Negative Scarcity And The Praxeology Of Open Source Software

Adam Reed, (Email: Areed2@CalStateLA.edu), California State University, Los Angeles

ABSTRACT

Economics, which in its 20th-century sense deals only with optimizing the exchange of values, may be generalized to praxeology (von Mises 1949,) a comprehensive science of human action. Under a generalized praxeological definition of scarcity there are contexts in which the scarcity of software becomes negative, so that a rational maximization of return on investment in developing such software occurs when the software is opened to free distribution under a GPL-equivalent license.

PRAXEOLOGY AND SCARCITY

Values are states of being that one seeks to gain or to keep (Rand 1957, 1961; 121.) Economics is the science of optimal allocation of resources for the attainment of values (Mises 1949; 3.) When a separate science of economics began in the 18th century, most of the resources used in the attainment of human values were allocated by individuals or within a household (hence the concept of, e.g., "home economics.") The study of trade and exchange, and of the production of goods and services for trade and exchange, was originally considered a mere branch of economics called *catallactics* (Whately 1831.) In the course of the 19th century, efficiencies due to division of labor led to industrialization and increasing use of resources through monetary exchange. By the end of the 19th century "economics" had become identified, in normal usage, with what had been called "catallactics." When, in the 20th century, further increases in the efficiency of production led to a shift of individual and family resources away from spending all of one's waking hours in labor for monetary exchange, and toward increased allocation of time to recreation and leisure, and a corresponding shift in the use of monetary resources toward leisure and self-realization, the term "economics" proved surprisingly resistant to a broadening of late 19th century usage. To fill the gap, Mises (1949) proposed the term *"Praxeology,"* so that "economics becomes a part , although the hitherto best elaborated part, of a more universal science, praxeology."

Exchangeable goods and services, the concern of traditional (catallactic) economics, do not lend themselves to simultaneous use by more than one person (I cannot use a computer keyboard that is being used by someone else) or (usually) for more than one purpose (I cannot use the same hour of my time both to deliver a lecture to my students and to play with my child.) Giving away some of my property, or some of my time, without adequate compensation leaves me with less value – less of what I aim to gain or keep – than I would have had otherwise. This phenomenon – that one has less value left if one gives something away – is called *scarcity.* Scarcity applies not only to material goods, but also to time – part of one's finite lifetime on Earth – used to produce things or ideas (e.g. computer software) that might then be used without compensation by others. It is because of the latter kind of scarcity that - in an economically rational society - the products of one's time and work (including intellectual work) are protected by law against uncompensated appropriation by others. Such laws recognize my right to condition the use of the use of intellectual property (such as software) produced by my work on conditions of my choice.

If the only value of software produced by my work were its exchange value (i.e. the only value considered by traditional, catallactic economics) then the only possible outcome of giving it away (or licensing its use without payment) would be to reduce the subsequent demand for (and therefore the market price of) future licenses to use the same software. Therefore the subsequent value of the product of my work to me would decrease, in line with the traditional catallactic notion of scarcity. However, when my software has value other than exchange value, it becomes possible for the value (to me) of my software to *increase* (rather than decrease) because I have made it

available to others without extrinsic payment. If scarcity is understood as a decrease in *value to me* when my work becomes available to others without payment, this would be an example of *negative* scarcity.

OPEN SOURCE AS A MECHANISM OF NEGATIVE SCARCITY

Consider anything that I make or improve for my own use. Its value to me comes primarily from its usefulness in obtaining or keeping my (other) values. That value – its utility to me – is not necessarily less if (as can be done with software) it is replicated without cost to me and used by others. Indeed, my own values may be enhanced if other people, or other enterprises, can use it also to attain their values, especially if their values coincide with or help me gain mine. The same is true of improvements that make some collection of software more useful, to me and to others, than it was before. To be sure, if I permit this I can no longer claim the market exchange value of that software. If the software is used in producing my other products, then I may also lose some of the competitive advantage I would have kept if I had kept my software to myself. On the other hand, it is possible that my benefit, from getting to use the improvements that others will eventually make to my software, will be greater than my relative loss of licensing income or of competitive advantages. This requires a license such as the *GPL* (GNU General Public License, see below) (Stallman 1989; Asay 2004; Wikipedia 2006) that protects my right to use improvements made to my software by others, and rational calculation of potential losses and benefits. The benefits, however, can be considerable (Raymond 2001.) To the extent that those benefits exceed costs, the value to me, of software that I have chosen to distribute on condition of benefiting from improvements made by recipients, will increase. Another way to say this is that, with properly conditioned open source distribution, software exhibits *negative scarcity*.

