew as there is demand. Given that, how are you going to be compensated for creating your product? In the digital environment this too is nearly straightforward although slightly more subtle. When I put my product into the cooking pot, I am giving away in effect a thousand or a million or really an infinite number of copies of that product. However, in my private utility calculations, multiple copies of this single product are not very valuable -- in fact the marginal utility of an additional copy *to me* is de facto zero.

But single copies of multiple products are, *to me or any other single user*, immensely valuable. In practice, then, I am giving away a million copies of something, for at least one copy of at least one other thing. That is a good trade and clearly a utility enhancing one for anyone who makes it. 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 Ghosh's argument is that it does not explain the 'trade'. What is the underlying story that accounts for an exchange relationship here? In fact no trade is necessary at all. It is still a narrowly rational act for any single individual to take from the pot without contributing -- and free ride on the contributions of others -- which means that the collective action dilemma is still fundamentally in place. The system would unravel not because free-riders use up the stock of the collective good or somehow devalue it, but because no one yet has any real incentive to contribute to that stock in the first place. The cooking pot will likely be empty.

I believe the solution to this puzzle lies in pushing the concept of non-rivalness one step further. Software in some circumstances is more than simply non-rival. Operating systems like Linux in particular, and most software in general, are actually subject to positive network externalities. Call it a network good, or an anti-rival good (an awkward but nicely descriptive term). What I mean by this is that the value of a piece of software to any particular user increases, as more people download and use the same software on their machines.

Compatibility in the standard sense of a network good is one reason why this is so. Just as it is more valuable for me to have a fax machine if others also have fax machines, as more computers in the world run a particular operating system it is easier to communicate and share applications and files across those computers, making each of them more valuable to the user. Open source software makes an additional

---

[58] Rishab Aiyer Ghosh, "Cooking Pot Markets: An Economic Model for the Trade in Free Goods and Services on the Internet," First Monday Issue 3.

[59] Ghosh, "Cooking Pot Markets" p. 16.

and very important use of network externalities, in debugging. Remember the argument that there exist an infinite number of paths through the lines of code in even a moderately complex program. The more users actively engaged in running a piece of software, the more likely that any particular bug will surface somewhere. And once a bug is identified it becomes possible to fix it, improving the software at a faster rate.

The point is that open source software is not simply a non-rival good in the sense that it can tolerate free riding without reducing the stock of good for contributors. It is actually anti-rival in the sense that *the system positively benefits from free riders*. Some small percentage of free riders will provide something of value to the system, even if it is just reporting a bug out of frustration. The more free riders in this setting, the better.

This argument holds only if there are a sufficient number of individuals who do not free ride -- in other words, a 'core' group that contributes to the creation of the good. We have already seen a set of motivations and incentives that taken together in different proportions for different individuals might inspire their active contributions. In more traditional language, highly interested people may be willing to contribute to a non-rival good, even if the good is non-excludable, because they are gaining other benefits (i.e. reputation, identity satisfaction). As the size of the group increases, if you assume a heterogeneous group with outliers who have a high level of interest and surplus resources of time and mindspace, then a large group is actually *more* likely to provide the good than is a small group. This is a twist on standard collective action arguments, where large groups are *less* likely to generate collective goods because of the difficulties that individuals have in making their particular contributions visible, and because of coordination problems among large numbers.[60]

The next section explains how Internet technology enables the social structure of open source communities to compensate for those challenges. Consider for the moment two significant implications. First, the economic logic of open source as I have described it suggests an intriguing spin on intellectual property rights (IPR) arguments. The starting point for conventional IPR thought is that the key challenge lies in creating incentives for the innovator by making sure that his or her private return is sufficient. The cost for doing that is usually a reduction in social return. If IPR regimes provide incentive to innovate, then distribution of that innovation takes care of itself through the market, in the sense that high quality and beneficial technology will reach at least some and probably many of its potential beneficiaries (depending on the price). Open source suggests an inversion of that logic. In this setting, the key challenge was developing incentives that were set appropriately to promote distribution, and letting innovation take care of itself.