WHY RECIPIENTS WILL MAKE IMPROVEMENTS

The business case for choosing between proprietary and open-source software hinges on two sets of costs and two sets of benefits: the cost and benefits of the obtained software in its original state, and the costs and benefits of making improvements to the software (also called "maintenance".) Sellers of proprietary software are fond of pointing out that, in the experience of many organizations, expenditures on making improvements to (nominally free) open-source software often exceed the combined cost of proprietary software *and* any additional payments to the vendor for maintenance. This focus on "Total Cost of Ownership" (TOC) often obscures (or even prevents) consideration of the relative value of improvements that can be made. Yet the cost of making improvements to open-source software is often greater *because* in the case of open-source software one can make improvements *that are much more valuable* to one's business than those that can be reasonably expected from a vendor of proprietary software.

In the course of my own experience in industry, as a lead user of both proprietary and open-source software, and an occasional maintainer of the latter, it was often the case that a time-critical project could not proceed without changes to software. In the typical case, as a major customer of the proprietary software vendor (I worked for an organization within Lucent Technologies) we would receive requested changes within two to four weeks of our request. In several cases, I was able to shortcut this delay by replacing the proprietary software component with an open-source equivalent, and then coding and testing and delivering the needed changes in the space of one or two days. Not only was it possible to use open-source software to avoid weeks of delay in time-critical projects, but also we had much greater confidence in the quality of the resulting changes, since we made the improvements ourselves and were able to evaluate the quality of the code of our modifications.

I was originally surprised by the very high level of coding competence and readability that is evident in open-source code. As compared with proprietary code that was sometimes made available to us under non-disclosure agreements, and at extra cost, it was clear that the writers of open-source software expected their code to be widely read, and to be the foundation of their personal reputations in the software community. The result was a much lower incidence of obscure, abstruse, or unreliable code in open-source than in proprietary software. Open-source code was often an aesthetic pleasure to read, and nearly always easy to understand and, when needed, improve.

Improvements that may be critically needed by user organizations tend to fall into one of 5 categories:

Bug Fixes

The software does not perform as expected because of a design or coding error. Since the code is available for inspection, correction, and re-manufacturing, the error is usually found and corrected much faster than even the fastest turn-around times in vendor corrections to proprietary software.

Vulnerabilities

The software may be made to perform undesirable operations as a result of manipulation by third parties. It is corrected so that the unauthorized operations will not be performed, while other operations will continue to be performed as expected.

Flexibility And Customization Options

Open-source software is often written for a specific project of its author, and originally accepts inputs and provides outputs in formats optimized for the original author's purpose, but not optimal (or even usable) for the purposes of other users. A user organization that needs to use the software for a different purpose may add options (typically controlled by environment variables, command-line language, or a customization file) to make input and output sources and formats more flexible and more readily customized.

Additional Capabilities

It is not unusual for available open-source software to contain most of the code that a user organization otherwise would have needed to write to implement an entirely new capability. This can be achieved either by adding the new capability to one of the existing applications in an open-source package, or by re-packaging the shared code as a library and adding the new capability as a new application.

Optimizations

Because the time and effort spent in optimizing the performance of a segment of code will not pay back unless the code is invoked often enough in the context of its original application, software is written so that only the frequently repeated segments are optimized. In the context of a new application, code segments that were rarely executed in previous contexts may now get executed often enough to justify, and even require, careful optimization. The newly optimized code will also improve somewhat the performance of the software in other contexts.

OPEN-SOURCE RECIPIENTS AS TESTERS

Most users of open source software will not be interested in, or even capable of, coding improvements to the source to the source of the software. They may nevertheless contribute to negative scarcity by serving as testers. Most proprietary software is only tested by a relatively small team of in-house testers, who practice a kind of testing triage, and actually test the product in only the subset of operational contexts that they expect to encounter most often when the software is deployed to their customers. The free distribution of open-source software provides a much larger "testing team" than any vendor of proprietary software could possibly afford to hire. Proprietary HTTP web servers, for example, are typically tested by a testing team of no more than a few hundred testers. The open-source *Apache* HTTP server, in contrast, is deployed in literally millions of user organizations, all of whom serve as a giant, multifaceted testing team.