The second implication is closely related, but more general. Analyzing open source software through the lens of collective action begins to look subtly misleading. Linux now appears less like a complicated collective action problem, than a complicated *coordination* problem. Contributions to innovation are not problematic in this setting. The more important challenge is to *coordinate those contributions on a focal point*, so that what emerges is technically useful. The third and critical step in explanation, then, is to make sense of the social and political structures that maintain coordination and manage complexity in the system.

---

[60] Russell Hardin, Collective Action. Baltimore: Johns Hopkins University Press, 1982, pp. 67-89 provides an extensive discussion of the group heterogeneity argument. See also. Gerald Marwell and Pamela Oliver, The Critical Mass in Collective Action: A Macro-social Theory. New York: Cambridge University Press, 1993, particularly chapter 3.

Maintaining Coordination and Managing Complexity

Coordination

The most important remaining question is this: why don't open source projects succumb to code forking? Coordinating the efforts of a large number of developers and managing the complexity of the resulting technical artifact requires an efficient organizing structure. Software engineers know from experience that code forking or its equivalent is a key threat in any large-scale development project. Some developers consider it almost a normal part of the process -- as individual developers and teams of developers build pieces of software there is a natural tendency to evolve multiple and inconsistent versions of the code base, leading the project off in different and often incompatible directions.

In the commercial software world, authoritative decision making within corporate boundaries cuts this process off before it goes too far. Outside of corporate boundaries code forking is much harder to restrain. Notice that this is precisely what did happen to Unix in the commercial world, resulting in a large number of incompatible proprietary versions of the software -- forked code. How has Linux managed this challenge?

Individual and social 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.

A similar argument applies even if 'trading' is not an important part of developers' mindsets. Open source creates a mildly counterintuitive dynamic: the more open a project is and the larger the existing community of developers, the less tendency to fork. It becomes very hard for a renegade to credibly claim that he or she, as a forker, could accumulate a bigger and better community of developers than already exists in the main code base.

A forked minority code base could not then 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 it could not provide as attractive a payoff in reputation, even if reputation were shared out more evenly within the forked community.[61] Reputation depends at least in part on the number of people who know of your repute, which means the larger the user base the more reputation you receive in return for any particular innovation. Forking would only split the 'audience'. There is another disincentive here: if you fork code, you can control your own contributions and thus your reputation -- but if others then fork your code and develop it further without your input, you could be exposed to reputation risk that would be difficult for you to control.

The open source community has developed a set of cultural norms that support this logic of incentives.[62] A prevalent norm assigns decision-making authority within the community. The key element of this norm is that *authority follows and derives from responsibility*. The more an individual contributes 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

[61] 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.

[62] 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.

'lieutenants' by consulting closely with them on an on-going basis, particularly when it comes to key decisions on how subsystems are to work together in the software package.

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. There is an additional, auxiliary norm that 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."[63]

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 born in the notion of compatibility between platforms, ease of networking, and positive network effects.[64] Unix programmers have a set of common standards for what is 'good code' and what is not-so-good code.[65] 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 ideology or zealotry (more characteristic of the Free Software Foundation). A cultural frame based in engineering principles (not anti-commercial ideology) and focused on high reliability and high performance products 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 legitimating concerns about 'maintainability' of code, 'clean-ness' of interfaces, clear and distinct modularity.[66] The clear mastery of technical rationality in this setting is made clear in the creed that developers say they rely on --- 'let the code decide'.

Leadership matters in setting a focal point and maintaining coordination on it. 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: that 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. Importantly, he provides a convincing example of how to manage the potential for ego-driven conflicts among very smart developers. Torvalds downplays his own importance in the story of Linux: while he acknowledges that his decision to release

---

[63] Raymond, "Homesteading the Noosphere," p. 127. One interesting additional piece of evidence for these norms is what has happened when the two norms pointed in different directions. Raymond (p. 128) 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".

[64] 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.

[65] Mike Gancarz, The Unix Philosophy.

[66] Ilkka Tuomi, "Learning from Linux".

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."[67]

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.[68] He continues to emphasize the fun and opportunities for self-expression in the context of "the feeling of belonging to a group that does something interesting" as his principle motivation. 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 decisions -- in the language of technical rationality that is currency for this community. Would a different 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). The logic of the argument to this point supports that belief.

There do exist sanctioning mechanisms to support the nexus of incentives, cultural norms, and leadership roles that maintain coordination. In principle, the GPL and other licenses could be enforced through legal remedies (this of course may lurk and constrain behavior even if it is not invoked). In practice, precisely how enforceable in the courts some aspects of these licenses are, remains unclear.[69] The sanctioning mechanisms that are visibly practiced within the open source community are two: 'flaming' and 'shunning'.[70] Flaming is 'public' condemnation (usually over email lists) of people who violate norms. "Flamefests" can be quite fierce in language and intensity, but tend ultimately to be self-limiting.[71]

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 that the community offers. It is not the same as excludability: 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, while the community can and does draw on its collective experience and knowledge to do the same. This is clearly a strong disincentive to strategic forking, for example, but it also constrains other less egregious forms of counter-normative behavior (such as aggressive ego self promotion).

The focal point logic of open source adds up to an ecosystem (not necessarily a 'market') with a winner-take-all dynamic.[72] "Liquidity" or its functional equivalent matters. There are substantial first-mover advantages. Developers are incentivized to join projects that are already large, successful, well managed, and productive. It is difficult to jump-start alternative projects in ecological niches (or market spaces) that are currently occupied.[73] A successful open source project like Linux could, by this reasoning, come

---

[67] "What Motivates Free Software Developers," Interview with Linus Torvalds, First Monday Issue 3.

[68] The documented history, particularly the archived email lists, supports Torvalds on this point.

[69] 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.

[70] Eric Raymond, "Homesteading the Noosphere," p. 129.

[71] The intensity seems to be self-limiting, in part because developers understand very well the old adage about sticks and stones vs. words.

[72] This is characteristic of economies with positive network externalities.

[73] This can have dysfunctional consequences that I consider in the conclusion.

to dominate functional categories. In fact, Linux has overwhelmed BSD Unix, absorbed its key ideas, and attracted much of the brain power that was going into it. Linux is taking away market share from commercial versions of Unix. How far this dynamic can progress may be limited principally by the open source model's ability to manage increasing complexity of the production process and the software itself.

Complexity

Standard arguments in organization theory predict that increasing complexity in a division of labor leads to formal organizational structures.[74] In contrast, much recent literature on the Internet and the 'new economy' argues (or sometimes asserts) that formal organizational structures are radically undermined by technologies that reduce the costs of communication and transactions.

Both of these arguments have a point. The Internet does reduce communications and transaction costs in many situations. As expected, this affects existing boundaries of organizations and industries -- sometimes in a revolutionary way. Schumpeter coined the phrase creative destruction to describe this kind of process. The creative half of that aphorism is a reminder that as old boundaries and organizations erode, new organizational structures are likely to arise. "Self-organization" is not an adequate theoretical explanation of this process, and it is not an accurate empirical description of open source software projects (or for that matter, anything else in the new economy).[75] Put simply, the Internet does not solve the problem of organization. Clearly, the Internet decreases or removes the costs of geographically widespread and time-dependent collaboration. But it does not remove other costs of collaboration -- decision making, human emotion and ego, etc. These need to be managed by other means. In principle, the Internet can increase the difficulty of collaboration since it can be set up to be non-excludable.

Software is a complex technical artifact. A typical full distribution of Linux contains more than 10 million lines of code. The open source process that produces, maintains, and extends this code base is also complex -- as I have explained. Managing complexity in this system depends on characteristics of the software itself, and characteristics of the social/political structure that embeds it.

The key to managing the level of complexity within the software itself, is *modular design*. 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 small programs are easy to understand, easy to maintain, consume fewer hardware system resources, and -- most importantly -- can be combined with other small programs to enable more complex functionalities. Unix programmers are taught to build integrated applications by constructing them out of a collection of small, unifunctional components.