THE USERS' INCENTIVE

The open-source negative scarcity cycle is completed when user organizations feed bug reports and contributed improvements back to the originator of the software distribution. The incentive to report bugs is obvious: the bug is interfering with the user's application of the software, and reporting the bug is a precondition of having it fixed in a subsequent update or release. Similarly, the integration of improvements coded by a user organization with others' improvements and bug fixes, in subsequent updates and releases of the software, is most

efficiently accomplished by feeding the improvements back to the author or owner of the open-source software package. While it is theoretically possible to keep one's improvements to oneself, one cannot subsequently benefit from improvements made by others without either contributing one's improvements back to the author, or going through an expensive re-integration of one's improvements with the newly updated package at every subsequent update or release. These facts create an overwhelming incentive for user organizations to share their improvements with the distribution.

THE GNU PUBLIC LICENSE (GPL)

The success of the current open-source paradigm depends on the legal recognition of intellectual property in software, and the consequent ability of the author of open-source code to impose conditions – most often the Gnu Public License – on others' use of the results of her work. Before Richard Stallman's invention of the GPL, software placed in the "public domain" was available to anyone without any conditions at all. This resulted in the intellectual-property equivalent of the "tragedy of the commons:" anyone could, and some did, use public-domain code at will, but keep improvements – and the body of code incorporating the public-domain software with their improvements – secret; gaining a competitive advantage over the original author (and thus discouraging the free distribution of software) or even selling the marginally improved product back, at a price, to the organization that paid for writing most of it.

The GNU Public License obliges anyone who would re-distribute GPL-licensed software to make the source of any product that incorporates the original code openly available, and to license the result under GPL. This prevents secondary authors from discouraging original contributions, either by gaining a competitive advantage over the original author by means of his own work, or by charging any contributor a payment for the use of his own contributions. The GPL minimizes the contribution of these factors to the overall scarcity of the software. In combination with the users' incentives to share potential and actual improvements with the originator of the software, this creates the condition of negative scarcity that is now characteristic of open-source software projects.

CONSORTIA AND COLLABORATIONS

The enterprise that sponsors an open source development project typically obtains, in return for its contributions, "ownership" in the sense of control over what user-contributed improvements are incorporated in the software distribution. Large-scale projects are often shared among several enterprises that contribute the work of their employees in return for assurance that their interests are served by the direction of the project. The Apache Software Foundation, which publishes the Apache HTTP server and several other open source projects, enjoys the participation of such companies as Lotus Development, IBM Corporation, Informatica, Red Hat Inc., Sun Microsystems, Transmeta Corporation, Fujitsu Siemens Computers, Novell Inc., and Google (Apache 2005.) As sometimes happens, business praxis has overtaken economics. The author hopes that this article, and especially the present conception of *negative scarcity*, will bring praxeological theory back into the picture.

REFERENCES

1. Apache Software Foundation. *Members of the Apache Software Foundation.* http://www.apache.org/foundation/members.html 2005.
2. Asay, Matt. *The GPL: Understanding the License that Governs Linux.* http://www.novell.com/coolsolutions/feature/1532.html January 16, 2004.
3. Mises, Ludwig von. *Human Action.* New Haven: Yale University Press, 1949.
4. Rand, Ayn. *Atlas Shrugged*. New York: Random House, 1957.
5. Rand, Ayn. *For The New Intellectual.* New York: Random House, 1961.
6. Raymond, Eric S. *The Cathedral and the Bazaar*. Sebastopol, California: O'Reilly Media, Inc., 2001.
7. Stallman, Richard. *GNU General Public License Version 1*. http://www.gnu.org/copyleft/copying-1.0.html 1989.
8. Whately, *Introductory Lectures on Political Economy.* London, 1831. Cited in Mises 1949.
9. Wikipedia. *GNU General Public License*. http://en.wikipedia.org/wiki/GNU_General_Public_License 2006.