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. The modules generate discrete outcomes and communicate with each other through 'pipes'. Programmers working on one module know

---

[74] For example see Henry Mintzberg, The Structuring of Organizations: A Synthesis of the Research. NJ: Prentice-Hall, 1979.

[75] An adequate theoretical argument about self-organization needs to specify the micromotives that drive individual units to organize themselves without any recourse to external structures or authority. It is possible to generate loose evolutionary analogies and computer simulations that show self-organization, but I have yet to see a convincing case in the internet economy -- or any other kind of economy -- since these depend on at a minimum on technologies as well as legal and financial structures that are exogenous to 'self-organization'.

two things: that the 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 spread out from a code change. Obviously, it is a powerful way to facilitate working in parallel on many different parts of the software 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. It is notable that one of the major improvements in Linux release 2.0 was moving from a monolithic kernel to one made up on independently loadable modules. The advantage, according to Torvalds, was that "programmers could work on different modules without risk of interference… 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."[76]

Torvalds' implicit point is simple: these engineering principles are important because they reduce organizational demands on the social/political structure. In no case, however, are those demands reduced to zero. This is simply another way of saying that libertarian and self-organization accounts of open source software are frankly naïve. The formal organization of authority is quite structured for larger open source projects. Torvalds, as I explained, sits atop a decision pyramid as a de facto benevolent dictator. Apache is governed by a committee.

The success and increased visibility of open source software over the last several years has brought with it 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.[77] 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 it is attractive to offer software that is good enough that it needs 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, and when they need it.

In February of 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."[78] The Apache Software Foundation is now formally incorporated as a non-profit and led by a board of directors.[79]

---

[76] Linus Torvalds, "The Linux Edge,", in Open Sources, p. 108.
[77] Paul J. DiMaggio and Walter W. Powell, “The Iron Cage Revisited: Institutional Isomorphism and Collective Rationality in Organizational Fields,” American Sociological Review 48. 1983
[78] http://opensource.org/for-hackers.html#marketing
[79] 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

Some degree of institutional isomorphism reduces the complexity of relationships between the open source process, and the increasing range of organizations that use open source software. How far this process can and will go is an important question. It is particularly important given the huge financial stakes that now exist 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.

V. Conclusion

Open source projects have demonstrated that a large and complex system of code can be built, maintained, developed, and extended in a non-proprietary setting where many developers work in a highly parallel, relatively unstructured way and without direct monetary compensation. The resulting piece of knowledge -- a technical artifact called software -- is remarkably powerful, efficient, and robust.

This paper explains the open source process -- how it came into being, how it works, and how it is organized. I use a compound argument about individual motivations, economic logic, and social structure to account for a process that is partly about collective action but even more fundamentally about coordination. This yields insights into the nature of collaboration in the digital economy. It may also yield insights about how human motivations operate in that setting. At a minimum these are important, pragmatic demonstrations about a production process within, and a product of, the digital economy, that is really quite distinct from modes of production characteristic of the pre-digital era.

This paper is not in any sense an attack on rational choice models of human behavior and organization. I do not see evidence for an explanation based on altruism (rational or otherwise). I do not accept the term 'self-organization' as a placeholder for undisclosed microfoundations or social mechanisms. Of course, I recognize that all human activity is not motivated by rational behavior or necessarily monetized. After all, most art is never sold but simply given away, and only a tiny proportion of poetry is ever copyrighted. But computing and the creation of software, even with its 'artistic' and creative elements, has become deeply embedded in an economic setting. The stakes are huge. Richard Stallman and the Free Software Foundation (among others) decry this fact from a moral perspective, but that does not make it untrue. An 'uneconomic' explanation of open source, if such a thing were really possible, would be uninteresting and also probably wrong. The challenge is to add substantive content to a 'rational' and 'economic' understanding of why people do what they do, and how they collaborate, in open source.[80]

This paper also is not an analysis of business models connected to open source (though that is itself a fascinating issue). I do not try to explain or justify the flip-floppy stock valuations of companies that provide services around Linux. It is not a paper about theories of innovation, intellectual property rights, or copyright per se (although I touch on each of these points). The open source process can be mined for insights and arguments in each of these areas, and I hope these disclaimers will be seen as an invitation to take on some of these questions.

The perspective that I have adopted in this paper yields its own important implications. I consider four in conclusion: the impact of the internet, the importance of path dependence, the politics of standards, and

[80] In my view, theorizing about collective action is not a matter of explaining whether or not it is rational. It is a matter of understanding under what conditions individuals find that the benefit of participation exceeds their cost. (this means, of course, understanding a great deal about how particular people in particular situations assess costs and benefits).

possible international distributional consequences. Finally, I engage in some (disciplined) speculation about the possible scalability and extension of the open source process to other realms of production.

The Internet

The primary impact of Internet communication is to reduce the costs associated with geographically widespread collaboration. This is a key element of the economic logic behind open source -- almost certainly a necessary (though not sufficient) condition. The open source process leverages large numbers of possible contributors, who make up a group that is heterogeneous in level of interest and resource endowments of time and mindspace. The good they contribute to is non-excludable. But it is also what I called 'anti-rival' in the sense that free riding is not merely neutral, it is indeed a positive thing, *once the core good has in fact been 'provided'*.

Given the individual motivations I laid out and the condition of heterogeneity, a large group is more likely to generate that core under these conditions than is a small group. The major counterpoint to this, is the costs of communicating within and organizing that larger group. The Internet reduces those costs dramatically. It also reduces the costs of free riding, which in this setting is good. Each free rider actually adds value to the final product. If the free rider is completely passive, she adds value by adding market share (and thus increasing the space for reputational returns to contributors). If she is just a little bit active, the free rider becomes a de facto tester and might just report a bug or request a new feature. The Internet reduces the communication costs of doing all that to just about zero.

The massive bandwidth of Internet communication is a change in kind, not just in degree. Geographical dispersion of developers might have been possible without the Internet -- people could have mailed tapes, CDs, and disks back and forth, or sent code over fax machines. This is expensive and awkward, but not impossible. Internet communication reduces those costs to a level where it is possible to engage easily in geographical dispersion. What is more fundamental is the ability to employ *functional dispersion*, and move close to real-time parallel processing of a complex task.

The Internet makes it possible for the open source community to engage in not only a more finely grained division of labor, but also truly *distributed innovation*. In a division of labor, no matter how fine, there is still a value chain (even if part of it is located in Cupertino and another part in Bangalore). The problem is still about getting the weakest link in that chain to deliver.[81] But in parallel distributed innovation, at the limit there is no weak link because there is no chain. Coordination costs may still constrain the distribution of innovation to some extent, and thus constrain in turn the maturation of parallel processing as a mode of production. But that is now a function of getting the social organization 'right', not technology per se.[82]

Path Dependence

The open source process has interesting implications for path dependent economics and the dynamics of increasing returns. One way to read the story of Linux is to say that this community has found a way to

[81] In other words, the rate-limiting factor is the efficiency of the least efficient link. See Gary Becker and Kevin Murphy, "The Division of Labor, Coordination Costs, and Knowledge Quarterly Journal of Economics. 107. Nov 1992 pp. 1137-1160

[82] The underlying argument is F.A. Hayek, "The Use of Knowledge in Society," American Economic Review 35. Sept 1945, pp. 519-30.

make the jump from QWERTY to DVORAK.[83] In this interpretation, other operating systems (primarily Microsoft products) are sub-optimal performers but are locked in by increasing returns and high costs of switching. A jump to a more optimal path would bring better returns in the long run, but it is hard to engineer because it requires a massive up front investment that no one has an incentive to provide (and the incumbent has great incentives to block). Coordinating the investment 'jump' among all the people who would have to make it, to make it worthwhile, is impossibly expensive -- even if there is a clear focal point for the alternative equilibrium. The system remains stuck in a sub-optimal path.

Linux is clearly challenging that equilibrium and may overturn it in the not so distant future. If this happens, the open source community will have demonstrated one way to successfully invest in a jump to a higher performance path. Traditional economic analysis will suggest that this was the outcome of subsidization -- after all, Linux is essentially free and thus the community of developers heavily subsidizes the jump. That is not wrong, but we should be careful not to assume typical related assumptions about the reasons why that subsidy is being offered. These developers are not making standard calculations about the returns that they can gain 'later on' nor are they providing subsidies in the interest of later exploiting monopoly power once they own the market. Indeed, the form of the subsidy (giving away the source code) is nearly a binding commitment *not* to do that.[84]

But open source may not be able to overcome its own related deficiency of path dependent economies. Would an operating system market dominated by Linux get stuck over time in its own sub-optimal architecture? The open source process has proved itself very good at incremental change. But might it keep a community of developers tracked on debugging and modifying a system with an old architecture (which could become flawed and inefficient simply because hardware and desired functionalities change), when they would actually be better off starting from scratch or at least revising more fundamental architectural decisions? In principle it may be harder, not easier, for a non-hierarchical community of this kind to move to a fundamentally new architecture -- and there may come a time when many wish there was a 'boss' who had the authority to order that shift. This is not news to developers -- Raymond for example recognizes the potential problem but is confident that the open source process will manage it successfully. Clearly how this plays out bears watching.

Standard Setting

Open source processes are powerful 'magnets' that attract standards to form around them. The economic logic (anti-rivalness, increasing returns, etc) is part of the reason. The other part is that the open source process removes intellectual property barriers that hamper standard setting. The commercialization of Linux suppliers does not change that. If one Linux supplier innovates and that innovation becomes popular in the market, other Linux suppliers will adopt the innovation more or less quickly -- open source ensures that they have access to the source code and the right to use it.

When different Linux distributions try out particular innovative paths, skeptical observers argue that Linux is balkanizing in the same way that Unix did, into incompatible proprietary versions. In retrospect,

[83] The allusion is to Paul David, "Clio and the Economics of QWERTY," American Economic Review 1985. David's argument has been subject to critique (see S. J. Liebowitz and Stephen E. Margolis, "The Fable of the Keys," Journal of Law and Economics April 1990) but I use it here for its value as analogy.

[84] Part of the business model puzzle that companies like Red Hat and VA Linux are trying to solve, is finding other ways to make money in this setting. But these business models remain experimental and in any case came into play after much of the subsidy was already in place. I want to pre-empt any argument that anyone might later make about forward-looking strategic rationality with standard motivations, because it simply doesn't fit with this history.

a better interpretation is simply that the system as a whole is testing out different methods of solving a problem, to see which is preferred. In 1997, for example, there was a substantial disagreement about the use of certain libraries (these are standardized pieces of software that….. EXPLAIN). Red Hat adopted a newer 'glibc' library while many other popular Linux releases stuck with the older 'libc' libraries. As Robert Young explains, "the debate raged for all of six months…and as 1998 drew to a close all the popular Linux distributions had either switched or announced plans to switch to the newer, more stable, more secure, and higher performance glibc libraries".[85] As Linux gains market share its magnetism as a standard will likely increase.

The market power of open source standards may also 'infect' other kinds of software outside of its immediate market. Bruce Perens relates a particularly interesting example of this dynamic.[86] In 1998, Linux developers were working to create an object-oriented desktop interface for the operating system. The first attempt to do this used an application called KDE, which was itself licensed under GPL, but depended on a proprietary graphical library named Qt (from a company called Troll Tech). To use Qt in most circumstances required a developer's license, at $15,000. In strict terms, then, KDE was not open source.

The promise of a graphical user interface for Linux was powerful, and many developers were willing to look past the problems with Qt or fudge the open source definition in this case in order to move forward. That temptation met with very strong resistance. A set of developers began working on a new project called GNOME that would compete with KDE while using a fully free graphical library (which was generally seen as technically inferior to Qt) . Another set of developers initiated a project called Harmony, to create a fully open source Qt clone for KDE. As GNOME improved and Harmony demonstrated that it could in fact replace Qt, Troll Tech recognized that Qt risked by surpassed by others and would be effectively shut out of the Linux marketplace unless it changed the license. The result was that Troll Tech released a fully open source license for Qt.

How these dynamics are likely to affect the politics and particularly the international politics of standard setting remains unclear. These politics are typically analyzed around bargaining power between and among firms and national governments, and (usually) with a lesser role for international institutions. Open source adds an interesting twist to the analytic problem. The open source community is not a firm (though there are firms that 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. The technological community that produces Linux was international from the start and remains highly international in scope; its motivations, economic logic, and social/political structures bear very little obligation to national governments.

We simply do not know how this community will interact with formal standards processes, which are deeply embedded in national, international, and global politics. It is plausible that the process may reiterate some of what we know about the influence of epistemic communities in environmental issues or non-governmental organizations in human rights issues. It is also possible that governments will try to promote certain standards around open source, or manipulate the dynamic in ways that would yield national advantage.[87] Yet it is difficult to see right now a viable strategy by which governments could achieve lasting advantage in this way.

---

[85] Robert Young, "Giving it Away: How Red Hat Software Stumbled Across a New Economic Model and Helped Improve an Industry," Open Sources, p. 124.

[86] Bruce Perens, "The Open Source Definition," in Open Sources, p. 175.

[87] For example, some French lawmakers have proposed amending the European Software Law of 1991 in order to mandate that governments only purchase software for which the source code has been released. See Sam Williams, "Open Season: French Law Would Increase Code Accessibility," Upside Today 1 May 2000

International Distributional Consequences

Is open source software likely to be a significant factor in the international economy of the early 21st century?

The answer to this question depends first on whether software per se, and information technology more generally, turns out to be 'merely' a new leading sector of the economy, or more profoundly a transformative tool that sweeps across and revolutionizes economic activity across a very wide range of sectors.[88] If the former, the impact of open source will be felt primarily in computing and information processing. There will be ripple effects in other areas of economic activity, but the causal force of software production models will be relatively limited overall. If the latter, it is reasonable to expect new possibilities for organization to arise across the economy, with massive changes in what is done and how it is done in a wide range of industries. Changes in how governments regulate economic activity, how property rights and control are imagined and implemented, will accompany this shift. In that scenario, the impact of the open source process could be orders of magnitude more significant. I believe that the second scenario is closer to the truth, but since it cannot be proved at this point in time, I take a restrained view of the possibilities below.

Software itself is ultimately a tool for manipulating information. If the tool is essentially free to anyone who wants it and freely modifiable to make it useful in whatever way the user can manage, then lots of people will grab the tool and experiment in doing different things with it. This is just as true across countries as it is within countries. The open source community has been international from the start and remains so. It transcends national boundaries in a profound way because its interests and its product are not tied to or dependent upon any government. Developers in China, Indonesia, and other developing countries contribute to open source software. More important, they all have access to the tool. The degree to which they can modify and fix this tool by themselves is limited only by their own knowledge and learning, not by property rights or prices imposed on them by a developed country owner.

For this reason open source software is likely to be a powerful instrument of bootstrapping. Contrast this with the post World War II era of development economics, where debates about technology transfer to developing countries raged around the question of 'appropriate technology'. At that time, developed country governments and international development institutions were making decisions about what was 'appropriate technology' to transfer to developing countries. Open source software shifts the decision-making prerogative into the hands of people in the developing countries. Andrew Leonard implies something similar when he says "every dollar Microsoft spends attacking software piracy in the third world [sic] is a dollar of advertising for Linux and free software."[89]

We should be cautious in thinking about what the free diffusion of tools means for the world economy. Certainly it will not create a profound leveling phenomenon that some information idealists would like to see as possible. Even when everyone has access to tools, some people can and will use those tools to add more value than others. The analogy is to an imaginary world where everyone had access to as many steam engines as they wanted, all at the same time, and for nearly no cost. It is still the case that

---

(www.upside.com/texis/mvm/open_season). This is a long way from legislation, but it illustrates what a government might in fact be able to leverage its market power to do.

[88] Stephen Cohen, Brad deLong and John Zysman, Tools for Thought BRIE Working Paper, Spring 2000.

[89] See http://www.salon.com/tech/fsp/outline/index.html.

development in the industrial era would have been uneven. But it very well might have been *less drastically uneven* than it is today.

There are deeper and more abstract possibilities to consider. Poor countries are often thought to lag in productivity because their economies are stuck in a less advanced division of labor. The sticking point is a lack of effective institutions for creating and maintaining coordination among specialized producers. The unfortunate result is that the division of labor as well as the size of production 'teams' is limited by incentives to shirk. Efforts to extract rents by 'holding up' other members of the team are prevalent.[90] As Becker and Murphy put it, in the absence of efficient and reliable institutions to compensate, "principal-agent conflicts, hold up problems, and breakdowns in supply and communication all tend to grow as the degree of specialization increases."[91]

The intriguing thing is that the open source process bypasses much of these impediments. Again we should be cautious. I am not arguing that developing countries can use the open source process to make up for lack of sufficient legal and economic infrastructure, or replace institutions by installing high bandwidth connections to the Internet. I am saying that there are interesting possibilities for developing more specialized divisions of labor around open source processes, or inserting into a more developed global system of distributed innovation, in ways that depend somewhat less on more traditional institutional infrastructures. Either could have a significant impact on the international distribution of wealth.

Extending Open Source

The most intriguing remaining question about open source is does the model extend to other realms of production. To frame the question more precisely, what are the boundary conditions of that extensibility? The key concepts -- user-driven innovation that takes place in a parallel distributed setting, distinct forms and mechanisms of cooperative behavior, 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.

There are two parameters around which to organize hypotheses about the conditions under which an open source process will be viable. The parameters are the *nature of the task* and *the motivations of the agents*.

The arguments in this paper suggest the following hypotheses about the nature of the task.

An open source process will work more effectively when:

- Contributions depend not on proprietary techniques but on knowledge that is widely available.
- There is a substantive 'core' that holds a promise of becoming something quite interesting.
- The product is perceived as important, valuable, and of widespread use.
- The product has a complexity of the kind that can be disaggregated into parallel modules.
- There are strong positive network effects.

The following hypotheses relate to the motivations of the agents.

An open source process will work more effectively when:

[90] Oliver E. Williamson, The Economic Institutions of Capitalism: Firms, Markets, and Relational Contracting. New York: Free Press, 1985.
[91] Gary S. Becker and Kevin M. Murphy, "The Division of Labor, Coordination Costs, and Knowledge, Quarterly Journal of Economics 107. November 1992, p. 1141.

- Contributors are confident that their efforts will actually generate a product, not simply be dissipated.
- Contributors value status and reputation at least in part as symbolic rewards, in addition to the instrumental (ie monetizable) aspects.
- Contributors gain knowledge by contributing to the project; they learn as they go.
- Ego is an important motivating factor.
- Contributors believe they are doing something that is 'good' or 'noble', or at least opposing something 'evil'.

Consider one example of a possible application: annotating the human genome. In June 2000 Craig Venter and Frances Collins announced jointly the completion of a 'first draft' of the human genome. This is raw data only, a huge mass of code in four letters that by itself is almost meaningless since less than 10% (possibly much less) of the information actually codes for genes. The next step is deciphering and annotating this data, figuring out where the genes are, what they code for, and how they interface with regulatory genes and which each other. It is the interesting and much more difficult task, and will probably go on for decades. How that task ultimately will be organized is still an open question, although the first generation efforts have taken place primarily in either government-sponsored or commercial settings. If my argument in this paper and the derivative hypotheses listed above are correct, this seems an obvious place where an open source process could (and perhaps should) develop.

This is one realm of knowledge creation that this paper suggests is suited to an open source process. I leave it to the reader's imagination to think of others.